by Mulyadi Santosa
<a_mulyadi/at/softhome.net>

About the author:

Mein Name ist Mulyadi Santosa. Ich lebe in Indonesien und arbeite als freiberuflicher Autor und Berater. Meine Interessen sind Clustering, Systemsicherheit und Netzwerke. Ich führe privat auch eine eigenen Firma im Cluster−Umfeld, die cost of the shelf Cluster verkauft. Mein Schwerpunkt liegt hauptsächlich auf OpenMosix und openSSI. In meiner Freizeit entspanne ich mich beim Lesen und Sport. Du kannst mir e−mail an die Adresse a_mulyadi@softhome schicken und mit mir diskutieren.

Linux−Interna mit Syscalltracker verfolgen

Abstract:

Manchmal möchten wir unser Linux−System etwas genauer beobachten. Es gibt eine Menge Log−Programme, Einbruchs−Erkennungs−Werkzeuge, Integritätsprüfer und so weiter. Dieses Mal möchte ich Euch mit einem Mechanismus bekannt machen, der Linux auf Kernel−Ebene überwacht und mehr Betriebssicherheit und einen weiteren Umfang bietet.

Einleitung

Eines Tages verfolgte ich eine Diskussion über Clustering−Middleware auf einer Mailing−Liste. Auf einmal gab es einen Diskussionszweig, in dem eine System−Anomalität diskutiert wurde, die durch einen Kernel−Patch verursacht worden war. Und dann antwortete jemand, dass er versuchte, das Problem anhand der von der ersten Person genannten Schritte, zu reproduzieren. Diese Person benutzte ein Werkzeug namens Syscalltracker, das ihm half, das auftretende Problem festzumachen. Und ich selbst fragte mich "Was für ein Werkzeug ist dieses Syscalltracker? Was kann es leisten?" Für einen Gelegenheitsnutzer wie mich, verursachte allein der Name "Syscalltracker" einen mysteriösen Verdacht. :−)

Syscalltracker

(http://syscalltrack.sourceforge.net) ist eine Sammlung von Kernel−Modulen, die einem hilft, Systemaufrufe, die intern vom Linux−Kernel getätigt werden, zu verfolgen. Wofür ist das gut? Ganz allgemein benutzen wir es, um den Grund für ein Fehlverhalten des Systems zu verfolgen, das kaum durch gewöhnliche Überwachungs− und Debugging−Mechanismen zu lösen ist. Hier ist ein einfaches Beispiel: Stell' Dir vor, Du hast eine Konfigurationsdatei namens inetd.conf im /etc−Verzeichnis (Grundkonfiguration für den INETd−Dämon). Du hast sie so konfiguriert, dass einige System−Dämonen aktiviert und andere deaktiviert sind. Und dann startest Du inetd und erst einmal klappt alles wunderbar. Aber dann verschwindet /etc/inetd.conf plötzlich. Zum Glück hast Du eine Sicherung und kannst die Datei schnell von einer aktuellen Sicherungskopie wiederherstellen. Dann startest Du das System erneut mit dieser Konfiguration. Dieses Mal hat irgendetwas neue Zeilen in die inetd.conf hinzugefügt und einen Befehl zum Starten mysteriöser Dämonen hinzugefügt. Nun wirst Du langsam verwirrt ... "Wer hat das getan?" "Ist dies durch einen Dämon geschehen, der selbst von inetd gestartet wurde?" "Gibt es hier einen Angriff und den Versuch, mein System zu kompromittieren?". Schnell scannst du mit "cat" das System−Protokoll und startest "top" und "ps", um nach ungewöhnlichen Prozessen oder Benutzern zu suchen, aber Du findest nichts dergleichen.

Es gibt eine Lösung, um diese Art von Problem zu verfolgen. Keine 100% perfekte Lösung, aber wirksam genug, um die wichtigsten Fälle in den Griff zu bekommen. Sie basiert auf der Tatsache, dass jede Aktion und jeder Befehl, der durch die Shell, ein Anwenderprogramm oder einen Dämon ausgelöst wird (in anderen Worten: JEDER Prozess) einen oder mehrere interne Systemprozeduren ausführt, die allgemein unter dem Namen *system calls* (Systemaufrufe) bekannt sind. Versuchst Du eine Datei zu löschen? Dann rufst Du *unlink* auf. Du führst ein Shell−Skript aus? Dann muss es *exec()* oder *execute()*aufrufen. Also wird praktisch jede Aktion in Bezug auf das System direkt als ein Systemaufruf interpretiert. Dieses ist die zugrundeliegende Idee, warum ein Überwachung auf Basis der Systemaufrufe eine so starke Waffe sein kann.

