

CS5231 Project: Rootkit Analysis in Bitblaze Environment

Anshuman Wadhera	Randy Valentius Kamajaya
A0106504U	U096820Y
Ashish Raste	Wu Jun
A0095975H	A0106507M

April 23, 2013

Supervised by Dr. Liang Zhenkai

Contents

1	Introduction	4
2	Background	4
3	Objective	5
4	Experiments and Results	5
4.1	To analyze the object dump of Adore-ng	5
4.2	Using strace utility to check the system calls	6
4.3	Understanding how insmod works	6
5	Approach and Evaluation	8
5.1	__this_module variable and module struct	8
5.2	Exporting adore-ng's symbol table	9
5.3	Using Tracecap plugin to find adore-ng's influence on kernel .	10
6	Conclusions and Future Work	12
6.1	Findings	12
6.2	Issues faced	12
6.3	Conclusion	13
6.4	Future Work	13

ACKNOWLEDGEMENT

We, the Team 07 of CS5231 class of 2013, are indebted to the constant support provided by our professor Dr. Liang Zhenkai who not only encouraged us to stream-line our thoughts at the kernel level but also showed us the right path for tackling the issues/doubts that we faced intermittently. His classes on System Security concepts kept up our enthusiasm throughout the semester without which we may not have achieved our goals.

Not to forget, CS5231's Teaching Assistants especially Mr. Dai Ting and Mr. Bodhisatta Barman Roy were always available to extend their support by giving us some crucial suggestions/tips which made us to think and explore the problem in our hands properly. Finally, kudos to our team RootBlazers.

1 Introduction

As we know, rootkit attacks have become increasingly popular and difficult to detect. Also, recent rootkits are more into modifying the kernel data structures dynamically like the FU rootkit for example. This motivates us to experiment how BitBlaze would detect and analyze the behavior of rootkits using its binary analysis platform when compared to the detection/analysis tools, which perform:

1. Anomaly detection that relies on heuristics
2. Misuse detection that relies on code signatures, and
3. Integrity checkers

Some examples of rootkits:

1. Early generation rootkits were able to modify main system files on the disk which were then counter-attacked by file system integrity checkers like Tripwire.
2. Next came the rootkits that installed hooks in the memory space of processes and modified their execution path, which were detectable by anomaly detection techniques that used heuristics and statistical deviations from their normal execution paths.

2 Background

A rootkit is a type of malicious software that is activated each time your system boots up. It is a program, script or set of software tools that allows an attacker full access to your PC or network. By full access, we mean administrator-level access. Rootkits are difficult to detect because they are activated before your system's Operating System has completely booted up. It's the fact that rootkits are so difficult to detect that makes them dangerous. A rootkit often allows the installation of hidden files, processes, hidden user accounts, and more in the systems OS. Rootkits are able to intercept data from terminals, network connections, and the keyboard.

A rootkit is not a single piece of software or code designed with a single task, rather it is a system of several programs designed to take control of a machine at the administrator level, and maintain that control undetected by the system's users or legitimate administrators. Rootkits are by far the most dangerous threats to computer security, even when they are authored by legitimate companies for legitimate reasons.

The main form of an attack for a rootkit is stealth. They will hide away, deep in the recesses of our computers. Because they have administrator-level access they can do things like hijack your Windows searches and hide any information about the rootkit itself, control your Anti-Virus software and tell it to ignore the rootkit, and hide from the list of active processes.

What makes the rootkit so dangerous is the ability to hide itself and its subordinate processes from view. A well written rootkit will change the way the operating systems sees and reports running programs and processes, effectively making it blind to the unwanted software. As the rootkit takes more and more control, it can make changes to the system's kernel, daemons, drivers, and configuration files, all to mask its presence. A successful root kit will re-write the login script allowing it to accept the attacker's login regardless of any changes made to the system by users or administrators. This hidden login gives the attacker unlimited control of the machine at will[1].

