Western Colorado University

# An Examination of Rootkits

Jared T. Smerdell

System Security CS360

April 24th, 2021

Jared Smerdell

## Introduction

Rootkits are pieces of software that are designed to grant unauthorized access to a computer or program. While not necessarily malware itself, rootkits are often used to conceal malicious packages as some can operate on any ring of privilege. The name rootkit can be broken into two words, “root” and “kit”. Root meaning the highest privilege level a user can achieve, and kit, the package of software used to implement the tool. Rootkits use a technique called cloaking which masks what is really happening. This can make them very difficult to detect and remove. While there are many malicious purposes for rootkits, they also have a few applications beyond this. Some other applications include video game anti-cheat software, enforcement of digital rights management (DRM), and antivirus programs. However, because a rootkit can have root access, it is another attack vector for a hacker.

## Brief History

The earliest known rootkit was written in 1990 by Lane Davis and Steven Drake who targeted the UNIX operating system, more specifically Sun Microsystems’ SunOS. Up until this point, rootkits had only been theorized; after receiving the Turing award in 1983, Ken Thompson, one of the creators of UNIX, gave a lecture and discussed the concept for a potential rootkit exploit. Near the end of Thompson’s paper, he writes, “As the level of program gets lower, these bugs will be harder and harder to detect. A well installed microcode bug will be almost impossible to detect” (Thompson 763), which time has proven to be true. Later, in 1999 the first rootkit appeared on a Windows machine running Windows NT called NTRootkit. Then, in 2009 the first rootkit for Mac OS X appeared. Even programable logic controllers for

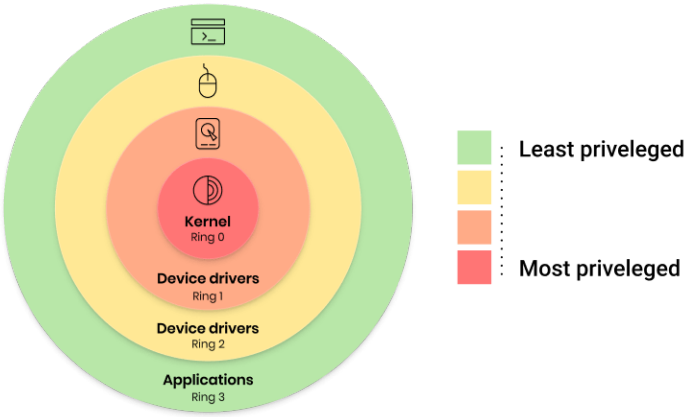
industrial uses have been targeted such as the Stuxnet worm, uncovered in 2010, which used a rootkit component to hide all malicious files and processes.

## Types of Rootkits

While rootkits can be generalized into one category of software, there are many different types of rootkits. The generally accepted breakdown are kernel-mode rootkits, user-mode rootkits and, bootkits. Each type has its own differences but they all try to be a persistent threat with root access.

## Kernel-Mode Rootkits

Kernel-Mode rootkits are one of the more dangerous types, these rootkits have free reign and access to the entire operating system. These rootkits are able to conceal themselves at a very low level and are extremely difficult to identify and remove. These rootkits reside in ring 0, or the highest level of privilege (Flatley 6). With ring 3 being the least amount of privilege. Everything that runs on a computer can be categorized into one of these rings. Each ring can not directly access a ring below it, so an application in ring 3 must rely on a system call to do an operation that requires ring 2 privilege. However, anything that has ring 0 privilege has full access to do any operations.



## User-Mode Rootkits

These rootkits do not reside in ring 0, however, they do often offer an entry point for an attack. Through privilege escalation, an attacker could potentially gain root access through a user-mode rootkit. User-mode rootkits may modify services to make them malicious, such that the attacker could gain root access. Some such examples include changing ‘login’ and ‘sshd’ so that a backdoor password can be included. These rootkits also have the ability to hide processes, files, and commands like ‘killall’ which would terminate the rootkit.