Interessiert?

Dann kannst Du es ja mal versuchen. In diesem Artikel verwende ich RedHat Linux 7.3 als Basis−System. Gehe mit Deinem Browser zu http://syscalltrack.sourceforge.net und lade Dir das Paket von der Download−Seite. In diesem Artikel benutze ich syscalltrack−0.82.tar.gz, das ungefähr 500kB groß ist. Entpacke das Paket in ein Verzeichnis, beispielsweise /usr/src:

```
# tar xzvf syscalltrack−0.82.tar.gz
```

Und dann vergewissere Dich, dass Du die Quellen des Linux−Kernels in /usr/src hast.

```
# rpm −qa | grep −i kernel−source
```

oder

ls −al /usr/src/linux−2.4

Wenn einer dieser Aufrufe ein negatives Ergebnis bringt, musst Du sie erst installieren. Sie sind auf der Redhat CD (#2):

rpm −replacepkgs −Uvh /Pfad/zu/Deinem/RPM/kernel−source−2.4.18−3.i386.rpm

Achte darauf, dass Du Syscalltracker auf der Grundlager derselben Kernelversion (und anderer zusätzlicher Patches), unter der Dein Linux−System gerade läuft, installieren MUSST. Beispiel: Wenn Du einen Standard RedHat 7.3 Kernel verwendest, dann musst du sie mit den Kernel−Quellen von der RedHat CD kompilieren. Oder, wenn Du Deinen eigenen Linux Kernel verwenden willst, musst Du Syscalltracker mit den Quellen dieses Kernels kompilieren.

Neben dem Kernel Quellcode, brauchst Du noch die Kernel−Konfigurationsdatei, um die Syscalltracker−Installation zu bewerkstelligen. Versuch's mal mit dem Inhalt von /boot:

ls −al config*

Wenn es eine Ausgabe wie 'config−2.4.18−3' gibt, dann musst Du diese Datei nach /usr/src/linux−2.4 kopieren. Benenne sie um in '.config'

cp /boot/config−2.4.18−3 /usr/src /linux−2.4/.config

Wenn diese Datei aus irgendwelchen Gründen nicht in /boot steht, dann kannst Du sie aus dem Kernel−Quellverzeichnis kopieren. Sie liegt unter dem *configs* Verzeichnis. Du musst eine auswählen, die zu Deinem gerade laufenden Kernel passt, also finde Deine laufende Version mit

uname −a

heraus. Dies sollte Dir die Version Deines Kernels ausgeben. Du kannst es erraten. Nehmen wir an, die Ausgabe enthält "kernel−2.4.18−3−i386", dann musst Du die Datei kernel−2.4.18−3−i386.config kopieren.

cd /usr/src/linux−2.4
cp configs/kernel−2.4.18−3−i386.config ./.config

Nun musst Du folgendes laufen lassen:

#cd /usr/src/linux−2.4.18−3
make mrproper
make menuconfig

Nimm die Einstellungen, die Du brauchst vor und wähle save/exit. Wenn Du einen selbstkompilierten Kernel verwendest, aber Deine alte Kernel−Konfigurationsdatei nicht mehr hast, dann musst Du vorsichtig die Einstellungen rekonstruieren, um künftige Probleme zu verhindern (ich hoffe, das ist nicht der Fall :−))

Syscalltracker kompilieren

Nun haben wir alle Anforderungen vorbereitet. Nun können wir mit dem Kompilieren von Syscalltracker beginnen:

cd /usr/src/syscalltrack−0.82
./configure (or ./configure −with−linux=/path/to/your/linux/kernel/source)
make && make install

Wenn die Kompilation erfolgreich verlaufen ist, wirst Du zwei neue Module finden:

