UNIX and Linux based Rootkits Techniques and Countermeasures

Andreas Bunten
DFN-CERT Services GmbH
Heidenkampsweg 41
D-20097 Hamburg
bunten@dfn-cert.de

Abstract

A rootkit enables an attacker to stay unnoticed on a compromised system and to use it for his purposes. This paper reviews techniques currently used by attackers on UNIX and Linux systems with a focus on kernel rootkits. Example rootkits are classified according to code injection and how the flow of execution is diverted within the kernel. The efficiency of different countermeasures is discussed for these examples.

Keywords: Malware analysis, kernel rootkits, incident response.

1 Motivation

An attack on a system connected to the internet typically shows the following pattern: The target is scanned for potentially insecure services. The attacked system is called *compromised* as soon as a vulnerability has been exploited and the attackers are able to execute commands. Local vulnerabilities are subsequently utilised until the attackers have the same privileges as the system administrator. The administrator will try to regain control as soon he realises the compromise. This can be done by eliminating the vulnerability, a full reinstall of the system, or simply by taking it offline. Therefore, if the attackers want to maintain their access to the system they have to take measures not to be noticed.

The typical attacker uses collections of programs to hide his presence and activities on compromised systems - so called *rootkits*. This article discusses the techniques employed by the attacker to stay invisible on the compromised system using rootkits.

1.1 The history of rootkits

Rootkits first appeared at the end of the 80's [Ditt2002] and consisted of tool collections used to manipulate UNIX logfiles in order to hide the presence of certain users [BTA1989]. Later on, attackers started to substitute programs like `ls`, `ps` or `netstat` to hide their activities. Such programs were soon packaged together with manipulated versions of `login` and similar programs to secretly log passwords. *Sniffers* were used to read unencrypted passwords from the local network.

In the mid 90's *kernel rootkits* appeared on Linux systems [Half1997]. These were loaded as modules at runtime to manipulate the core of the operating system itself. This technique allowed very thorough obfuscation of the attacker's activities e.g. by redirecting

system calls instead of exchanging single programs. By the end of the 90's kernel rootkits existed for practically all modern UNIX-like operating systems. At the same time rootkits for Microsoft Windows appeared [Hogl1999]. Although attackers had to cope with a lack of documentation they were able to replay the development of rootkits in the UNIX-world within a few years. Current Windows rootkits are kernel-based and provide similar functionality as their UNIX counterparts.

In recent years the methods used by attackers have been refined. New ways of code injection have been developed, allowing to patch a rootkit into the running kernel without the usage of modules. Other rootkits manipulate existing modules or kernel images on disk to install themselves. While manipulation of the kernel by the redirection of system calls is relatively easy to detect, modern rootkits have become more sophisticated and more difficult to detect. The flow of execution is diverted at different places and at deeper levels within the kernel. Furthermore, obfuscation techniques are employed in the rootkit binaries to prevent analysis if the rootkit is found. In response, the countermeasures have also evolved. Aside from simple checksum based approaches new methods of detection, like runtime measurement of system calls, have been developed. Critical resources of the operating system are checked for consistency to discover the diversion of the execution flow by the rootkit.

1.2 Technical terms

The concepts of user and kernel mode as well as system calls are briefly introduced in a general manner for UNIX-like operating systems in order to be able to characterise rootkits. More information on the subject concerning UNIX [Bach1986] and Linux [BC2002] can be found in the literature.

1.2.1 User and kernel mode

UNIX-like operating systems distinguish between processes running within the kernel (running in *kernel mode*) and processes started by users (running in *user mode*) including the special user `root`. Memory belonging to the kernel can be accessed via the device files `/dev/kmem` and `/dev/mem`. Depending on the configuration, data can be read and maybe also written. Very few programs use this interface and access is generally restricted to the user `root`. Even though access to kernel memory is possible this way, there are not necessarily symbol tables or similar mechanisms, which will give any help finding particular data structures or functions.

Functions inside the kernel can only be accessed by the use of *system calls*, which provide a well-defined and static interface. Even the user `root` can not run arbitrary code in kernel mode. Though, most modern operating systems are able to load *kernel modules* at runtime. Special system calls are used to bring the code of the module into the kernel. Modules are often used for drivers of multimedia or hotplug hardware.

1.2.2 Execution of a system call

The command '`ls /tmp`' gives a listing of all files located in the directory `/tmp`. The program `ls` uses the system call `open()` to access the directory for reading. The flow of execution can be seen in figure 1. The basic steps are:

- The program `ls` makes a call to `open(/tmp, O_RDONLY)`. To do this, parameters are loaded in the corresponding registers and the interrupt reserved for system calls is triggered.
- The system switches into kernel mode to handle the interrupt. The *interrupt descriptor table* (IDT) is referenced and the corresponding *interrupt handler* is called.
- The interrupt handler references the *syscall table* to finally call the function `sys_open()` with the parameters provided by `ls`.