## Bootkits

Bootkits work by infecting the master boot record (MBR) or boot sector. These rootkits load before the operating system is loaded so they are very difficult to track and prevent. These rootkits load before the operating system and are persistent, meaning they tend to stay on a system for extended periods of time without being noticed. During the boot-up phase of a computer, the BIOS (basic input/output system) searches for the first boot device, if this device is the bootkit, it is loaded first then redirects to a section on the disk that will load the rootkit (7). After which, the OS is then loaded. Because the OS is not yet running there are not many countermeasures to verify the integrity of the boot devices which leaves the system extremely vulnerable.

## Executing a Rootkit

Rootkits are not very difficult to get ahold of, online there are many source code files that an individual could use to run themselves. Using a closed environment one could test the functionality of these readily available rootkits.

After running the root-mode rootkit, the process is able to hide and protect itself from an outside source. Using sudo to attempt to remove the rootkit resulted in an error, “Module rootkit does not exist.” Hiding the rootkit ensures that a system administrator or user does not see it, or has the ability to remove it. Once the rootkit becomes unhidden and unprotected it could then be successfully uninstalled. While this rootkit had a way to easily reveal and unprotect itself, this is almost always not the case.

```
smerdy@smerdy-laptop:~/Desktop/rootkit-master$ ./client --hide
smerdy@smerdy-laptop:~/Desktop/rootkit-master$ ./client --protect
smerdy@smerdy-laptop:~/Desktop/rootkit-master$ sudo rmmod rootkit.ko
ERROR: Module rootkit does not exist in /proc/modules
smerdy@smerdy-laptop:~/Desktop/rootkit-master$ ./client --unhide
smerdy@smerdy-laptop:~/Desktop/rootkit-master$ ./client --unprotect
smerdy@smerdy-laptop:~/Desktop/rootkit-master$ sudo rmmod rootkit.ko
smerdy@smerdy-laptop:~/Desktop/rootkit-master$ ./client --root-shell
smerdy@smerdy-laptop:~/Desktop/rootkit-master$ whoami
smerdy
smerdy@smerdy-laptop:~/Desktop/rootkit-master$
```

A core essential for a rootkit is the ability for an attacker to gain root access, the above screenshot demonstrates this. The first command “whoami” shows a normal user with no root privileges. Using the built-in command “--root-shell”, this normal user is able to gain root access. There is no authentication or password verification, they are simply given root access.

```
smerdy@smerdy-laptop:~/Desktop/rootkit-master$ whoami
smerdy
smerdy@smerdy-laptop:~/Desktop/rootkit-master$ ./client --root-shell
root@smerdy-laptop:~/Desktop/rootkit-master# whoami
root
root@smerdy-laptop:~/Desktop/rootkit-master#
```

Without root access, a user can not edit the “/etc/passwd” file which contains the privileges of all the users on the system. In this

```
smerdy:x:0:0:smerdy,,,:/home/smerdy:/bin/bash
notRoot:x:000:1001::/home/notRoot:/bin/sh
"/etc/passwd"
"/etc/passwd" E212: Can't open file for writing
Press ENTER or type command to continue
```

demonstration, using the rootkit, the normal user is able to edit this file and give themselves the root privileges on the entire computer. By changing both the user privilege and the user group, normal users could change their privilege and potentially keep themselves persistent on a system. Obviously, an attacker would spend more time covering their tracks so that it's not apparent that they have the privileges of root.

```
smerdy@smerdy-laptop:~/Desktop/rootkit-master$ grep "smerdy" /etc/passwd
smerdy:x:0:0:smerdy,,,:/home/smerdy:/bin/bash
smerdy@smerdy-laptop:~/Desktop/rootkit-master$
```

Lastly, a core essential for a rootkit is the ability to hide running processes. If, for instance, this process is malware it would be advantageous for the process to be hidden. This demonstration first shows the command "ps" which lists all the currently running processes on the shell. There is bash and the command ps running when the processes are listed. Whenever ps is executed bash should always show up as it is the parent process to everything else that gets listed. In this case, the process ID of bash is 1618. Using the command "--hide-pid=1618" the process is then hidden. Looking at the running processes after bash is hidden shows that it is no longer there. The rootkit successfully hid the chosen process.

```
smerdy@smerdy-laptop:~/Desktop/rootkit-master$ ps
  PID TTY          TIME CMD
 1618 pts/0    00:00:01 bash
 2375 pts/0    00:00:00 ps
smerdy@smerdy-laptop:~/Desktop/rootkit-master$ ./client --hide-pid=1618
smerdy@smerdy-laptop:~/Desktop/rootkit-master$ ps
  PID TTY          TIME CMD
 2378 pts/0    00:00:00 ps
smerdy@smerdy-laptop:~/Desktop/rootkit-master$
```