1. /lib/modules/2.4.18−3/syscalltrack−0.82/sct_rules.o

2. /lib/modules/2.4.18−3/syscalltrack−0.82/sct_hijack.o

Dieses sind die Module, die für Deine Systemüberwachung verantwortlich sind. Der Autor selbst benutzt den Ausdruck *system call hijacking (Systemaufruf−Entführung)*, was bedeutet, dass der Systemaufruf unterbrochen wird und die vorgeschriebene Arbeit getan wird, bevor die eigentliche Prozedur ausgeführt wird. Als nächstes musst Du die Module laden. Hierfür gibt es ein vorgefertigtes Skript.

sct_load (als root)

Stell' sicher, dass der Kernel die Module geladen hat, indem Du *lsmod* eingibst. Du solltest in etwa das folgende sehen:

```
Module     Size  Used by    Not tainted
sct_rules  257788   2
sct_hijack 110176   1      [sct_rules]
<&hellip;..cut&hellip;&hellip;&hellip;..>
```

Regeln

Herzlichen Glückwunsch! Du hast die Module geladen und somit läuft der Syscalltracker. Aber wir sind noch nicht fertig. Du musst nun *Regeln* schreiben, die der Syscalltracker braucht, um entsprechend arbeiten zu können. Lass' uns mit einer einfachen anfangen:

```
rule
 {
   syscall_name=unlink
   rule_name=unlink_rule1
   action
   {
     type=LOG
     log_format {%comm : %params delete by %euid --> %suid}
   }
   when=before
}
```

Jede Regel, die für Syscalltracker deklariert wird, beginnt mit dem *reservierten* Wort "rule", dem ein "{" folgt. Danach musst Du deklarieren, welchen Systemaufruf Du beobachten musst. Dies erfolgt mit dem Parameter "**syscall_name**". Es gibt eine Menge Systemaufrufe, aus denen Du wählen kannst. Um eine komplette Liste zu bekommen, schau in die Datei '/usr/local/lib/syscalltrack–0.82/syscalls.dat–2.4.18–3'. Manchmal ist es schwierig, die Bedeutung einiger Systemaufrufe zu erraten, aber es gibt auch leichte. Für jetzt greife ich mal *unlink* heraus. Dieser Systemaufruf wird jedes Mal ausgeführt, wenn jemand oder etwas versucht, eine Datei zu löschen. Ich denke, dies ist eine gute Wahl für den Anfang, die Idee ist es also, alle Löschvorgänge, die auf Deinem System geschehen, zu überwachen.

Dem Parameter "**rule_name**" muss der Name der Regel übergeben werden. Dies ist ein frei wählbarer Eintrag, schreib' einfach einen leicht zu verstehenden Namen. Ich wähle "**unlink_rule1**". In dem Abschnitt "**action**", musst Du eintragen, welche Aktion Syscalltracker ausführen soll, wenn es einen entsprechenden Systemaufruf gibt. Syscalltracker unterstützt verschiedene Aktionen, aber hier benutzen wir den Typ **LOG**. Diese Aktion wird eine Logdatei nach **/dev/log** schreiben. Laut TODO–Liste der Website gibt es Pläne, Systemaufrufe neu zu schreiben. Dies bedeutet, dass Du Systemaufrufparameter manipulieren und Deine eigenen Parameter einfügen kannst. :–)

Für die **LOG** Aktion musst Du ein Ausgabeformat definieren. Es gibt bereits eingebaute Makros, um eine umfassende Ausgabe zu erhalten

```
%ruleid -> Name der Regel, die den Systemaufruf betrifft
%sid    -> Identifikationsnummer des Systemaufrufs
%sname  -> Name des Systemaufrufs
%params -> Parameter des Systemsaufrufs
%pid    -> ID des Prozesses, der den Systemaufruf tätigt
%uid    -> Benutzerkennung, die den Systemaufruf ausführt
%euid   -> effektive Benutzerkennung, die den Systemaufruf ausführt
%suid   -> aufgezeichnete Benutzerkennung, die den Systemaufruf ausführt
%gid    -> die Benutzergruppe, von der der Benutzer den Systemaufruf ausführt
%egid   -> die effektive Benutzergruppe, von der der Benutzer den Systemaufruf ausführt
```

```
%sgid   -> die aufgezeichnete Benutzergruppe, von der der Benutzer den Systemaufruf ausführt
%comm   -> Name des Befehls, der den Systemaufruf ausführt
%retval -> Rückgabewert des Systemaufrufs. Funktioniert nur für die
           LOG Aktionction mit dem Typ "after"
```

