S.A.S.U.K.E. - Rootkit Detector Group 4

TUM
Chair of IT Security
Rootkit Programming WS2014/15

Martin Herrman Gurusiddesha Chandrasekhara

January 26, 2015

Contents

1 Introduction

S.A.S.U.K.E. stands for Search And Subvert User-manipulated Kernel Exploits. To detect the rootkits of the other groups we used two different approaches: detection from kernel-space and detection from user-space. The reason for implenting two different methods was that while detection from within the kernel is much easier than from user-space, not everyone will be able to write, use, or understand LKMs. This way we can also see which rootkits hide very good (no easy way of detection from user-space) or not so good (easily detected from user-space).

Our kernel-mode detection program consists of a single kernel module. Once inserted, it will run multiple tests and write a detection log to a file. Using this we were able to detect every rootkit in at least one way (most in multiple ways).

Our user-mode detection program, on the other hand, consists of a shell script and a c program. While it is expected to be run with root privileges, it will not try to insert any kernel code to accomplish its task.

Chapter 2 will describe the details of the implementation. Chapter 3 will give an overview over the different rootkits and describe which tests the failed and which they passed.

2 Design of S.A.S.U.K.E.

2.1 Loadable Kernel Module

There are a few requirements for building the Detection LKM:

- a sane kernel build environment exists
- the *System.map* file is present
- the kernel version is either 3.16.4 or at least reasonably similar

Once these are met, one can just use `make` inside the source folder to build the module and insert it with `insmod`. A log of the detection procedure can be found in the file `/sasuke.log` after the LKM has been successfully inserted.

The following sections will now describe the different approaches used to detect rootkits.

2.1.1 Detecting manipulation of system calls

Because the hooking of system calls is something you can expect pretty much every rootkit to do, this was one of the first features to be implemented. There are several ways to accomplish this, many of which our tool will be able to detect.

The simplest way is to overwrite the pointers to different system calls in the system call table. Detecting this is very easy if you can rely on a valid *System.map* file (for limitations see Chapter 4). Our tool simply compares the pointer for a specific system call in the system call table to the pointer which is present in the *System.map* file of this kernel. If they do not match, a malicious manipulation of the system call table is very likely. The following code excerpt shows this check for the `read` system call:

```
#include "sysmap.h"

void **sys_call_table = (void *) sysmap_sys_call_table;

if((void *)sys_call_table[__NR_read] == (void *) sysmap_sys_read) {
        strncpy(message, "read␣␣␣␣␣␣␣␣-␣OK\n", 64);
        write_to_file(message, strlen(message));
} else {
        hooked_syscalls++;
        strncpy(message, "read␣␣␣␣␣␣␣␣-␣NOT␣OK!\n", 64);
        write_to_file(message, strlen(message));
}
```

Even though this detection method is very simple, nearly all of the rootkits tested could be detected this way.

But of course "nearly all" is not good enough in the case of security. For this reason, another group of system call hooking methods is checked for: the direct manipulation of instructions. An attacker can overwrite the first few instructions of the system call to direct it elsewhere, e.g. his own function. This can again be detected by a simple comparison if one knows which instructions are supposed to be executed at the beginning of the manipulated system call, as is illustrated by the following example which is again using the `read` system call:

```
#include "sysmap.h"

unsigned int tmp[4] = { 0x00000000, 0x00000000,
                0x00000000, 0x00000000 };
unsigned int original_read[4] = { 0xD5894855, 0xEC834853,
                0x74894848, 0x61E80824 };

if(memcmp(original_read, (void *) sysmap_sys_read,
                sizeof(unsigned int)*4) != 0) {

        memcpy(tmp, (void *) sysmap_sys_read, sizeof(unsigned int)*4);
        memset(message, 0, 128);
        sprintf(message, "read␣␣␣␣␣␣␣␣␣␣␣␣␣-␣%08X␣%08X␣%08X␣%08X\n",
                tmp[0], tmp[1], tmp[2], tmp[3]);
        memset(message+127, '\0', 1);
        write_to_file(message, strlen(message));
        hooked_syscalls++;

} else {

        strncpy(message, "read␣␣␣␣␣␣␣␣␣␣␣␣␣-␣OK\n", 64);
        write_to_file(message, strlen(message));

}
```

The advantage of using this technique is that it can also be used for other kernel functions which are not referred to in the system call table. Examples for this are `packet_rcv`, `packet_rcv_spkt`, and `tpacket_rcv` which are commonly manipulated to hide packets from applications like `tcpdump`.

