LinuxFocus article number 320
http://linuxfocus.org

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

About the author:

My name is Mulyadi Santosa. I live in Indonesia and work as freelance writer and consultant. My interest is clustering, system security and networking. I also run small privately own company in clustering field aiming for selling cost-of-the-shelf clustering. OpenMosix and openSSI really drawn my focus. In my spare time, reading and sport are great leisure. You can send me e-mail to a_mulyadi@softhome and discuss about it

Tracking Linux Internals with Syscalltracker

Abstract:

Sometimes we want to watch our Linux system more closely. The are many of logging prograns, instrusion detection tools, integrity checking tools and so on. This time I want to introduce you to a mechanism to watch Linux at kernel level, which offer more reliability and more broader coverage.

Introduction

One day I hang on mailing list discussing about clustering middleware tool. Accidentally there was a thread discussing system anomaly caused by a kernel patch. And then a person replied that he try to recontruct the problem according to steps reported by first person. This person uses a tool called Syscalltracker to help him pinning the problem that arises. And (for myself) I wonder "What kind of tool is Syscaltracker? What is its capability? " For casual user like me, the name "Syscalltracker" alone left me with mysterious suspect :-)

Syscalltracker

(http://syscalltrack.sourceforge.net) is a collection of kernel modules to help one tracing system calls issued internally by Linux kernel. What is it for? Generally we can use it to trace the cause of some system misbehaviour that is hard to solve by ordinary tracing or debugging mechanism. Here is a simple example: suppose you have configuration file in /etc directory named inetd.conf (base configuration for INETd daemon). You had configured it to run several system daemon and disabled other. And then you start up inetd and everything went fine at first place. But, suddenly /etc/inetd.conf dissapeared. Luckily you have the backup and you quickly restore it from fresh backup. And then you restart inetd base on newly configuration. This time something has add newlines to inetd.conf and inserted a command to start mysterious daemons. You got confused... "who did this?" "Is it caused by a daemon started by inetd itself?" "Is there any exploit trying to compromised my system?". Quickly you "cat" over the system log, fire up "top" and "ps" to look for unusual process or user that exist, but none of them is visible.

There is a solution for tracing this kind of problem. Not a 100% perfect solution but sufficient enough for handling major cases. It is based on the fact that every action or command issued from a shell, user program or daemon (in other word, ANY process) must be executing one or more internal system procedure widely known as *system calls*. Are you trying to delete a file? It means you call *unlink*. You ran a shell script? Then it must be calling *exec()* or *execve()*. So virtually every action related to the system is directly interpreted as system call. This is the basic idea why tracking based on system calls can be a tough weapon.

Interested?

Then you can give it a try. In this article, I use Redhat Linux 7.3 as a base system. Just point your browser to http://syscalltrack.sourceforge.net and download the package from the download section. For this article I use syscalltrack-0.82.tar.gz with approximately 500kB in size. Unpack this package in a directory, let’s assume in /usr/src:

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

And then verify that you have the Linux kernel source code at /usr/src.

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

OR

```
# ls -al /usr/src/linux-2.4
```

If one of them gives a negative result, then you need to install them. It is on Redhat CD (#2):

```
# rpm -replacepkgs -Uvh /path/to/your/RPM/kernel-source-2.4.18-3.i386.rpm
```

Pay attention that you HAVE TO compile Syscaltracker based on same kernel version (and other additional patches) that your Linux system is currently running. For example: if you use a standard RedHat 7.3 kernel, then you have to compile with the kernel source from the Redhat CD. Or if you want to use your own Linux kernel, later on the you must compile Syscalltracker based on this kernel source.

Besides kernel source code, to ease Syscalltracker compilation, you also need the kernel configuration file from Redhat. Try to check the content of /boot:

```
# ls -al config*
```

If there is an output like ’config-2.4.18-3’ then you need to copy this file to /usr/src/linux-2.4. Rename it to ’.config’

```
# cp /boot/config-2.4.18-3 /usr/src /linux-2.4/.config
```

If this file is somehow missing, the you can copy it from kernel source directory. It is located under the *configs* directory. You must pick one that matches your currently running kernel. So first find it out with

```
# uname -a
```

What you are running. It should display the version of the kernel. You can guess it. Suppose the output contains "kernel-2.4.18-3-i386" then you need yo copy kernel-2.4.18-3-i386.config

```
# cd /usr/src/linux-2.4
# cp configs/kernel-2.4.18-3-i386.config ./.config
```

Now you just have to run

```
#cd /usr/src/linux-2.4.18-3
# make mrproper
# make menuconfig
```

Do the settings of your need and then save/exit. If you use a self compiled kernel but lost you old kernel config file then you must carefully reconstruct it to avoid future problems (I hope not :-))

Compile Syscalltracker

Until now, we have prepared all of the requirements. We can go into Syscalltracker compilation:

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

if the compilation is successful, you will found two new modules:

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

Those are the modules that are responsible for you system tracing. The author itself uses the term *system call hijacking*, means intercepting the sytem call and do the pre scripted work before executing original the procedure. Next you need to load them. There is a built script to do it

```
# sct_load (as root)
```

Confirm that the kernel loaded it using *lsmod*. You should see something like:

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

Rules

Congratulations! You have loaded the module and thus you are running the Syscalltracker. But it is not over. You still need to write *rules* needed by Syscalltracker to work properly. Let’s begin with a simple one

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

Every rule declared for Syscalltracker begins with the *reserved word* "rule" followed by an "{". After this you must declare which system call you need to observe using the parameter "**syscall_name**". There are many system calls you can from pick. For a complete list try to look into the file '/usr/local/lib/syscalltrack-0.82/syscalls.dat-2.4.18-3. The meaning of some system calls is sometimes hard guess, but there are also easy ones. For now I'll pick *unlink()*. This system call is excuted every time someone or something trys to delete a file. I think this is a good choice for start, so the idea is to watch the deletions that happen on our system.