## Detection

As with any computer threat, detection is key, rootkits are no exception. The issue lies with the nature of how they operate. When something is created whose purpose is to disguise and hide, it can be quite a problem identifying them. There are currently many different ways that rootkits can be identified. The major ways of detection are signature-based, heuristic/behavior-based, cross-view-based, integrity-based, and hardware-based. Often times antivirus software uses a combination of these different detection techniques to yield more accurate results.

## Signature-Based

Signature-based rootkit detection relies on the simple fact that rootkits tend to have similar tendencies. For example, when a rootkit attempts to unload itself from a system, there will still be traces of it. Antivirus software can sometimes detect the "fingerprints" left and recognize the system is infected (Todd 92). This is especially helpful if the rootkit in questing has been used for a significant amount of time or is well-published. However, this method is not as effective for rootkits that are built from the ground up and have not yet been identified.

## Cross-View-Based

Cross-view detection uses two different snapshots of the system and compares them to look for anomalies. One snapshot of the system is from a high-level view or of an external non-infected system; the other snapshot is of a much lower view. The high-level view is assumed to be the true view, and the low view is examined. If there is a difference between how they operate this could indicate an infection.

## Heuristic or Behavior-Based

Behavior-based detection uses inference to identify if a system has been infected. Antivirus software may track how long some actions take and what the CPU is doing. Let's say for instance a system call takes place, however, when tracing this call it does not follow the expected path. This could be an indicator of a rootkit. If a rootkit has a flaw this could also be detected, perhaps the rootkit is not completely obscuring itself from the process table, or that it crashes the system after an operating system update. Behavior-based detection is extremely complex and for this reason, brings up many false positives.

## Hardware-Based

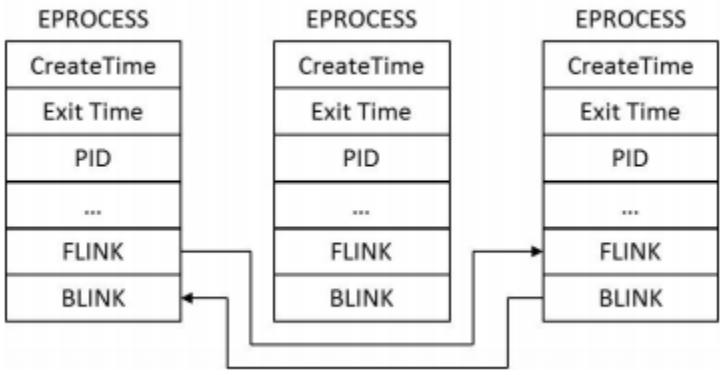
Hardware-based uses external hardware to essentially compete with the rootkit. Having external hardware allows the monitoring of activities at a much lower level, whereas, the software can only go so deep into the functioning of the system. Because external hardware is not directly connected to the possibly infected system, it is less likely to be modified by a rootkit (93). One example of a hardware-based detection device is a separate PCI card that monitors the operating system and kernel. This is much more effective than software-based programs because it does not rely on the system's OS to monitor.

## Stealth Techniques

There are a plethora of ways that a rootkit can hide itself with the main ones being hooking, patching, and direct kernel object manipulation (DKOM). Hooking exploits the way applications make function calls. Once an application makes a function call, it looks up the

address in an import address table (IAT), this is where the rootkit hooks. Instead of accessing the address, the rootkit will first execute its own code then jump to the original function. So while the original function is still executed, so it's the code from the rootkit. The be even more persistent, the rootkit itself can overwrite the code that the IAT points to (Flately 11). This ensures that even if the address is changed, the rootkit code will still execute.