The most famous rootkit was one that was installed by some Sony audio CDs. Sony hid a rootkit on people's computer as part of its Digital Rights Managment strategy. This gave them effective control of a user's PC. A security expert called Mark Russinovich (of Sysinternals) discovered the Sony rootkit, and it made the news the world over. Sony had to issue a download so that people get the rootkit off their computers. They also recalled all the music CDs that had the rootkit software [2].

3 Objective

The main goal of this project would be to understand the rootkit threat from a kernel's perspective as well as the low level functionalities in the Linux kernel. As rootkit detection is difficult, BitBlaze must be put in use to detect a modified behaviour of certain processes like **ps**. By using Dynamic and Static analysis features of BitBlaze, we can narrow down the address region in the kernel where a rootkit hooks its functions. Ultimately, we want to be able to detect the changes made by the rootkit to the system, which can serve as an indicator to tell whether a user's system has been infected by rootkits or not.

4 Experiments and Results

In this project, we take adore-ng rootkit for our experiments and hence we analyse its behaviour. To find the kernel behaviour of adore-ng, we did a number of experiments which were aimed at finding the kernel instructions that it executed in order to hide itself and other processes. In the following experiments and throughout our work, we used Redhat 7.3 image with Linux 2.4 kernel which also had the adore-ng's source to experiment upon. Here after we will refer to this work environment as Redhat machine. Now let us get into the details of the experiments.

4.1 To analyze the object dump of Adore-ng

Aim: To find whether

1. Adore-ng gets into the kernel using any well defined kernel APIs (or)
2. Does it use any tricky hacks like hard-coded assembly to get into the kernel memory space

Method: Run the Redhat machine which has the adore-ng source, `insmod adore-ng` after doing a `configure`, and compile adore-ng using `make`. The steps for

collecting the object dump of adore-ng are listed below:

```
cd adore-ng
./configure
make
objdump -S adore-ng.o > /root/adore-ng_source_dump (1)
objdump -d adore-ng.o > /root/adore-ng_executable_dump (2)
```

In the above sequence of commands, (1) dumps the source code along with the assembly instructions and (2) dumps the assembler contents of the executable sections. Now we have two files which has the object dump of adore-ng's compiled source.

Result: Analysing the object dump files didn't give us any lead to find the behaviour of adore-ng because it shows the assembly instructions which were obtained statically from the object file of adore. May be we missed important aspect here. Hence we needed to find some method which could record the instructions that adore executed dynamically.

4.2 Using strace utility to check the system calls

Aim: To find whether the address information of the system calls made by adore-ng can be exposed using the **strace** utility.

Method: After installing adore-ng through insmod in the Redhat machine, we did a **strace** on the *modified* **ava** utility (please see **ava.c** in the codes directory) which can hide a process using the **switch -i**. We then obtained the system calls as given in the snapshot shown in Figure 1 below. We can see that the **ava** utility simply unlinks the pid given to it from the **/proc** directory after gaining necessary privileges.

Result: We understood that **ava** utility acts as a hacker's control program in the user space. So when a hacker needs to hide a process, **ava** is invoked to tell adore-ng which process is needed to be hidden by adore-ng's LKM which would have been loaded in the kernel already. In fact **ava** triggers the invocation of the modified **lookup()** function to mark the process needed to be hidden.

4.3 Understanding how insmod works

Aim: Trying to hook the functions in the kernel or insmod, to monitor every kernel module's loading behavior.