Für dieses Beispiel habe ich das folgende geschrieben:

.log_format {%comm : %params delete by %euid −−> %suid}

Dies bedeutet: "Ich möchte jeden Befehl protokollieren, der den Systemaufruf names *unlink* ausführt, und zwar mit der effektiven und der aufgezeichneten Benutzerkennung".

Beim Parameter **when**, können wir zwischen "**before (vorher)**" und "**after (nachher)**" wählen. Der Unterschied ist klar, wenn wir "**before**" verwenden, dann wird die Protokollierung ausgeführt, bevor der Systemaufruf ausgeführt wird. Wenn wir "**after**" wählen, dann findet die Protokollierung erst nach dem Ausführen des Systemaufrufs statt.

Die Regel wird mit "}" abgeschlossen. Diese ganze Regel kann in eine normale Textdatei geschrieben werden, lass sie uns zum Beispiel "**try.conf**" nennen und in **/tmp** speichern. Als nächstes musst Du diese Regel in Syscalltracker einbinden.

sct_config upload /tmp/try.conf

Wenn die Regel richtig geschrieben wurde, erhältst Du die Meldung "Successfully uploaded rules from file '/tmp/try.conf' ".

OK, alles hat geklappt. Nun kommt die Testphase. Wechsle auf die Konsole, zum Beispiel xterm innerhalb von xwindow. Auf einer Konsole beobachtest Du Systcalltrackers Protokoll:

sctlog

Nun wirst Du bald eine Ausgabe als Ergebnis der Unterbrechung in der Ausführung des Systemaufrufs sehen, wenn Deine Regel greift. Gib auf einer anderen Konsole etwas wie das folgende ein:

```
# cd /tmp
# touch ./dummy
# rm ./dummy
```

Beim Verwenden der oben genannten Regel, wirst Du nun diese Ausgabe auf sctlog sehen:

```
"rm" : "./dummy" delete by 0 --> 0
```

Von dieser Meldung kannst Du darauf schließen, das dieses passiert:

Der Befehl "**rm**" mit dem Parameter "**./dummy** führt den Systemaufruf *unlink()* aus. Oder, in anderen Worten, wird rm benutzt, um eine Datei zu löschen. Dieser Befehl verwendet eine effektive Benutzerkennung, die 0 (also Root) ist."

Hier folgt eine weitere Beispielregel

```
rule
{
   syscall_name = unlink
   rule_name = prevent_delete
   filter_expression {PARAMS[1]=="/etc/passwd" && UID == 0}
   action {
     type = FAIL
     error_code = -1
   }
   when = before
}
```

Dies ist unserem ersten Beispiel ähnlich, aber hier benutzen wir die **FAIL** Aktion. Exklusiv für **FAIL** müssen wir einen Rückgabewert für den unterbrochenen Systemaufruf definieren. Hier benutze ich "−1" also "operation not permitted" (Ausführung nicht erlaubt). Die komplette Liste dieser Nummern kann unter /usr/include/asm/errno.h eingesehen werden.

In der Zeile, die "filter expression" enthält, definiere ich eine Bedingung, unter der die Überprüfung stattfindet, wenn der erste Parameter des Systemaufrufs "**/etc/passwd**" ist. Hierfür brauchen wir die Variable **PARAMS**. Merke: Jeder Abschnitt hat seine eigenen geforderten Parameter. Diese Überprüfung ist noch nicht perfekt, denn es könnte jemand etwas wie "cd /etc && rm −f ./passwd" benutzen. Aber für einen Anfang ist es okay. Wir prüfen auch, ob die UID gleich 0, also root, ist.