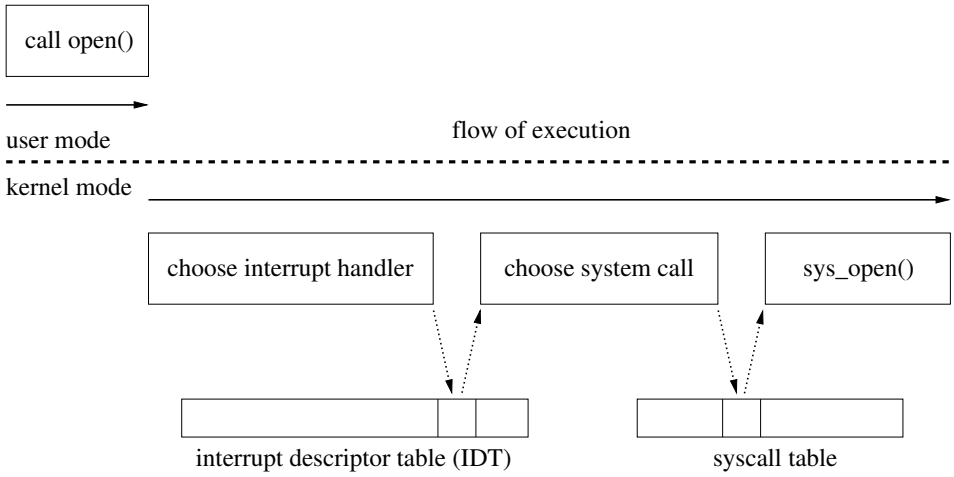

Figure 1: *The system call* `open()` *is used. The flow of execution is transfered to kernel mode, where the IDT and the syscall table are used to determine which function to call.*

A file handle will be returned to the calling program `ls`, which will then use more system calls like `getdents()` to actually read all the entries of the directory and to print them out. It is noteworthy that a few central resources like the IDT and the syscall table are used to determine what function inside the kernel is called.

2 User mode rootkits

The first rootkits worked in user mode and have mostly been abandoned. However, old rootkits are still found while analysing compromised machines. Especially on unusual hardware where ports of current rootkits are not available.

2.1 Techniques used by early rootkits

To give the full picture, examples are briefly presented for logfile filtering and for rootkits, which exchange system binaries.

2.1.1 Manipulation of logfiles

The programs for the manipulation of logfiles used in early rootkits [BTA1989] are still in use - though in more sophisticated versions. Typically logfiles are filtered automatically. The attacker provides a few keywords and the program scans various logfiles and deletes all

lines containing the keywords. Such a keyword might be the IP of the attacking machine. Figure 2 shows the last lines of a logfiles before and after usage of such a tool.

```
linux:/home/rks # tail -3 /var/log/messages
Jan 13 14:16:10 linux kernel: VFS: Disk change detected on device fd(2,0)
Jan 13 14:19:39 linux sshd[1517]: WARNING: /etc/ssh/primes does not exist,\
   using old prime
Jan 13 14:19:43 linux sshd[1517]: Accepted password for seg from\
   192.168.1.17 port 36262 ssh2

linux:/home/rks # sh t0rnsb seg
 * sauber by socked [07.27.97]
 *
 * Cleaning logs.. This may take a bit depending on the size of the logs.
 * Cleaning messages (414 lines)...1 lines removed!
 * Cleaning boot.log (7mlines)...0 lines removed!
[...]
 * Alles sauber mein Meister !

linux:/home/rks # tail -3 /var/log/messages
Jan 13 14:16:10 linux kernel: VFS: Disk change detected on device fd(2,0)
Jan 13 14:19:39 linux sshd[1517]: WARNING: /etc/ssh/primes does not exist,\
   using old prime
Jan 13 14:20:21 linux syslogd 1.3-3: restart.
```

Figure 2: *The last three lines of a logfile are shown before and after the tool* `t0rnsb` *is used to delete all lines containing the keyword 'seg'.*

All lines denoting the login of user `seg` are deleted from the logfile using the program `t0rnsb` from the rootkit `t0rnkit`. The program searches various logfiles for the given keyword. However, the limitations of this approach become obvious at a closer look. On this system every login is accompanied by an error message concerning the file `/etc/ssh/primes`. This line was not deleted since it did not contain the keyword. Furthermore the daemon responsible for the logging has been restarted, which is also a telltale sign of intrusion. A system administrator closely inspecting the logfiles will always have good chances to find traces even though the logfiles were manipulated.

2.1.2 Exchanging system binaries

In order to hide the processes and files of the attacker, early rootkits started to substitute system binaries for manipulated versions. The manipulated version of `ls` does not show all files anymore, `ps` will forget about some processes and `netstat` drops some network connections from the output. In the year 2000 the rootkit `t0rnkit` was widely used [CERT2000]. It can still sometimes be found on compromised systems. The rootkit replaces the programs `login`, `ifconfig`, `ps`, `du`, `ls`, `netstat`, `in.fingerd`, `find` and `top` by manipulated versions. The configuration (and sniffer logs) are stored in `/usr/src/.puta`. When a manipulated version of one of the binaries is called, files containing keywords are read and the output is filtered accordingly.

2.2 Detection

As the need arose, mechanisms were invented to identify manipulated system binaries by the use of checksums or by other means.