Method: We know that **insmod** calls **init_module()** system call which loads adore-ng LKM into the kernel memory space. This **init_module()** system call

```

root@rh73vm0:~# strace -p 26701
read(0, "3\n", 1024) = 2
write(1, "Checking for adore 0.12 or higher ...", 39) = 39
open("/proc/123456", O_RDONLY|O_CREAT, 0) = -1 ENOENT (No such file or directory)
close(-1) = -1 EBADF (Bad file descriptor)
unlink("/proc/123456") = -1 ENOENT (No such file or directory)
getresuid32(0xbffff428, 0xbffff42c, 0xbffff430) = 0
getuid32() = 0
open("/proc/fullprivs", O_RDONLY|O_CREAT, 0) = -1 ENOENT (No such file or director
y)
close(-1) = -1 EBADF (Bad file descriptor)
unlink("/proc/fullprivs") = -1 ENOENT (No such file or directory)
getuid32() = 0
write(1, "Adore 1.54 installed. Good luck.", 33) = 33
write(1, "Did nothing or failed.\n", 23) = 23
open("/proc/hidden-8", O_RDONLY|O_CREAT, 0) = -1 ENOENT (No such file or directory)
close(-1) = -1 EBADF (Bad file descriptor)
unlink("/proc/hidden-8") = -1 ENOENT (No such file or directory)
write(1, "process hidden successfully\n", 27) = 27
munmap(0x40014000, 4086) = 0
_exit(0) = 7
[root@rh73vm0 root]#

root@rh73vm0:~/adore-ng# ./ava
enter the pid to hide: 8
Checking for adore 0.12 or higher ...
Adore 1.54 installed. Good luck.
Did nothing or failed.
process hidden successfully
[root@rh73vm0 adore-ng]#

```

Figure 1: strace output on ava utility

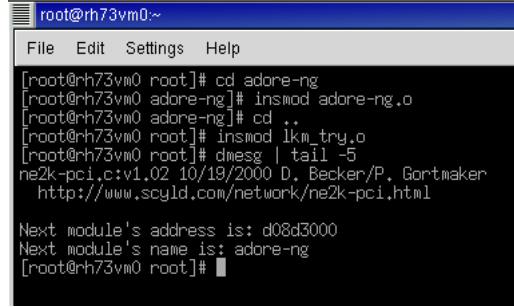
invokes adore-ng's initialization routine after it loads adore-ng. This is taken care by insmod utility which passes the address of the `init_module` subroutine of adore-ng to the `init_module()` system call. One interesting aspect is that this address is passed via the `init_module` subroutine present in the Standard C library [3].

From the brief knowledge obtained as mentioned above, we got to know that `__this_module` object of the struct `module` in the kernel has some information fields related to an LKM such as the kernel symbols it has exported, number of such symbols, its starting address in the kernel space etc [4]. From this information, we did a small experiment to find out the address of adore-ng in the kernel space.

1. We insmod'ed adore-ng first
2. Then we created our own LKM having the statement to print out the address of the next module (adore-ng in our case) using `__this_module.next` pointer. Here we understood the concept that the LKMs are maintained by the kernel in a list structure and are being loaded at the beginning of the list. Hence our LKM's next pointer points to adore-ng.

Verification of this process:

1. Before insmod'ing adore-ng, we installed a couple of modules and checked that the addresses being returned were not constant.
2. Next we insmod'ed adore-ng, used the struct `module` of the kernel in our LKM `lkm_try` (please refer to the source `lkm_try.c` in the codes directory for more details) to print the next module's address and its name using `__this_module.next` and `__this_module.next->name` respectively. This is shown in the Figure 2 below.



```
root@rh73vm0:~  
File Edit Settings Help  
[root@rh73vm0 root]# cd adore-ng  
[root@rh73vm0 adore-ng]# insmod adore-ng.o  
[root@rh73vm0 adore-ng]# cd ..  
[root@rh73vm0 root]# insmod lkmod_try.o  
[root@rh73vm0 root]# dmesg | tail -5  
ne2k-pci.c:v1.02 10/19/2000 D. Becker/P. Gortmaker  
http://www.scyld.com/network/ne2k-pci.html  
  
Next module's address is: d08d3000  
Next module's name is: adore-ng  
[root@rh73vm0 root]#
```

Figure 2: Extracting adore-ng’s address through our LKM lkmod_try

Result: One interesting aspect that we observed is that the size of adore-ng calculated by this process is less than the *size* field shown by lsmod and hence we feel that the block of memory assigned to adore-ng is not continuous. This is due to the fact that the virtual address space given to an LKM is contiguous but in the physical memory LKMs are not allocated in a continuous space.