On the parameter "**rule_name**", you must supply the name for the rule. This is a free-to-choose entry, just write an easy-to-understand name. I choose "**unlink_rule1**". At the section "**action**", you need to write what action Syscalltraker does whenever there is match for system call's name. Syscalltracker is supporting some actions, but here we use **LOG** type. This action will write the log onto **/dev/log**. According to the TODO list on the website, there is plan to do system call rewriting. This means you can manipulate system call parameters and inject your own parameter :-)

For **LOG** action, you need to define output format. There are built in macros to get detailed output.

```
%ruleid -> rule name that matches intercepted system call
%sid    -> system call identification number
%sname  -> system call name
%params -> system call parameter
%pid    -> ID from process that call the system call
%uid    -> user ID which executing the system call
%euid   -> effective user id that executing system call
%suid   -> recorded user id that executing the system call
%gid    -> the group id from user that execute this system call
%egid   -> effective group id from user that run the system call
%sgid   -> recorded group id of the user that ran the system call
%comm   -> name of command that executed the system call
%retval -> return value of system call. Only works
           for LOG action with type "after"
```

For this example I wrote

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

It means "I want to log every command that is executing the system call named *unlink*, with the effetive user id and recorded user id"

In parameter **when**, we can pick between "**before**" or "**after**". The difference is clear, if we use "**before**" then the recording will be done before the system call is executed. If we use "**after**" then the recording will take place after the system call has been executed.

Close, the rule with "}" . This whole rule can be written inside normal text file, for example let’s name it "**try.conf**" and save it on **/tmp**. Next you need to load this rule into Syscalltracker

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

If the rule is written correctly, you will get the output "Successfully uploaded rules from file ’/tmp/try.conf’ ".

OK, all things went fine. Now comes the testing phase. Start to console, let say xterm inside Xwindow. On one console, watch the Syscalltracker’s log using

```
# sctlog
```

Soon you will see the output as the result of intercept mechanism of system call if one matches your rule. On another console, do something like:

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

Using the above rule, you will likely get this output on sctlog

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

From this output, you will soon learn that this happens:

The command "**rm**" with the parameter "**./dummy** is executing the system call *unlink()*. Or in other words rm was used to delete a file. This command is using effective user id equal to 0 (or root)"

Here is another example rule

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

This is similar to our first example, but here we use the **FAIL** action. Exclusively for **FAIL**, we must define the return value for the intercepted system call. Here I use "-1" or "operation not permitted". The comlete list of this numbers can be found at /usr/include/asm/errno.h.

On the line containing "filter expression", I define a condition for which the checking is done if the first parameter (of system call) equals to "**/etc/passwd**". This is why we need the **PARAMS** variable. Note: every section has its own required parameters. Also, this checking is not perfect because one can use something like "cd /etc && rm -f ./passwd". But this is OK to start. We also check if the UID equals to 0 (root).

Add this rule to your previous rule file and reload :

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

Pay attention that rule sequence is important. If you declare rule "prevent_delete" before "unlink_rule1", then if you do :

# rm -f /etc/passwd

you will match "prevent_delete" first and the whole action will fail. The "unlink_rule1" will be ignored. But if you swap it ("unlink_rule1" before "prevent_delete") then you just get the log without stopping the action !

There is another system call that is interesting to be observed. It is called *ptrace*. From "man ptrace", you can learn that this system call is used to observe and control the execution and control the execution of another program. In good hands, this *ptrace* can be handy tool for debugging, but in the wrong hand, it can be used for analyzing security holes and exploit them. So let’s add the rule to log them.

In order to do so, just use rule like this

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

Notice that we declare ptrace as system call to be watched. To test this, use the *strace* program to try the rule. First load above rule into syscalltracker and run sctlog. Then run strace against *ls* for example:

# strace /bin/ls

In sctlog, you should get several line like these:

```
"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
```

For those who did not know strace, this is a tool (a powerful one) to trace system call issued inside a executable file. Strace internally is using ptrace to hook itself into a target program so it can trace it. Actually, strace and Syscalltracker are an ideal combo for system and file auditing, so I think it is worth to mention it. Redhat 7.3/8/9 already has it. Just install the RPM (here Redhat 7.3):

```
# rpm -Uvh /path/to/your/Redhat/RPM/strace-4.4-4.i386.rpm
```

Now you are one step further in diagnosing your system. Syscalltracker gives the user flexibility and you can learn about system calls. In addition, if you want to view the rules that have been loaded, just do:

```
# sct_config download
```

To erase all rules that have been loaded, type:

```
# sct_config delete
```

At last, if you don’t need Syscalltracker anymore, you can remove it from memory :

```
# sct_unload
```

There is a possibility that this command fails to remove Syscalltracker with the warning "Device or resource busy". If this happens, it is possible that Syscalltracker is running. Just let it work for a while and try again. Syscalltracker is safe for the system and it won’t add excessive load to the running system (unless you add tons of rules :-)). The conclusion: Just let Syscalltracker sit on the kernel and do the work. Giving it enough rules, you can begin to watch your system more closely.