2.2.1 Checksums

The manipulated binaries might have the same length and timestamp as the original programs but will differ in their checksums. Programs like `tripwire` or `aide` [Aide2004] automate the task of generating checksums for system binaries and important configuration files. These tools allow to check the current system against these checksums at a later time, in order to uncover a compromise.

```
linux:/home/rks # aide -C
AIDE found differences between database and filesystem!!
Start timestamp: 2003-01-13 21:32:11
Summary:
Total number of files=17520,added files=16,removed files=0,changed files=43
[...]
changed:/bin
changed:/bin/ls
changed:/bin/netstat
changed:/bin/ps
changed:/bin/login
[...]

linux:/home/rks # rpm -Vva
[...]
S.5....T   /bin/ls
S.5.....   /bin/netstat
SM5.....   /bin/ps
.M5....T   /bin/login
[...]
```

Figure 3: `Aide` *is run and the checksums of several programs have changed. The packet system* `rpm` *can provide the same information. The '5' in its output stands for a changed MD5 checksum.*

The drawback of this method is the time-consuming overhead of maintaining such a set of checksums. Accordingly, this approach is unfeasible for many administrators. Naturally, the checksums and the binaries needed for the test should be stored offline to be safe from manipulation. A packet system like `rpm` can provide a limited alternative since it also maintains a set of checksums to manage software updates. Figure 3 shows `aide` and `rpm` finding traces of a user mode rootkit.

2.2.2 Special rootkit detectors

Programs like `chkrootkit` [Chk2004] or `Rootkit Hunter` [Rkh2004] automate the task to look for signatures of known rootkits. The signatures often consist of configuration directories with a fixed path or certain strings appearing in the manipulated version of a binary. If e.g. the directory `/usr/src/.puta` exists, it is very probable that the rootkit `t0rnkit` (version 6.66) has been installed. The signatures will not match if a very recent or a slightly modified version of a rootkit is in use. The problem is known from other signature based methods in intrusion detection systems and virus scanners. To fill this gap, the tools also look for general traces left by rootkits. Figure 4 shows the output of `chkrootkit` detecting a rootkit.

```
linux:/home/tools/chkrootkit-0.38 # ./chkrootkit
[...]
Checking 'login'... INFECTED
Checking 'ls'... not infected
Checking 'lsof'... not infected
Checking 'mail'... not infected
Checking 'mingetty'... not infected
Checking 'netstat'... not infected
Checking 'named'... not found
Checking 'passwd'... not infected
Checking 'pidof'... not infected
Checking 'pop2'... not found
Checking 'pop3'... not found
Checking 'ps'... INFECTED
[...]
Searching for t0rn's default files and dirs... Possible t0rn rootkit installed
[...]
```

Figure 4: `Chkrootkit` *is run and known strings from the rootkit* `t0rnkit` *are identified in the programs* `login` *and* `ps`*.*

2.2.3 Manual search

There are numerous ways of looking for rootkits manually. Two examples are shown in figure 5. The content of a directory can be listed in many ways and differences to the output of `ls` should raise suspicion. Looking at the system calls done by `ls`, an `open()` of the file `/usr/src/.puta/.1file` attracts attention. The file is known to be part of the rootkit `t0rnkit`.

```
linux:/usr/src # ls -al
total 8
drwxr-xr-x   3 root     root         4096 Jan 13 21:34 .
drwxr-xr-x  25 root     root         4096 Jan 13 21:34 ..

linux:/usr/src # echo .*
. .. .puta

linux:/usr/src # tar -cf - . | tar -tvf - | grep "^d"
drwxr-xr-x root/root         0 2003-01-13 21:34:51 ./
drwxr-xr-x root/root         0 2003-01-13 20:19:24 ./.puta/

linux:/usr/src # strace /bin/ls / 2> /tmp/tracefile
linux:/usr/src # cat /tmp/tracefile | grep open
[...]
open("/usr/src/.puta/.1file", O_RDONLY) = 3
```

Figure 5: *Various means are used to display the files in the current directory and it contains more than* `ls` *displays.* `Strace` *is used to show all system calls done by* `ls` *to* `open()` *files. Strangely, the file* `/usr/src/.puta/.1file` *is accessed.*

3 Kernel Rootkits

Today, most rootkits found on compromised systems are kernel-based. The first kernel rootkits were probably written for Linux [Half1997], but the techniques were quickly transfered to other operating systems like Solaris [PlT1999] and modern BSD systems [PrT1999]. Malicious code is transfered into the kernel e.g. as loadable kernel modules using the methods provided by the operating system to load drivers at runtime (cf. 1.2.1). Once in the kernel, system calls are manipulated to hide the attacker. A kernel rootkit by definition manipulates the innermost core of the operating system and can therefore change any information processed by the system. While this, in theory, allows a perfect simulation of the original system, being in fact compromised, it is near to impossible to realise in practice [Skou2003].

A classification of kernel rootkits can be implemented by assessing their methods to inject code into the kernel and by the way the flow of execution is manipulated within the kernel. The following three examples present the most common methods found in kernel rootkits.