5 Approach and Evaluation

To detect the influence on the kernel caused by adore-ng, we need to analyse adore-ng’s kernel behaviour, get its information related to the kernel and verify the approach that we employ. Following subsections talk mainly about the mechanisms and drawbacks of the methods that we used to get information about adore-ng, and why we can use the Bitblaze [5] to reveal adore-ng’s influence on the kernel. More specifically, we focus on the process hiding feature of adore-ng and use Temu’s Tracecap plugin[6] to analyse it.

5.1 `__this_module` variable and module struct

The mechanism of adore-ng hiding itself from the `ps` listing is mentioned in [7] *i.e* it wipes the information written to the *syslog* that is related to it, hiding the existence of its LKM. In our approach we focus on the LKM. We know from our previous experiments that the kernel uses a double linked circular list to manage all the LKMs, and when a LKM is loaded it is inserted in the list head with its pointer member *next* pointing to the address previously pointed to by the list head [8].

Additionally, in the kernel space, `__this_module` variable refers to the list head. Adore-ng uses it to remove its LKM from the list by inserting a new *clean.o* module that modifies the address to which *next* pointer points to. Similarly, we can insert our own module which takes place of the *clean.o* to get the adore-ng’s address. The key fields of the `struct module` in the Redhat machine is given below

We can verify our approach by using the struct member name, and we get other information from `struct module` as we did in section 4.3 under Experiments. As

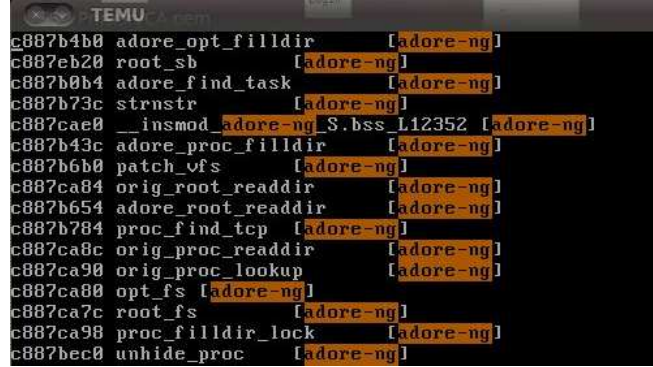
```
/*linux/module.h*/
struct module_symbol{
    unsigned long value;
    const char *name;
}

struct module{
    ...
    struct module *next;
    const char *name;
    ...
    unsigned nsyms;
    ...
    struct module_symbol *syms;
    ...
}
```

this method requires *adore-ng*'s kernel module to be in the list, by using the *lsmod* command we can reveal the existence of *adore-ng*. The drawback here is apparent that this method is useless under a real environment where an LKM rootkit is already installed and has hidden itself from the *lsmod* listing. Hence we must not rely on *lsmod* for listing out *adore-ng*.

5.2 Exporting *adore-ng*'s symbol table

Another way to get the information of *adore-ng*'s functions addresses is by exporting the kernel module's symbol table. As mentioned in [8], we can export an LKM's symbols using `EXPORT_SYMBOL` function so that they can be used by other modules. From this perspective, we tried to export *adore-ng*'s symbols by commenting out `EXPORT_NO_SYMBOL` routine in its source. Thus it would be easy to know the functions' addresses in *adore-ng*'s kernel module when we do a `cat /proc/ksyms | grep "adore-ng"`. These addresses can then be used for binary analysis in Bitblaze environment. We found that *adore-ng*'s symbol information can also be obtained through `__this_module` object and so now we can mutually verify both the methods. Apparently, we can not access the source code of a rootkit in a real environment. As this approach requires us to modify the source code of *adore-ng*'s kernel module, we can't choose to comment out `EXPORT_NO_SYMBOL` statement of *adore-ng*. And hence we decided to stick `struct module` to get the information regarding *adore-ng*'s symbols. Figure 3 below shows some symbols which were exported and which were used by *adore-ng* after its loaded.



```
TEMU
c887b4b0 adore_opt_filldir [adore-ng]
c887eb20 root_sb [adore-ng]
c887b0b4 adore_find_task [adore-ng]
c887b73c strnstr [adore-ng]
c887cae0 __insmod_adore-ng_S.bss_L12352 [adore-ng]
c887b43c adore_proc_filldir [adore-ng]
c887b6b0 patch_vfs [adore-ng]
c887ca84 orig_root_readdir [adore-ng]
c887b654 adore_root_readdir [adore-ng]
c887b784 proc_find_tcp [adore-ng]
c887ca8c orig_proc_readdir [adore-ng]
c887ca90 orig_proc_lookup [adore-ng]
c887ca80 opt_fs [adore-ng]
c887ca7c root_fs [adore-ng]
c887ca98 proc_filldir_lock [adore-ng]
c887bec0 unhide_proc [adore-ng]
```