A sample output of our LKM for those two features looks like this:

```
[Checking the pointers in the system call table...]
read       - NOT OK!
getdents   - NOT OK!
getdents64 - OK
recvmsg    - NOT OK!
open       - OK
close      - OK
readlink   - OK
readlinkat - OK
kill       - OK
There are 3 manipulated pointers in the system call table.

[Checking the first bytes of some system calls and other important functions...]
read            - OK
getdents        - OK
getdents64      - OK
recvmsg         - OK
```

```
open            - OK
close           - OK
readlink        - OK
readlinkat      - OK
kill            - OK
packet_rcv      - 30BDB968 5441C3A0 89485355 EC8348FB
packet_rcv_spkt - 30BF2168 8948C3A0 8A5241FB 8B4C7D47
tpacket_rcv     - 30BE6D68 5441C3A0 89485355 EC8348FB
There are 3 manipulated functions.
```

As you can easily see, the pointers to the `read`, `getdents`, and `recvmsg` system calls as well as the instructions of the `packet_rcv`, `packet_rcv_spkt`, and `tpacket_rcv` functions have been manipulated. In the latter case the instructions present are displayed for further manual analysis.

2.1.2 Detecting hidden processes

While a manipulated `getdents` system call may be indicative of hidden processes (the `ps` command uses the *proc*-filesystem), there are other ways to further analyse this. The kernel provides the macro `for_each_process` which can be used to iterate every process that is getting scheduled. If there is a process that is displayed in the kernel but not inside user-space, it can be assumed that there is something malicious happening.

The following code is an excerpt from our detection of hidden processes:

```
#include <linux/sched.h>

struct task_struct *task;

for_each_process(task) {
        memset(message, 0, 128);
        sprintf(message, "[%05d] %s\n", task->pid, task->comm);
        write_to_file(message, strlen(message));
        procs++;
}
```

After looping, the number of detected processes is also written to the log file. This way the user can easily compare the number to the one he gets from user-space programs such as `ps`. They will usually not match, though. The reason for this is that the LKM will also display an `insmod` process, because this is all happening while the module is still being inserted. The user-space count will also include the processes that are used for counting (like `ps ax` and `wc -l`). After removing all processes which would only be present in one of the two checks, the user is left with two process counts. If they do not match maliciously hidden processes can safely be assumed.

2.1.3 Detecting netfilter hooks

A very easy and clean way to implement port knocking is using the netfilter API already provided by the kernel. As easy as it is to use it is also to detect. The reason for this is that the kernel exports the `nf_hooks` symbol which can be used to list all present hooks. If a standard kernel is used (as is the case for all virtual machines used in this course), there should be no hooks present. Because of this one can assume each hook in this situation to be hostile. The following code excerpt checks for hooks:

```
#include <linux/list.h>
#include <linux/netfilter.h>

struct list_head *cursor;
struct nf_hook_ops *cur;
struct module *mod;

/* iterate all netfilter hooks */
```

```
list_for_each(cursor, &nf_hooks[PF_INET][NF_INET_LOCAL_IN]) {
        cur = list_entry(cursor, struct nf_hook_ops, list);
        mod = cur->owner;

        memset(message, 0, 128);
        sprintf(message, "[%s] 0x%08lX\n", mod->name, (unsigned long) cur->hook);
        write_to_file(message, strlen(message));

        netfilter_hooks++;
}
```

PF_INET specifies that only hooks to the IPv4 stack should be displayed if they are coming to or from this machine (NF_INET_LOCAL_IN).

The fact that there is a field containing the owner is extremely interesting. If a careless attacker were to set this correctly, it would be a huge vulnerability. This field contains a copy of the `struct module` which uniquely identifies a module within the kernel. Using this we would be able to insert a hidden module back into the appropriate kernel data structures and unload it cleanly using the `rmmod` command.