3.1 Adore BSD 0.34

The rootkit `Adore` was written for Linux and modern BSD systems like FreeBSD. It was one of the first kernel rootkits, but can still be found on compromised machines. `Adore` allows an attacker to hide files, processes and network connections. Configuration is done by a userspace tool called `ava`. Neither a mechanism to reload the rootkit when the system is restarted, nor a backdoor are part of the rootkit. An attacker has to solve these tasks by himself, e.g. by installing a manipulated SSH daemon and hiding the process and its network connections using `Adore`.

3.1.1 Code injection

The rootkit is loaded into the kernel as a module and the regular interface provided by the operating system is used. A second module is used to delete `Adore` from the data structures used by the kernel to store module information. This makes the rootkit invisible for the regular system administration tools. To prevent the injection of code by this means, module support can be switched off on a given system.

3.1.2 Flow of execution

The rootkit manipulates the syscall table to divert the execution of 15 system calls to its own code. This is the most widely used method of the attackers to change the flow of execution in the kernel. A schematic view can be seen in figure 6.

3.2 SucKIT 1.3b

The rootkit `SucKIT` was published and well documented in 2001 [SD2001]. It runs under Linux and is widely used. The rootkit is very easy to use and already includes a reload-mechanism for system reboots and a backdoor. The backdoor is only activated after a certain packet was send to the system.

The mechanism to reload the rootkit works as follows. A UNIX-like system executes `/sbin/init` at system start. `SucKIT` replaces this file with it's own loader. The loader injects the rootkit into the kernel and executes the original `init`, which was renamed. To

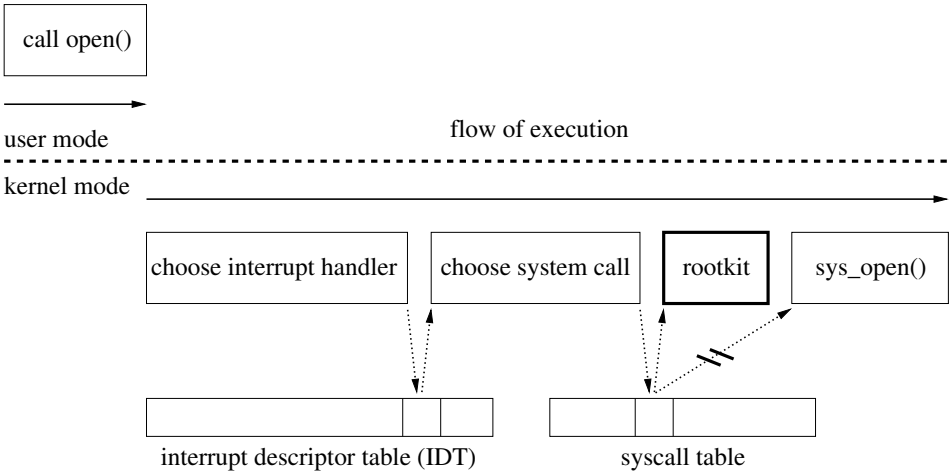

Figure 6: *The syscall table is manipulated and the rootkit is called first when any process uses the system call* `open()`.

hide this manipulation, the renamed original is hidden using the mechanisms of the rootkit. Furthermore, any access to the file `/sbin/init` is redirected to the renamed and hidden original. When the system administrator generates a checksum of the file, it will match the one calculated before the rootkit was installed as seen in figure 7.

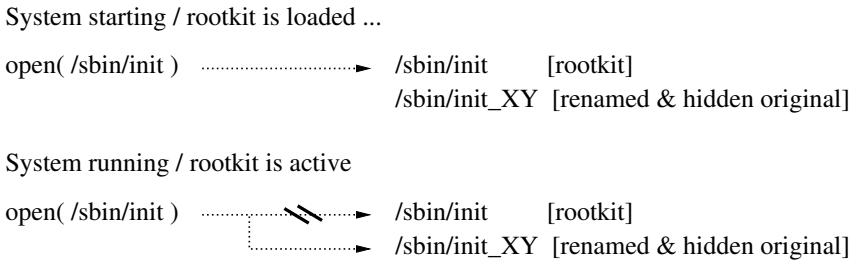

Figure 7: *The rootkit* `SucKIT` *replaces* `/sbin/init` *with its own loader. To hide this fact the rootkit diverts every access to the file* `/sbin/init` *to the hidden and renamed original.*

The ease of use of `SucKIT` includes not having to know the exact version of the operating system. A compiled binary of `SucKIT` can be installed on systems running any Linux version from the 2.2 or 2.4 release trees.

3.2.1 Code injection

The rootkit transfers itself into the kernel by the use of `/dev/kmem`. The method is more complicated than using kernel modules, but can not as easily be blocked. The code of the rootkit is injected in several stages:

- The addresses of the syscall table and of the function `kmalloc()` within the kernel are found by searching kernel memory for certain patterns.
- The function `kmalloc()` is internal to the kernel and needed to reserve space in kernel memory. The address of `kmalloc()` is put into an unused entry of the syscall table.