Figure 3: Symbols exported and used by adore-ng, extracted by listing out /proc/ksyms

5.3 Using Tracecap plugin to find adore-ng’s influence on kernel

We know that GNU/Linux implements a VFS(Virtual Filesystem Switch) to hide the implementations of different filesystems supported by it([8] and [9]). We also know that the Linux kernel sees almost every object(process, directories, etc.) in the system as a file. Thus all operations on a process are operations on a file. The operations on a specific filesystem is different from others *i.e* the operations on a file is determined at the time of accessing the file.

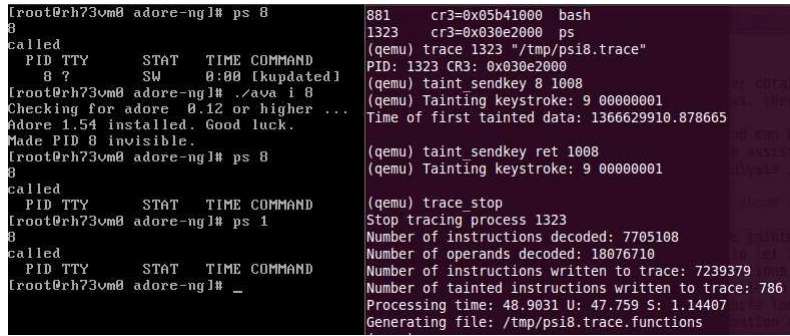
By analysing adore-ng’s source code and referring to its process-hiding behaviour mentioned in [7], we find that the adore-ng’s process hiding logic is mainly about marking the process id needed to be hidden and modifying related filesystem’s function pointers. When an user enters the *ps* command, adore-ng will check whether a process’s information should be returned or not. If the process is hidden by ava(adore-ng’s control program in the user space), it wouldn’t return the hidden process information, otherwise it returns the information. To return a visible process’s (unharmed process) information, adore-ng calls the original kernel function. It gives us a hint that the kernel’s behaviour is influenced by adore-ng. To verify this, we bring Bitblaze into action.

Under the Bitblaze environment, we can still find the hidden process in Temu which can not be seen when we are in the infected system. We use the Tracecap plugin of Temu to taint the memory region where adore-ng acts. To make sure that adore-ng’s specific function’s (hooks that are placed in the kernel by adore) are tainted, we modified the *ps* command so that the arguments passed to it can be tainted. After obtaining the tainted trace file, if we can extract the addresses of the functions used by adore-ng to hide a process, then we can say that the kernel is affected by it.

From a defender’s point of view, this method can be employed under a real environment since modifying an userspace program deliberately can be feasible. And the assistance of previous methods is not necessary which makes the analysis easier. The drawback is that the binary analysis is onerous and finding the tainted

points/instructions is not so easy if we don't know the source code structure of a rootkit.