Patching is similar to hooking but it does not directly modify the call tables. This helps the rootkit fly under the radar as there are no apparent changes. One way attackers use patching is to change the control flow of a program (14). For instance, instead of having the process jump to 0xAABBCCDD, the jump is to 0xBEEFBEEF which is where the malicious code is stored. Lastly, DKOM or direct kernel object manipulation is used to hide the rootkit. This method, however, is well known and therefore much easier for antivirus software to detect. DKOM can be more effective than hooking if done correctly. This is because it bypasses the kernel's object manager and with this any validation checks that the kernel does. Because DKOM only affects objects in memory, it can not hide files, but it does an excellent job of hiding processes (16). There is a list of currently running processes on a system, the rootkit is able to hide by changing the links in the list. The previous processe's forward link is then replaced with the link to the following process link. Then that process's backward link is changed to the previous process. This removes all links to the hidden process so it is undocumented and thus undetected.



## Source Code

Looking at a few different source codes from different rootkits, one key essential to the functionality is the *kallsyms_lookup_name* this function is used to find the address of the *sys_call_table* which is then edited to change its original functionality.

```
sys_call_table = (void **)kallsyms_lookup_name("sys_call_table");
pr_info("Found sys_call_table at %p\n", sys_call_table);
```

Sometimes it is not as easy as simply looking up the address to find the *sys_call_table*, in a different rootkit's source code it uses a heuristic to identify the *sys_call_table.* This snippet of source code searches through an entire address space starting with 0xc0000000 to 0xd0000000.

```
#if defined __i386__
    #define START_ADDRESS 0xc0000000
    #define END_ADDRESS 0xd0000000
```

```
void **find_syscall_table(void) {
    void **sctable;
    void *i = (void*) START_ADDRESS;

    while (i < END_ADDRESS) {
        sctable = (void **) i;
        if (sctable[__NR_close] == (void *) sys_close) {
            size_t j;
            const unsigned int SYS_CALL_NUM = 300;
            for (j = 0; j < SYS_CALL_NUM; j ++) {
                if (sctable[j] == NULL) {
                    goto skip;
                }
            }
            return sctable;
        }
skip:
        ;
        i += sizeof(void *);
    }

    return NULL;
}
```

The first if statement of the function checks an address can use *sys_close*, this is a close that only works from the user to the kernel space. So if the *sys_call_table* is at this address, then the function should succeed. In the for loop, the function checks if there are at least 300 system calls, with no pointers being NULL. If the address passes both of these tests, then the *sys_call_table* is most likely at this address. In summary, the rootkit scans through the addresses and looks for a table that can use both *sys_close* and has at least 300 calls inside it.

The rootkit also has functions that deal with hooking. The first is *hook_create* which replaces a function pointer at an address with a new function pointer while keeping track of the original function pointer.

```
int hook_create(void **modified_at_address, void *modified_function)
```

There is another function that is able to retrieve the original function pointer so that it can be called when inside a hook.

```
void *hook_get_original(void *modified_function)
```

Lastly, there is a function called *hook_remove_all* which restores all the overwritten function pointer in case the rootkit needs to be reversed.

```
void hook_remove_all(void)
```

## Conclusion

Rootkits will always be around and are very versatile pieces of software ranging from anti-theft protection to malware. They come in different forms such as bootkits and kernel-mode rootkits. Their purpose is to remain undetected and persist on the victim's system by using a variety of techniques such as hooking, patch, and direct kernel object manipulation to conceal themselves. Antivirus software currently employs a variety of tactics, usually in conjunction with others, to help detect and remove the rootkits. Hardware-based detection being the most effective as it is an outside system looking in. At the core of all rootkits, they change how the system call table works in order to redirect processes and operations. While not malicious in nature rootkits are an effective way for an attacker to maintain persistence on a system. Just like any other virus or malware, it is a constant cat and mouse game of virus detection against improved stealth. With no clear end in sight, they are here to stay and will continue to persist.

**Works Cited**

Flatley, Bridget. "Rootkit Detection Using A Cross-View Clean Boot Method". Air Force Institute Of Technology, 2013.

Thompson, Ken. "Reflections On Trusting Trust". Communications Of The ACM, vol 27, no. 8, 1984, pp. 761-763. Association For Computing Machinery (ACM), doi:10.1145/358198.358210.

Todd, A et al. Advances In Digital Forensics III. Springer, 2007, pp. 89-105.