2.1.4 Hidden LKM detection

Detection of the hidden LKM depends how well it's hidden from the kernel data structures. In order to hide it from `/sys/module/` one can remove it from the list of `struct module *` and the list of `struct kobject *`, remove it from the `kern_fs` tree or just hide the file. If one just hides the file, we can simply get the rootkit from both the `kern_fs` as well as the list of `struct kobject *`. If one just deletes the `kern_fs`, we can still find the hidden module in the list of `struct kobject *`.

2.2 Detection from user-space

We also wrote a tool which can detect rootkits from user-space. It's functionality is limited to two different checks but performs those extremely well. Furthermore, because of its simplicity, it is able to run on almost any Linux system. It can be used to detect hidden processes as well as hidden sockets. It consists of a helper programm written in C and a shell script which does most of the work.

2.2.1 C programm for TCP socket detection

To detect hidden sockets the helper program is necessary. It expects a single integer as parameter and interprets this as TCP port to bind to. If another process is already listening on this socket (whether it is hidden or not), binding to it will fail. This program is being compiled and called by our main shell script.

2.2.2 Shell script

The shell script first checks whether it is running with root privileges. Afterwards it uses the command `kill -0` on every possible PID. The reason for this is that while a process might not be shown in the *proc*-filesystem, it is usually still able to receive signals. For each detected process it then checks whether there is a corresponding folder in the *proc*-filesystem. If this is not the case, its PID and the instructions used to start the process are printed:

```
PID_MAX=$(cat /proc/sys/kernel/pid_max)

for pid in `seq 0 $PID_MAX`; do
        kill -0 $pid 1> /dev/null 2> /dev/null

        if [ $? -eq 0 ]; then
                if [ $(ls /proc | grep -o $pid | wc -l) != "1" ]; then
                        echo "$pid $(cat /proc/$pid/cmdline)"
                fi
```

```
        fi
done;
```

Even though this may sound fool-proof, there is a catch: maliciously hidden processes are not the only processes which are detected. Depending on the system configuration, there might be a varying number of processes that are essentially hidden. The command `kill -0` will work on them and they will have a directory within *proc* that is not visible. Gnome is a very good example for this. It causes a lot of such processes to be present. Usually a maliciously hidden process will stand out, though. They tend to be towards the end of this list (as they are usually started after the system is fully booted) and were in this case always providing remote access, making them easily detectable.

To show all opened TCP sockets we are using the helper program, as was already stated before. This program is written in C and compiled by the shell script on demand. The shell script then iterates each possible port and instructs the program to try to bind to it:

```
cc -o check_sockets check_sockets.c > /dev/null && echo "[OK]"

if [ $? -eq 1 ]; then
        echo "Compilation␣failed!"
        exit 1
fi

for port in `seq 1 65535`; do
        ./check_sockets $port 1> /dev/null 2> /dev/null
        if [ $? -eq 1 ]; then
                echo "$port␣is␣in␣use"
        fi
done;

rm ./check_sockets
```

The results are not filtered for known "good" ports. It is solely the users responsibility to check whether they are maliciously open or not. This script, of course, takes some time to terminate, as it has to check every possible PID and every possible TCP port manually.

3 Detection

We ran our detection tools on the virtual machine of each group. We also ran it on our own group to be able to compare it to the other rootkits.

3.1 Group 1

We were not able to compile our code at all on this groups virtual machine. No tests were run.

3.2 Group 2

The machine of group two was demonstrating some weird behavior (e.g. `ip` command not working) which immediately indicated tampering. Trying to connect to the machine using `scp` was also extremely slow.

The user-mode checks revealed the presence of a hidden `sshd`-process with PID `2842` which is listening on port `5167`. After further analysis using our LKM, further evidence of a rootkit were detected. The pointers to the `read` and `recvmsg` system calls were manipulated. Furthermore, the first instructions of the `packet_rcv`, `packet_rcv_spkt`, and `tpacket_rcv` functions were also manipulated.