Specifically, to make sure that the **ps** utility can be tainted, we downloaded the `/proc` file system utilities from [11]. We make some minor modifications to it so that it waits for a pid input from the terminal, thus we can use the Tracecap plugin to taint the pid to get the instructions executed in the kernel space. As for the process hiding part, `adore-ng` replaces the `/proc` filesystem's `readdir()` and `filldir()` function pointers with its corresponding functions (`adore_proc_readdir()` and `adore_proc_filldir()`) pointers respectively. Refer to Figure 4 for the tainting process carried out on the modified **ps** utility.



```

[root@rh73vm8 adore-ng]# ps 8
8
called
  PID TTY          STAT TIME  COMMAND
  8 ?           SW      0:00 [kupdated]
[root@rh73vm8 adore-ng]# ./ava i 8
Checking for adore 0.12 or higher ...
adore 1.54 installed. Good luck.
Made PID 8 invisible.
[root@rh73vm8 adore-ng]# ps 8
8
called
  PID TTY          STAT TIME  COMMAND
  8 ?           SW      0:00 [kupdated]
[root@rh73vm8 adore-ng]# ps 1
1
called
  PID TTY          STAT TIME  COMMAND
[root@rh73vm8 adore-ng]# _

881  cr3=0x05b41000  bash
1323  cr3=0x030e2000  ps
(qemu) trace 1323 "/tmp/ps18.trace"
PID: 1323 CR3: 0x030e2000
(qemu) taint sendkey 8 1008
(qemu) Tainting keystroke: 9 00000001
Time of first tainted data: 1366629910.878665
(qemu) taint sendkey ret 1008
(qemu) Tainting keystroke: 9 00000001
(qemu) trace stop
Stop tracing process 1323
Number of instructions decoded: 7705108
Number of operands decoded: 18076710
Number of instructions written to trace: 7239379
Number of tainted instructions written to trace: 786
Processing time: 48.9031 U; 47.759 S; 1.14407
Generating file: /tmp/ps18.trace.functions

```

Figure 4: Tainting the **ps** process' pid input using Tracecap plugin

In Figure 4 above, the left terminal is Redhat machine's while the right one is the Temu running on the host machine. Following steps describe our tainting process.

1. Using the modified **ps** utility, we get a process'(having pid 8) information (Note that we can taint any process having pid ≤ 9).
2. Then we hide the process having pid 8 with the help of `ava` utility.
3. Entering the **ps** command once again to ensure that the process with pid 8 has been hidden.
4. Now we are ready to taint the **ps** utility. Hence we wait for the input pid (numbered 8) to be sent from the Temu terminal running on the host machine.
5. We use the Tracecap plugin to send the taint key 8 to **ps** that triggers the trace.
6. Using Vine's `trace_reader` utility, we convert the *binary* trace collected from the above step to human readable assembly type instructions. Please refer to the files `psv_trace_address` and `psi_trace_address` which have the instructions containing `adore_proc_filldir` symbol address (which we obtained as `0xc887b43c`). `psi_trace_address` is extracted from the trace file having the tainted instructions of the process (with pid 8) hidden, and `psv_trace_address` contains the same instructions as `psi_trace_address` but is obtained when the process (with pid 8) is *not* hidden.

Notes:

1. Please note that we have also included the original trace files named `psi_trace` and `psv_trace` which were obtained from the `trace_reader` output. Going through the instructions of these files will be cumbersome as their combined size crosses 1GB and hence we recommend to extract only the instructions having `adore_proc_filldir` address or the taint mark **T1** for our verification purpose.
2. There is a "called" word on the `ps` output to denote that the modified `ps` utility is running.

6 Conclusions and Future Work

In this section, we discuss about some interesting findings, issues that we faced, conclusions and the areas to be explored upon in future.

6.1 Findings

The following findings on `adore-ng` were interesting to see which tempts us to do more research in the working of rootkits.

1. We can't hide the `init` process through `adore-ng`. This was verified by running the `ava` utility with `-i` switch and observing the output of our modified `ps` program on pid 1 which showed the details of the `init` process.
2. Tainting the pid input to the modified `ps` program generates a trace file through the Temu plugin whose size differs in each run. Please note that this wasn't the case when we had the `filter_kernel_all` flag off *i.e* the flag which tells to record all kernel instructions for a process. So when we had this flag on, we observed changes in the trace file size. We learnt that this is because the kernel may or may not execute some instructions like swapping pages.
3. We couldn't taint the processes having pid greater than 9 *i.e* the ones having two decimals. One reason for this may be that the `taint_sendkey` just sends the first digit of the input that is being tainted to the running application.