- `kmalloc()` is executed as a system call and memory in the kernel is allocated.
- The rootkit is written to the freshly reserved space in the kernel.
- The address of the rootkit is put into the unused entry of the syscall table, overwriting the address of `kmalloc()`.
- The rootkit is called as a system call and finally running in kernel mode.

3.2.2 Flow of execution

The rootkit `SucKIT` manipulates the syscall table and diverts the execution of 24 system calls to its own code. In contrast to the method used by `Adore`, the syscall table is first copied and then the copy is modified. This way the original stays untouched and tools designed to check the consistency of critical kernel resources will find no traces looking at the syscall table alone. The interrupt handler used for system calls is then manipulated to use the malicious copy of the syscall table. The manipulation is shown in figure 8.

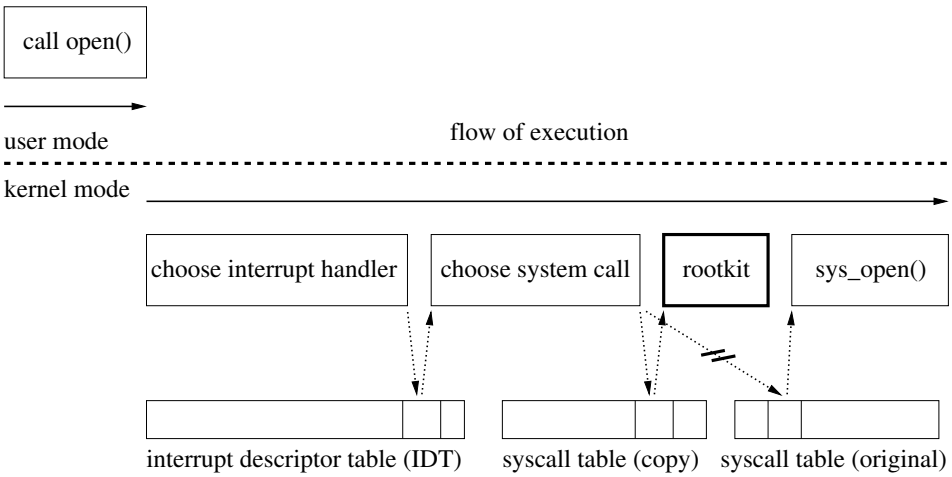

Figure 8: *The rootkit* `SucKIT` *first copies the syscall table and then only modifies the copied version. The interrupt handler for system calls is changed to use the manipulated copy of the syscall table.*

3.3 Adore-NG 1.41

The rootkit `Adore-NG` is a recode of the original `Adore` and will serve as the third example in version 1.41 for Linux. The rootkit uses more sophisticated techniques than `Adore`, but is not yet in widespread use. The rootkit provides neither backdoor nor reload-mechanism by itself. The attacker has to solve these tasks himself as is the case with `Adore`. In addition to the usual features, `Adore-NG` automatically filters log messages generated by hidden processes. This greatly improves the ease of use, since the attacker will not give his presence away as easily by little mistakes. Child processes are hidden automatically if the parent was hidden. The hiding of files and processes has also been improved.

3.3.1 Code injection

The rootkit is loaded into the kernel as a module using the interface provided by the operating system. The injection can be prevented by turning off module support. `Adore-NG` comes with tools for the infection of existing kernel modules [tr2003] and potentially the kernel itself [Jbtz2002] like a virus. Since an infected regular kernel module is recognised as legitimate, the attacker does not need to hide it.

3.3.2 Flow of execution

The virtual filesystem layer (VFS) is a layer of abstraction, residing between the system call interface and the actual implementation of different filesystems. `Adore-NG` redirects the flow of execution in the VFS and does not manipulate central resources like the interrupt descriptor table or the syscall table. Since on a UNIX-like system practically everything is handled as a file, `Adore-NG` can do anything a traditional kernel rootkit can do by controlling the VFS. Detection of the rootkit becomes more difficult, since monitoring a single critical resource is not enough.

4 Countermeasures

Countermeasures should form a closed chain from *prevention* and *detection* to *reaction*. In the following some aspects of prevention and detection will be presented which have to be the basis of any appropriate reaction. The latter is an extensive topic by itself which goes beyond the scope of this article.

4.1 Prevention

Many guidelines on best practise concerning the protection of systems connected to the internet exist. In the following two methods will briefly be described, which specifically try to prevent the injection of code into the kernel.

4.1.1 Deactivation of loadable modules

Many rootkits will simply fail to operate, if the dynamic loading of drivers via the module interface is deactivated. But, since no drivers can be loaded into the kernel at runtime anymore, administrative overhead is created at the same time. Its amount depends on the system in question. On server systems which seldomly get new hardware this might be feasible, but a workstation with multimedia hardware might become unmanageable. Some rootkits can still be loaded via `/dev/kmem`.

4.1.2 Protecting /dev/kmem

Since there are many ways to access the device file `/dev/kmem`, the only effective protection can be provided by *mandatory access controls*. These can be used to limit the access even for the user `root`. On Linux mandatory access controls can be added by special kernel patches and are already included in some Distributions like Trusted Debian (aka Adamantix) and SE Linux. On some systems like Trusted Solaris or some modern BSD systems, mandatory access controls are already included. Very few programs depend on write access to `/dev/kmem`. Depending on the hardware configuration the X-server `XFree86` might be

one of them. Deactivation of module support combined with mandatory access controls for `/dev/kmem` will effectively keep most rootkits outside the kernel.

