

Back-dooring FreeBSD

An Introduction to FreeBSD Rootkit Hacking

Robert Escriva

RPI Security Club
Rensselaer Polytechnic Institute

RPI ACM, September 22, 2008



Outline

1 Introduction

- Overview
- Prerequisites
- First Hands On Look

2 Examples

- Hello, World!
- EBG13
- Process Hiding
- Putting It Together



Overview.

Goals of this lecture.

- Give some useful examples of what rootkits can accomplish.
- Show how easy it is to subvert certain aspects of the operating system.
- Show techniques for detecting the presence of a rootkit.



Overview.

After this lecture you should be able to...

- Describe several ways in which you could subvert an entire operating system given root privileges.
- Show one or more ways in which rootkits can be detected (not definitively).



Academic Prerequisites.

General knowledge that will aid in your understanding of material presented.

- Experience reading/writing C code.
- Knowledge of kernel-level functionality (system calls, etc.).
 - Kernel interface: `read`, `write`, `stat`, etc.
 - Basic file-system functions.
 - Process, threads.



Tools Necessary.

Some things that make following along with the examples easier.

- FreeBSD VMware image I've prepared.



A First Look.

Boot up the VMware image and look around.

- Is your VMware image rooted?
- How do you know it is/isn't?
- Does anything look out of place or act strangely? (bash being the default shell does *not* count)



Take A Second Look.

Your vmware image *is* rooted.

```
■ image_A == image_B
■ freebsd# ls /root/ | grep treasure
■ freebsd# stat /root/treasure
 88 17524 -rw-r-r- 1 root wheel 0 0 "Sep 19
12:10:23 2008" "Sep 19 12:10:23 2008" "Sep 19
12:10:23 2008" "Sep 19 12:10:23 2008" 4096 0 0
/root/treasure
```



A Basic Example.

The treasure file shows you how to get the Hello World example.

- A brief, simple overview that shows how to declare modules, and system calls.
- Look through the code; there are 6 major parts:
 - `hello_args`
 - `hello`
 - `hello_sysent`
 - `load`
 - `offset`
 - Declaring the syscall.



hello_args

The arguments to the system call.

```
struct hello_args {};
```

- All elements of the struct must be of size `register_t`.
- System call arg structs are declared in `sys/sysproto.h`



hello

The system call function.

- Declared as

```
static int hello(struct thread *td, void  
*syscall_args)
```

- Prints a message and exits.

- Prototype is in `sys/sysent.h`



hello_sysent

The `sysent` struct.

- Details number of arguments, the function to call, and the audit event associated with that function.
- Structure is declared in `sys/sysent.h`



load

Kernel module event handler routines.

- Is an event handler for the module.
- Called at module load and unload (all that we use of it).
- Called in other cases as well; see `sys/module.h`



Declaring the syscall.

Making the syscall function callable.

- `static int offset = NO_SYSCALL`
- This indicates that the next open syscall entry should be used.
- The `SYSCALL_MODULE` macro is used to declare the syscall.
- I've also included the expanded form of the macro in the code.



An example of system call hooking.

Forget AES; ROT13 is the way to go.

- Show the code for this example with `freebsd# mkdir /pwned`
- This is a good example of basic system call hooking.
- Specifically hooks `read`.
 - Only do anything on read calls that ask for only 1 byte of data.
 - Only do anything on read calls reading from file descriptor 0.
 - Only change alphabetical text (all else goes through unchanged).
- From my testing, this does not impact ability to log in, nor have any disastrous consequences.
- *Note:* Do not assume `tcsh` reads from file descriptor 0 reliably.



Key Points.

Key pieces of code within the `rot13` module

- `read_hook` is the function that will replace `read`.
- `load` replaces the function pointer to `read` in the `sysent` table with one to `read_hook`.

```
sysent[SYS_read].sy_call = (sy_call_t *)read_hook;
```



Shortcomings.

Ways in which the `rot13` KLD falls short.

- It changes an entry in the `sysent` table.
- It does not work with `tcsh` which is the default.



Fixes.

Ways in which the `rot13` KLD could be improved.

- Hook the function that looks up system calls.
`/usr/src/sys/i386/i386/trap.c`
- Do a more advanced check on keyboard input so that it can work for `tcsh`
- Possibly not ROT13 non-echoed input.



How Can You Hide A Process?

```
perl -e "syscall(37, 1337, 1337)" #37 is kill
```

- How about doing it without altering the scheduling of the process?
- In FreeBSD, processes are not scheduled; threads are scheduled.
- Tools such as `ps` and `top` check the status of processes.
- What if you could delete data structures that tools like `ps` and `top` use without changing the threads associated with the process?



Key Points.

Pieces of the code worth looking at.

- `process_args`
- `process`
- All the rest is similar to code already covered.



Shortcomings.

Ways in which the `process` KLD falls short.

- It modifies internal kernel structures (some code may crash on exit).
- It does not completely hide a process (examine the thread data structures to find the process).
- Sending signals may not work (try it and find out).



Fixes.

Ways in which the `process` KLD could be improved.

- Cleanup all references to the process (e.g. parent process' references to it, etc...).
- Don't let the process be found (it won't crash if it doesn't exit).
- `/usr/src/sys/kern/kern_exit.c`



How do you hide files?

```
perl -e "syscall(188, '/pwned'); " #188 is stat
```

- Remove it from all directory listings.
- Block `stat`, `open`, etc. from finding it.
- Fix filesystem timestamps to hide the change (if necessary).
- This example shows the first (all three are necessary).



Key Points.

Most of this should be familiar.

- `getdirentries_hook` hooks the syscall `getdirentries`.
- Notice it removes `dirent` structs from the buffer if they match the name of the file to hide.



Shortcomings.

There are some glaring issues here.

- Files are hidden from directory listings only; if you know the name of the file you are looking for it is easy to find it.
- When writing to the hidden file, it updates filesystem timestamps. There is nothing to keep these unchanged.
- The file `foo` will be hidden regardless of whether it is `/etc/foo`, `/bin/foo`, or `foo`.



Fixes.

Ways to improve the `hide_file` module

- Hook `open`, `stat`, etc.
- Patch the timestamp routines so that they do not change.
- Any ideas on the last one?



42

What is the answer to life, the universe and everything?

- This KLD has been hiding all the examples.
- Sending a UDP packet on port 42 to the machine will open the example.



Key Points.

Some interesting, new things.

- `getdirentries_hook` **modifications.**
- `open_hook` **redirects opening** `/boot/loader.conf` to `/boot/42ader.conf`.
- `udp_input_hook` **hooks all inbound UDP packets.**



Summary

- KLDs are **not** too intimidating to write if you are patient.
- If the presence of a rootkit is suspected, **no** function provided by the kernel is trustworthy.
- Such techniques should **not** be used maliciously.
- Can anyone think of ways to use KLDs beneficially?



Presentation Materials

All presentation materials will be available online at:

<http://robescriva.com/2008/09/back-dooring-freebsd-acm/>

All presentation materials from the RPI-SEC presentation online at:

<http://robescriva.com/2008/08/back-dooring-freebsd/>



For Further Reading I



J. Kong.

Designing BSD Rootkits: An Introduction to Kernel Hacking.
No Starch Press, 2007.



Kernel Source.

`/usr/src`



man Pages.

`man whatever`