Füge diese Regel Deiner ersten hinzu und lade neu:

```
# sct_config delete
# sct_config upload /tmp/try.conf
```

Achte darauf, dass die Reihenfolge der Regeln wichtig ist. Wenn Du "prevent_delete" vor "unlink_rule1" definierst, passiert folgendes: Wenn Du:

rm −f /etc/passwd

eingibst, wird zuerst die Regel "prevent_delete" ausgeführt und die gesamte Aktion schlägt fehl. Die Regel "unlink_rule1" wird ignoriert. Aber wenn Du die Reihenfolge vertauschst ("unlink_rule1" vor "prevent_delete") erhältst Du nur den Protokolleintrag, ohne die Aktion zu stoppen!

Es gibt einen weiteren Systemaufruf, dessen Überwachung interessant ist. Er heißt *ptrace*. In "man ptrace" kannst Du lernen, dass dieser Systemaufruf verwendet wird, um die Ausführung und Kontrolle der Ausführung eines anderen Programms zu beobachten und zu steuern. In guten Händen kann *ptrace* ein hilfreiches Werkzeug für debugging−Zwecke sein, aber in den falschen Händen kann es dazu benutzt werden, um Sicherheitslöcher zu analysieren und auszunutzen. Lass uns also eine Regel hinzufügen, es zu protokollieren.

Um dies zu tun, verwende eine Regel wie diese:

```
rule
{
    syscall_name=ptrace
    rule_name=ptrace_rule1
    action {
        type=LOG
      log_format {%comm : %params issued ptrace by %euid --> %suid}
  }
    when=before
}
```

Beachte, dass wir prtrace als zu überwachenden Systemaufruf definieren. Um dies zu testen, benutze das *strace* Programm. Lade zunächst die oben genannte Regel in Syscalltracker und starte dann sctlog. Dann lasse strace gegen *ls* laufen, zum Beispiel so:

strace /bin/ls

In sctlog solltest Du nun einige Zeilen, wie die folgenden bekommen:

```
"strace" : 3, 2019, 24, -1073748200 issued ptrace   by 0 --> 0
"strace" : 24, 2019, 1, 0 issued ptrace   by 0 --> 0
"strace" : 3, 2019, 44, -1073748200 issued ptrace   by 0 --> 0
"strace" : 3, 2019, 24, -1073748200 issued ptrace   by 0 --> 0
"strace" : 3, 2019, 0, -1073748216 issued ptrace   by 0 --> 0
"strace" : 7, 2019, 1, 0 issued ptrace   by 0 --> 0
```

Für die, die bisher strace noch nicht kannten: Dies ist ein Werkzeug (und zwar ein mächtiges), das Systemaufrufe verfolgt, die innerhalb einer ausführbaren Datei getätigt werden. Strace benutzt intern ptrace, um sich selbst in das Zielprogramm zu hängen, so dass dieses verfolgt werden kann. Tatsächlich sind strace und Syscalltracker das ideale Doppelpack für die Überwachung von System und Dateien, daher denke ich, dass es diese Erwähnung wert ist. RedHat 7.378/9 haben es schon. Installiere einfach das RPM (hier für RedHat 7.3)

rpm –Uvh /path/to/your/Redhat/RPM/strace–4.4–4.i386.rpm

Nun bist Du einen Schritt weiter bei der Diagnose Deines Systems. Syscalltracker gibt dem Nutzer Flexibiliät und Du kannst etwas über Systemaufrufe lernen. Übrigens, wenn Du Dir die geladenen Regeln ansehen möchtest, gib einfach das folgende ein:

sct_config download

Um alle geladenen Regeln zu löschen, tippe:

sct_config delete

Und schließlich kannst Du Syscalltracker aus dem Speicher löschen, wenn Du es nicht mehr brauchst:

```
# sct_unload
```

Es besteht die Möglichkeit, dass dieser Befehl darin scheitert, Syscalltracker zu entladen und die Warnung "Device or resource busy" ausgibt. Wenn dies passiert, kann es sein, dass Syscalltracker gerade läuft. Syscalltracker ist sicher für das System und es wird das laufende System nicht über die Maßen beanspruchen (solange Du nicht gerade Tonnen von Regeln hinzufügst :–)). Das Fazit: Lass Syscalltracker einfach auf dem Kernel sitzen und seine Arbeit tun. Wenn Du ihm genug Regeln gibst, kannst Du anfangen, Dein System näher zu überwachen.