4.1.3 Anti-rootkit modules

Several kernel modules have been developed with the only purpose of stopping a kernel rootkit from loading. To do this, system calls are manipulated in the same manner as rootkits do. In addition, integrity checkers might be included, which test if the anti-rootkit module is still working or if it has been sabotaged by a kernel rootkit. An examples of this approach is the Linux kernel module `StMichael`. Some rootkits have specifically been adapted to deal with `StMichael`, though.

4.2 Detection

Minor malfunctions often raise suspicion and the administrator might want to check for an installed rootkit. Since nothing definite is known yet, taking a crucial server offline is not an option at this point. Furthermore, there is usually a strict timelimit. These constraints limit the possibilities of the investigator severely. The main methods used to detect kernel rootkits are briefly introduced in the following.

4.2.1 Checksums and special rootkit detectors

Programs to generate and verify checksums have been introduced for user mode rootkits in section 2.2.1, but are still useful with respect to kernel rootkits even though these can redirect access to any file. Many attackers are startlingly inapt and even if the rootkit is used properly, it will never create a perfect simulation of the uncompromised system. Rootkit detectors, as mentioned in section 2.2.2, can be used to search for traces of kernel rootkits. Though, they will rely strongly on signatures for known rootkits.

4.2.2 Critical kernel structures

Saving critical kernel structures and comparing offline copies with current versions will detect many kernel rootkits. A tool for Linux to do this is `KSTAT` [Fusy2002]. Sample output is seen in figure 9.

```
[... before installation of the rootkit ...]

linux:/home/tools/KSTAT24/2.4.16 # ./kstat -s 0
No System Call Address Modified

[... after installation of the rootkit ...]

linux:/home/tools/KSTAT24/2.4.16 # ./kstat -s 0
sys_fork                        0xf880c7a0 WARNING! should be at 0xc01058fc
sys_write                       0xf880ca30 WARNING! should be at 0xc013193c
(...)
sys_ni_syscall                  0xf880c5b0 WARNING! should be at 0xc013ec5c
linux:/home/tools/KSTAT24/2.4.16 # ./kstat -s 1
Restoring system calls addresses...
```

Figure 9: *The syscall table has been recorded beforehand. Then* `KSTAT` *is used to check for changes in the syscall table once before and once after the installation of a rootkit.*

Unfortunately there are many resources, which need to be monitored. A rootkit may stay unnoticed as soon as it manipulates a different resource or no central structures at all (cf. 3.2).

4.2.3 Runtime measurements of system calls

Many rootkits employ complex algorithms to hide the presence of the attacker. Therefore, the number of instructions executed in a given system call might differ widely between the original system and a compromised system on which a rootkit has been installed. Counting the number of instructions used to execute a certain system call is a very general approach to rootkit-detection [Rutk2002]. The output of such a test done with the reference implementation by Jan Rutkowski can be seen in figure 10. At the best, measurements have been done beforehand on the original system. Though, often generic values for different kernel versions are sufficient to identify a rootkit, since the overhead created by some rootkits is large enough.

```
linux:/home/tools/rktest # ./patchfinder -c referenz_2.4.16
* FIFO scheduling policy has been set.
* each test will take 1000 iteration
* testing... done.
* dropping realtime schedulng policy.

    test name      | current | clear | diff  | status
-----------------------------------------------------
open_file          |    7110|    1442|   5668| ALERT!
stat_file          |    7050|    1255|   5795| ALERT!
read_file          |     608|     608|      0|  ok
open_kmem          |    7124|    1510|   5614| ALERT!
readdir_root       |    6497|    2750|   3747| ALERT!
readdir_proc       |   14422|    2401|  12021| ALERT!
read_proc_net_tcp  |   11750|   11750|      0|  ok
lseek_kmem         |     220|     220|      0|  ok
read_kmem          |     327|     327|      0|  ok
```

Figure 10: *The number of executed instructions for certain system calls has been measured beforehand (column 'clear') and is now compared to current values. The values differ widely for some system calls which raises an alert.*

4.2.4 Forensic analysis

A forensic analysis will theoretically by able to detect anything that was written to disk and memory dumps can also be investigated. Such an analysis does not qualify as an ad hoc countermeasure since it is very time-consuming and the system has to be taken offline at least shortly. If checksums exist, the analysis might be done faster and the investigator can much more easily discover the course of events.

4.2.5 Controlling the network

The compromised system will mostly be used to do things, which generate new traffic. Passive monitoring of the network might reveal this unusual traffic. Active scans for open ports can be used to detect some backdoors. Intrusion detection systems (IDS) might be

able to detect the rootkit in transit or activity generated by it. Intrusion detection systems are a huge research field by their own right and will not be discussed here in further detail.

