CS3210 Project 2 -- A syscall interposer to detect malicious employee behavior

Authors: Ben Chan and Robert Harrison
11/1/2012

Overview:

As this is to be done via the LKM, we can alter the system call table and insert monitoring command to gather information from the syscall.
Among nearly 300 syscall currently available inside the linux kernel, we have picked the following syscalls (attached below) to monitor as:
(1) Monitoring all syscalls will produce too many noise messages, and reduce the chance to locate suspicious behavior.
(2) Picked syscalls can monitor most of the possible ways to copy out files, whether valuable or not.

These syscall will be recorded:

fork
open
creat
execve
mount
access
readlink
mmap
ioperm
setuid
setreuid
vfork
pread
setresuid
mremap
fdatasync
fsync
readv
setfsuid

Assumption:

Assume there is a firewall to protect outside world directly getting data from any drive. And all users (except the admin) do not have the root privileges. Then, all the network file transfers will be recorded as they need to 'open' the file (or similar syscall) to copy files.

Flow of the LKM:

Inserting the module (Init):

(1) Searching through the memory page to locate sys_call_table (as it may different memory addresses on different machine even with the same kernel version)

(2) Create a file under /proc directory for retrieving data using user program.

(3) Set the procfs property to use seq_file for large data outflow.

(4) Disable memory page protection to overwrite original syscall.

(5) Exchange original syscall with our own implemented syscall and keep the old one for calling and restoring later.

(6) Enable memory page protection to prevent others screwing up the memory page. Done.

During any modified syscall is called:

(1) Users called the syscall and get captured by the system.

(2) System will switch to kernel mode and interrupt.

(3) System will look up the sys_call_table to locate the corresponding syscall function, which some of them will be our own implementation.

(4) System called that function and that function will capture the timestamp, process id and arguments for that syscall.

(5) These info will be wrapped by a struct called msg. And stored under the msg

() The function will call original one to continue and return.

() Once finished, system switch back to user mode and continue to run the user program.

Regular log dump (to be done by syscallLogPull.c)

(1) Link up the procfs file (Error will occur if program executer is not the root).

(2) Create a file with current timestamp (in seconds) under the /var/log/sclog directory (Again, error will occur if program executer is not the root).

(3) Keep reading from the proc file and put the log messages into the log file we have created before.

(4) Output summary for the log file. Done.

Removing the module (Exit):

(1) Check if the table is modified (by a boolean value defined inside the module). If not, done. Otherwise, continue.

(2) Replaced the all modified syscall with original one (saved before).

(3) Remove all messages inside the msg_head. Report if any.

(4) Remove the proc file under the /proc directory. Done.

Design choice:

(1) How to decide which syscall to intercept

It is chosen based on how users may export valuable files to outside world with syscall.

(2) How to work with sys_call_table

After Linux Kernel Version goes after 2.6, there is no direct exported sys_call_table. So, we have to find a way to locate the sys_call_table and modify the functions that we have picked for monitor.

There are two choices:

1. Locate the address from /boot/System.map

Pros:

Precise address reference.

Cons:

If the sys_call_table is not included in the System.map, then it cannot be looked up. Have to be careful with the table not to screw it up. And the map is large, takes time to lookup.

---

2. Search through the memory page and check if the reference is matched with a pre-defined syscall reference, e.g. sys_open.

Pros:

Able to locate address reference. Fast lookup.

Cons:

The pre-defined syscall reference, as mentioned by the linux kernel builder, is going to be removed from the linux kernel. So, the module may not work (and of course it will not work as the syscall function prototype is changed among different version of the kernel).

We picked 2. over 1. as it is a safer method, and the look up speed is fast.

(3) How to modify the syscall

We use function wrapper for each picked syscall. All the wrapper does is to record the process id, timestamp and any printable arguments and call the original syscall function.

But, syscall definition may change when kernel keeps evolving, this module may not be as portability as it should be. But with constant effort on upgrading this module, the effect of this can be minimized.

(4) How to log the information from the modules to storage device

There are several choices to achieve this

1. writing files inside the module;
2. printk those messages and then store it by running dmesg;
3. use procfs to transfer the message when a root program reads it, then cron the program to constantly run

In all choices we have, we choose to use the procfs method. First, io action inside kernel module is a bad thing (ref: http://www.linuxjournal.com/article/8110).

Second, we can decide when to get the message buffer if we do not do the cron job. More importantly, the program itself can keep track which messages they have already receive from the procfs. We believe there will have obvious performance degrade as messages are stored inside the linked lists of the proc file.

---Testing and Performance Analysis---

To test out our design, we wrote a simple program that will copy a file 100 times using fopen and fread methods. Since we implemented the interception of each sys_call identically, we felt the interception times would be very similar so there was no need to copy files using the other methods. When we collected 10 different times that our copy file test took to run without the sys_call_log module loaded into the kernel we got the following:

.04874 seconds
.03905
.03788
.07089
.03863
.03868
.05909
.03817
.19530
.04622

avg: .06127 seconds

Now with lkm loaded:

.46104
.29018
.20310
.40750
.78828
.26094
.21353
.25243
.28479
.27882

avg: .34410 seconds

As you can see, the average time with our interceptor module loaded into the kernel was almost 6 times slower. This can be improved by directly printk the message to the console and save it for later use instead of storing inside the procfs.