6.2 Issues faced

1. In the trace files that we obtained, we found the address of `adore_proc_filldir` symbol which `adore-ng` uses to hide the processes. But when we observe all such instructions involving this symbol's address, we see that there is no **T1** flag in them which would have told us that the particular instruction in consideration is tainted.
2. We could only find an approach to detect the process hiding behaviour of `adore-ng` since we could modify the `ps` utility which can take a pid (a number) as an input. But we couldn't think of an approach to tackle the file-hiding behaviour of `adore-ng`. Thinking along the same lines, one can immediately ask why not modify `ls` process and taint its input. The caveat here is that it

doesn't accept a number as an input to be tainted but a string. So there must be some way that shall be explored to taint/detect the file-hiding behaviour of `adore-ng`.

6.3 Conclusion

We conclude that rootkits such as `adore-ng` can place its hooks in the VFS of Linux kernel when they are loaded, so that the high-level utilities run through those hooks resulting in a malicious behaviour. With the help of Bitblaze, now we have an approach which combines Dynamic and Static analysis of a binary to detect a rootkit's hooks, where we recommend to observe whether any unknown kernel symbol is referred in the trace file obtained after tainting any high-level utility's input.

6.4 Future Work

We have the following areas to be covered/worked upon in future.

1. Use the Bitblaze environment to observe the file-hiding behaviour of `adore-ng` and hence find a method to taint a memory region which is influenced by the file-hiding hooks of `adore-ng`.
2. Analyse other stealthy rootkits using Bitblaze to get to know their malicious behaviour in the kernel.

References

- [1] Rootkit: Definition, Prevention and Removal
<http://www.zsecurity.com/articles-rootkits.php>
- [2] Rootkit: Sony BMG Rootkit Scandal, Wikipedia
<http://en.wikipedia.org/wiki/Rootkit>
- [3] Linux Loadable Kernel Module HOWTO:
<http://www.tldp.org/HOWTO/Module-HOWTO/x627.html>
- [4] Infecting Loadable Kernel Modules: <http://www.phrack.org/issues.html?issue=68&id=11>
- [5] Bitblaze <http://bitblaze.cs.berkeley.edu/>
- [6] Temu: <http://bitblaze.cs.berkeley.edu/release/temu-1.0/howto.html>
- [7] Walter Zhou 2008. *adore-ng-0.56_implementation_analyse*
<http://bbs.chinaunix.net/thread-1955252-1-1.html>
- [8] Daniel P. Bovet, Marco Cesati 2005 *Understanding the Linux Kernel, 3rd Edition*, O'Reilly, ISBN: 0-596-00565-2
- [9] Robert Love, 2005 *Linux Kernel Development, 3rd Edition*, Addison-Wesley, ISBN 978-0-672-32946-3
- [10] Thorsten Holtz, *Open Chaos Malware Unix*, Laboratory for Distributed Dependable Systems/ Chaos Computer Club Cologne, RWTH Aachen, 2004.
<https://wiki.koeln.ccc.de/images/4/49/Openchaos-malware-unix.pdf>
- [11] Ps program source code: <http://procps.sourceforge.net/>
- [12] Christopher Kruegel, William Robertson and Giovanni Vigna, "Detecting Kernel-Level Rootkits Through Binary Analysis", *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*, 2004.
- [13] Heng Yin, Zhenkai Liang and Dawn Song, "HookFinder: Identifying and Understanding Malware Hooking Behaviors", *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, February 2008.
- [14] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena, "BitBlaze: A New Approach to Computer Security via Binary Analysis", *Proceedings of the 4th International Conference on Information Systems Security (ICISS)*, December 2008.