4.2.6 Manual search

There is no perfect rootkit and the attackers themselves tend to make mistakes. A manual search for traces in logfiles and in the filesystem might always generate useful evidence. E.g. the rootkit `SucKIT` uses advanced techniques to hide itself, but is trivial to reveal as soon as `/sbin/init` is checked for the existence of the rootkit loader as seen in figure 11. The file `/sbin/init` is simply renamed using the command `mv` which uses the system call `rename()`. The system call `rename()` is not one of the 24 system calls manipulated by `SucKIT` and is hence not redirected to the hidden and renamed original file. Another helpful method of rootkit discovery is a careful search in the `/proc` directory, which will often expose hidden processes.

```
linux:/sbin # ls -al init*
-rwxr-xr-x    1 root     root        392124 Jan  6  2003 init
linux:/sbin # mv init init.bak
linux:/sbin # ls -al init*
-rwxr-xr-x    1 root     root         28984 Jan  6  2003 init.bak
linux:/sbin # ./init.bak
/dev/null
Detected version: 1.3b
use:
./init.bak <uivfp> [args]
u       - uninstall
i       - make pid invisible
v       - make pid visible
f [0/1] - toggle file hiding
p [0/1] - toggle pid hiding
linux:/sbin #
```

Figure 11: *The investigator is specifically checking for the rootkit* `SucKIT`*: The file* `/sbin/init` *is listed, renamed and listed again. By renaming, the filesize has changed. Executing the binary reveals it to be the user mode frontend of the rootkit.*

5 Applying the countermeasures

The above methods for prevention and detection of kernel rootkits are applied to the example rootkits introduced in section 3. In order to use a single test platform (Linux 2.4.16 on Intel 32 bit) the Linux version of Adore was used instead of Adore BSD, which provides the same features.

For a thorough evaluation of the prevention methods presented in section 4.1, local configuration and policies have to be taken into account. Since this is beyond the scope of this paper, table 2 lists the theoretical impact of mandatory access controls and the deactivation of module support to give a basic overview. An entry of 'ok' implies proper usage of the method, which is by no means trivial in the case of mandatory access controls.

Table 4 lists how successful the different detection tools are with reference to the example rootkits. The test was limited to the methods which can be applied at runtime. Checksums have been created from crucial files and directories of the system beforehand. After

Measure	Adore 0.34	SucKIT 1.3b	Adore-NG 1.41
Deactivate module support	ok	not enough	ok
Mandatory access control lists	ok	ok	ok

Table 2: *The ideal impact of prevention methods on the example rootkits. It is assumed that the methods are applied properly and corresponding usage policies exist and are observed.*

the rootkit was installed, a directory was created and hidden in one of the crucial directories. Rootkits which do not provide a backdoor themselves have been provided with a manipulated SSH daemon, which was then hidden using the mechanisms of the rootkit.

Measure	Adore 0.34	SucKIT 1.3b	Adore-NG 1.41
Checksums tested with `aide 0.7`	ok	ok (`/sbin/init`)	ok
Critical structures with `kstat 2.4`	ok	not detected	not detected
Rootkit detector `chkrootkit 0.43`	ok (hidden process)	ok (`/sbin/init`)	ok
Runtime measurement with `patchfinder`	ok	module did not load	system crashed
Manual search in `/proc`	ok	ok	not detected

Table 4: *Applying the runtime detection methods to the example rootkits.*

Table 4 is briefly discussed by the column. The rootkit `Adore` can be found using any detection method proposed. `Chkrootkit` only finds `SucKIT` when a process is hidden. The rootkit is found by `aide` because the inode number of `/sbin/init` changes needlessly. This might easily be changed by the author of the rootkit. `KSTAT` does not detect `SucKIT` because it only checks the original syscall table even though it is not used anymore (cf.3.2). `Chkrootkit` detects the rootkit. Again, because the rootkit loader in `/sbin/init` is easy to find. The kernel module needed for the runtime measurement could not be loaded while `SucKIT` is active. This might raise suspicion and lead to the detection of the rootkit. A manual search for hidden processes in `/proc` reveals the hidden process. The rootkit `Adore-NG` is also found by checksums. Since no central resources are modified, `KSTAT` does not detect the rootkit. `Chkrootkit` detects the rootkit for the same reason the checksums are successful: files are hidden inside a crucial directory. The system crashes every time the module for runtime measurement is loaded while `Adore-NG` is active. As above, this might raise suspicion. A manual search in `/proc` does not reveal the hidden processes of `Adore-NG`.

The tool `KSTAT` is also able to produce a process list from the internal kernel structures and to compare this list with the output of `ps`. Hidden processes of all rootkits discussed above can be detected that way.

6 Summary and future directions

The development of rootkits has been presented with a focus on kernel rootkits. Example rootkits have been classified according to code injection and how the flow of execution is diverted within the kernel. The efficiency of different countermeasures has been tested discussed with reference to these examples.