After looking through the *kernfs* we were able to find an entry which stood out: `chytryroot`. We determined that while they removed their rootkit from the list of modules, they neglected to remove it from the *kernfs*, too.

3.3 Group 3

The machine of group three was also demonstrating some weird behavior (memory issues). Due to this we were not able to run our user-mode tool on their updated version. On their initial version, however, we were able to run it and determined that they are hiding a `sshd` process with PID `2446` which is listening on port `22`. After further analysis using our LKM, further evidence of a rootkit were detected. The first few instructions of `packet_rcv`, `packet_rcv_spkt`, and `tpacket_rcv` functions were manipulated. We were not able to detect any hooked system calls, however. We also found a Netfilter hook used for port knocking.

3.4 Group 4

We were able to detect our own rootkit using our tools. The user-mode script showed the hidden `dropbear` process (PID `3187`) which is listening on TCP port `9999`. Using the LKM we were able to detect three manipulated pointers in the system call table. The manipulated system calls where `read`, `getdents`, and `recvmsg`. We furthermore found another three functions which had manipulated instructions: `packet_rcv`, `packet_rcv_spkt`, and `tpacket_rcv`. We found one netfilter hook for IPv4 packets destined for or originating from the virtual machine. We found the name of the hidden module (`narooto`) inside the list of `kobjects`.

3.5 Group 5

This groups virtual machine was also behaving weirdly. Changing the used DNS server in `/etc/resolv.conf` to `8.8.8.8` and then using ping caused a kernel panic. Furthermore `ssh` and `scp` were not working.

The user-mode checks revealed that they are using a `nc` server (PID `2529`) listening on TCP port `4321` as a remote shell. They also use a shell script (PID `2515`) to manage this. Both processes are hidden. The shell script itself is hidden from commands like `ls`. It's location is `/root/asdfghja.sh`.

Using our LKM we were able to further determine that they manipulated the first instructions of the `read`, `recvmsg`, `packet_rcv_spkt`, and `tpacket_rcv` functions/system calls. We also detected that they are using a netfilter hook for their port knocking.

3.6 Group 6

The user-mode checks revealed that group 6 is running a hidden `sshd` process (PID `2834` on TCP port `7865`. Using our LKM we determined that the pointers to the system calls `read`, `getdents`, and `recvmsg` are manipulated. Furthermore, the first few instructions of the `tpacket_rcv` functions are manipulated, too. Port knocking didn't seem to be enabled, there were no netfilter hooks and we could simply connect to the hidden `sshd`.

3.7 Group 7

We were not able to run our user-mode script on this virtual machine as pipes are broken. Using our LKM we determined that the pointers to the system calls `read`, `getdents`, and `open` are manipulated. Furthermore, the first few instructions of the `packet_rcv` functions are manipulated, too. They have a hidden `nc` process (PID `2834`) providing them a remote shell. Port knocking is enabled using a Netfilter hook. We found the name of the hidden module (`marvin`) inside the list of `kobjects`.

3.8 Comparison of the different rootkits

The following table gives an overview over which groups' rootkit we were able to detect in which way using our kernel module. It is intended to show which ways of detecting hidden rootkits are most robust. It is not listing which rootkits were found using the user-space shell script as we were able to detect every rootkit (which would actually run the script) using this.

	SysCall Pointer	SysCall Instructions	PID hidden from */proc*	NF hook	Hidden modules
Group 1					
Group 2	✓	✓	✓	✗	✓
Group 3	✗	✓	✓	✓	✗
Group 4	✓	✓	✓	✓	✓
Group 5	✗	✓	✓	✓	✗
Group 6	✓	✓	✓	✗	✗
Group 7	✓	✓	✓	✓	✓

4 Conclusion

As seen during the analysis section of this paper, we were easily able to detect all the rootkits. Using our tools this was only possible, however, because of the very special and ideal environment for this.

Our LKM relies on a valid *System.map* file which has not been tempered, for example. If this would not be present, we either could not have used some features of our module or would have to find another way to obtain the correct kernel symbols.

Concluding it can be said that we were pretty disappointed with already existing rootkit detection software for linux. After talking with group 7 about this we believe it would be a worthwhile project to work together and release a better detection suite.