The assessment of the prevention method's impact shown in table 2 looks encouraging. However, mandatory access controls are not only the most successful preventive measure, but also create a large administrative overhead and are difficult to configure correctly. Even if applied properly, a vulnerability in the kernel itself can still be used to circumvent them. An administrative error, e.g. not reactivating access controls after patching, might provide ample opportunities for the attacker. This shows that mandatory access controls are an effective, though not ultimate, protection against rootkits. In section 5 the task of rootkit detection was aided by checksums the administrator has created beforehand. In case checksums are not available `Adore-NG` is very hard to find. Generally, if no processes or files are hidden, the investigator has to find the rootkit itself in the kernel. For rootkits which do not rely on obvious central resources like the syscall table or the interrupt descriptor table (e.g. `Adore-NG`) this is very difficult.

With the notable exception of runtime analysis the methods of detection discussed in section 4 are not universal enough to cope with the new mechanisms employed by current rootkits. Any rootkit can be detected in theory, if enough effort is dedicated to the task. Yet, investigators are usually bound by constraints like the time available to spent on the search and the need to keep the system online. Hence, more general detection methods are needed which can easily be applied at runtime. One possibility is a consistency check of the kernel, based on abstract descriptions instead of comparing saved copies of some kernel structures with the current versions. The authors of `Adore-NG` follow such an approach: the execution flow of the kernel is examined at runtime to hide the rootkit in the VFS layer [Teso2003]. On the one hand this exemplifies that current rootkits are more advanced than the tools used to detect them. But on the other hand this also shows directions to be taken towards more generic tools for the investigator. I.e. the analysis of the execution flow could also be used to define consistency checks of the running kernel at an abstract level. While still topic for future research, such tools and methods should combine generic rootkit detection with applicability for a system administrator.

References

[Aide2004] R. Lehti; P. Virolainen, *Homepage Aide*, `http://www.cs.tut.fi/~rammer/aide.html`, 2004.

[BC2002] D. Bovet; M. Cesati, *Understanding the Linux Kernel, 2nd Edition*. O'Reilly. ISBN-0596002130, 2002.

[BTA1989] "Black Tie Affair", *Hiding Out Under Unix*. Phrack Magazin, Issue 25, Vol. 3, File 6. `http://www.phrack.org/phrack/25/P25-06`, 1989.

[Bach1986] M. Bach, *The Design of the UNIX Operating System*. Prentice-Hall. ISBN-0132017997, 1986.

[CERT2000] CERT Coordination Center, *CERT/CC Incident Note IN-2000-10: Widespread Exploitation of rpc.statd and wu-ftpd Vulnerabilities*, http://www.cert.org/incident_notes/IN-2000-10.html, 2000.

[Chk2004] , N. Murilo; K. Steding-Jessen *Homepage Chkrootkit*, http://www.chkrootkit.org, 2004.

[Ditt2002] D. Dittrich, *"Root Kits" and hiding files/directories/processes after a break-in*. http://staff.washington.edu/dittrich/misc/faqs/rootkits.faq, 2002.

[Fusy2002] "Fusys", *Homepage kstat*, http://www.s0ftpj.org/en/tools.html, 2002.

[Half1997] "Halflife", *Abuse of the Linux Kernel for Fun and Profit*. Phrack Magazin, Issue 50, Vol. 7, File 5. http://www.phrack.org/phrack/50/P50-05, 1997.

[Hogl1999] G. Hoglund, *A real NT Rootkit, patching the NT Kenerl*. Phrack Magazin, Issue 55, Vol. 9, File 5. http://www.phrack.org/phrack/55/P55-06, 1999.

[Jbtz2002] "jbtzhm", *Static Kernel Patching*. Phrack Magazin, Issue 60, Vol. 11, File 8. http://www.phrack.org/phrack/60/p60-0x08, 2002.

[PlT1999] "Plasmoid"; "THC", *Solaris Loadable Kernel Modules: "Attacking Solaris with loadable kernel modules"*. http://packetstormsecurity.nl/groups/thc/slkm-1.0.html, 1999.

[PrT1999] "Pragmatic"; "THC", *Attacking FreeBSD with Kernel Modules*, http://www.thehackerschoice.com/papers/bsdkern.html, 1999.

[Rkh2004] , M. Boelen; S. Dudzinski *Homepage Rootkit Hunter*, http://www.rootkit.nl/projects/rootkit_hunter.html, 2004.

[Rutk2002] J. K. Rutkowski, *Execution Path Analysis: finding kernel based rootkits*. Phrack Magazin, Issue 59, Vol. 11, File 10. http://www.phrack.org/phrack/59/p59-0x0b.txt, 2002.

[SD2001] "SD"; "Devik", *Linux on-the-fly kernel patching without LKM*. Phrack Magazin, Issue 58, Vol. 10, File 7. http://www.phrack.org/phrack/58/p58-0x07, 2001.

[Skou2003] E. Skoudis, *Malware. Fighting malicious Code*. Prentice-Hall. ISBN-0131014056, 2003.

[Teso2003] "team teso", *Codeflow Analyse*. Vortrag auf dem 19. Chaos Communications Congress, Berlin. http://www.team-teso.net/articles/19c3-speech/, 2003.

[tr2003] "truff", *Infecting loadable kernel modules*. Phrack Magazin, Issue 61, Vol. 11, File 10. http://www.phrack.org/phrack/61/p61-0x0a_Infecting_Loadable_Kernel_Modules.txt,2003.