ICMP-based Rootkit

Roman Bernikov*, Nazar Kononenko*, Ivan-Vitalii Petrychko*
* *Faculty of Applied Sciences of Ukrainian Catholic University, L'viv, Ukraine*

***Abstract*—The paper's aim is to introduce Rootkit – a collection of computer software designed to enable access to a computer without being detected. This package is designed strictly for educational and research purposes. Here, we will discuss our main findings and implementations. The paper consists of several sections:**

- **Introduction. This section provides a brief review of our rootkit, its capabilities, and core concepts.**
- **Implementation. Here, the main implementation choices are explained, as well as the reasons that lead to them.**
- **Performance impact. In this section, we compare the performance of different rootkit implementations.**
- **References. This page mentions all materials and articles that we've used in our work.**

I. Introduction

A rootkit is a type of malicious software designed to provide unauthorized access or control over a computer system while remaining hidden from detection. Rootkits typically operate in the kernel of a computer's operating system, giving them significant control while avoiding detection.

The unique aspect of our rootkit is its use of the Internet Control Message Protocol (ICMP), which allows us to send signals to infected computers. This functionality allows us to run specific software remotely wherever we want.

The operating system kernel is often modified to prevent threats from rootkits. Thus, we tried to create one that will use different approaches for different kernel versions to be more flexible for different Linux kernel versions.

II. Implementation

A. Loadable kernel module

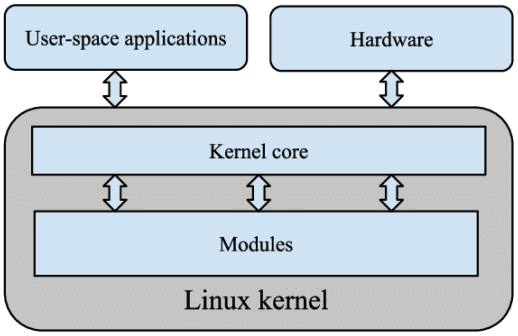


Fig. 1: Linux kernel structure.

A loadable kernel module (LKM), Fig. 1, is an object file that contains code to extend the running kernel, or so-called base kernel, of an operating system. LKMs are typically used to add support for new hardware (as device drivers) and/or filesystems or for adding system calls. When the functionality provided by an LKM is no longer required, it can be unloaded in order to free memory and other resources. As it runs in the kernel of an operating system, we can have access to many resources and, in the same way, be undetected. That's the ideal place to place our malicious software for a Rootkit.

B. Syscall table hijacking

The basic implementation we started with included changing the syscall table. Since the address of the table varies from machine to machine, we needed to find it in our runtime. To locate the table, we use the `syscall_kallsyms_lookup_name()`, which is defined in `kallsyms.h` and used to get the address of functions. Then, we replaced the address of the system call with the address of the system call we modified. This requires changing the CPU mode to allow to write to read-only pages in kernel mode (so-called ring 0). On x86 processors, the bit 16, "Write protect" of the CR0 register is responsible for this. There are no such registers for ARM processors, so our rootkit works only on x86 processors.

Here is a code example of how to do that:

```
static inline void write_cr0_forced(unsigned
    long val) {
   unsigned long __force_order;
   asm volatile(
      "mov %0, %%cr0"
      : "+r"(val), "+m"(__force_order));
}
static inline void protect_memory(void) {
   write_cr0_forced(cr0);
}
static inline void unprotect_memory(void) {
   write_cr0_forced(cr0 & ~0x00010000);
}
```

In general, our algorithm for hooking syscalls looks as shown in Fig. 2.

C. Implemented hooks and their functionality

As was previously said, we implement malicious logic by hooking syscalls responsible for various Linux commands. Here is a list of Linux syscalls that we used to hook, along with a description of the new functionality:

- sys_kill: hooking this syscall allows to intercept signal together with process ID contained in the registers **RSI** and **RDI** respectively. Our implementation filters signals by their value and various logic depending on it.
  Example of hooked signals:

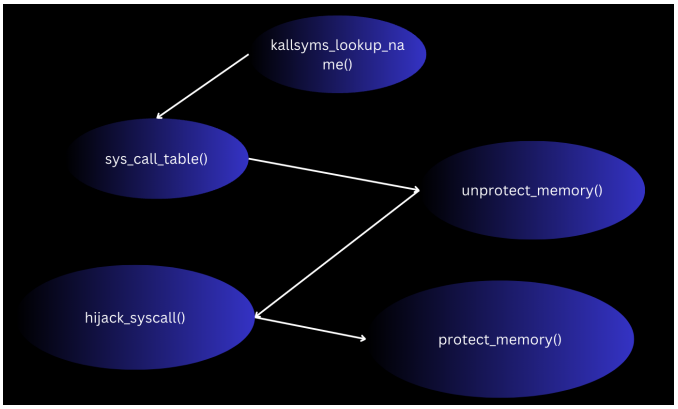


Fig. 2: Syscall Table hijacking algorithm.

```
#define HIDEMODULE 64 // Hides rootkit
    from LKM list
#define SHOWMODULE 63 // Returns rootkit
    back
#define HIDEPROCESS 62 // Hides specified
    by <pid> process
#define SHOWPROCESS 61 // Reveals hidden
    process
```

- sys_openat: hooking openat syscall allows to disable creating or opening files with specific prefix. The implementation consists of obtaining pathname from the pointer in the **RSI** register, filtering it for prefix presence, and if match – returning **-ENOENT** signal that will result in an exception for absent permissions.
- sys_getdents64: hooking getdents64 syscall allows hide files with specific prefixes and process with saved **PID**. Implementation obtains **dirent** struct, which represents a pointer to a buffer where directory entries are stored and file descriptor referring to the currently open directory. Hook implementation requires copying directory entries from user space to kernel space while returning filtered entries back.
- sys_unlink: hooking unlink syscall allows to forbid deleting a file from the system. When invoked, it receives a pointer to the path to the file in **RDI** register, so to implement the required logic will be enough to filter it for prefix presence.
- sys_execve: hooking execve syscall allows to block the execution of any program whose name contains the specified prefix (`antivirus` in our case). The logic is the same as for sys_unlink syscall: check if the path pointed by the **RDI** string contains a prefix and return **EACCES** signal if so.
- sys_rmdir: hooking rmdir syscall allows us to forbid deleting a directory that contains the specific word in the name (`virus` in our case). We check for name, stored in the string pointed by the **RDI** register and return -EACCES.
- sys_unlinkat: the unlinkat system call operates in exactly the same way as either unlink or rmdir so the logic of hooking is the same.

D. Ftrace

The hijacking of the syscall table involves modifications to the table itself, which is detectable. Additionally, the process of unprotecting memory is closely tied to the architecture of the processor, making it a delicate operation. Given also that the development for the Linux kernel varies significantly with each kernel version, we have adopted two distinct methodologies to address these challenges: hijacking syscall table and replacing original syscalls with hooks **(see here)** and by using ftrace – an internal tracer designed to help out developers and designers of systems to find what is going on inside the kernel.

The ftrace was originally created to attach callbacks to the beginning of functions to record and trace the flow of the kernel. We used those callbacks to hook function calls.

The main difference between the first and second approaches is that in the case of trace implementation, we don't directly change syscalls for our hooks, but instead, we work with a syscall wrapper that can be used for debugging, monitoring, live patching or, in our case, hooking. This allows us to inject our logic into the ftrace flow and replace the execution of the original syscall with our hook.

The main advantages of this approach are greater invisibility, as it is more difficult to track, and version independence (as we don't need to work with the syscall table and its permissions).

E. ICMP-Based Command Execution

In addition to hooking the system calls, our rootkit incorporates a feature leveraging Internet Control Message Protocol (ICMP) packets for remote command execution. ICMP is a fundamental protocol within the Internet Protocol Suite and is commonly used for network diagnostics. However, our rootkit repurposes ICMP packets as a covert communication channel for executing commands.

1) Netfilter: Netfilter is a powerful framework within the Linux kernel that facilitates network packet filtering, manipulation, and logging. It achieves this by providing a series of hooks at various points in the network protocol stack (Fig. 3). These hooks enable kernel modules to intercept network packets as they traverse through the system, allowing for customized packet processing.

The framework defines multiple hooks, each triggered at a specific point in a packet's journey through the kernel's network stack. These hooks include:

- NF_INET_PRE_ROUTING: This hook is the earliest interception point immediately after the network interface receives a packet before any routing decisions are made. It allows inspection and modification of incoming packets before the kernel processes them further.
- NF_INET_LOCAL_IN: Triggered for packets destined for the local host, following the routing decision.
- NF_INET_FORWARD: Engaged for packets that the kernel has decided to forward to another destination.

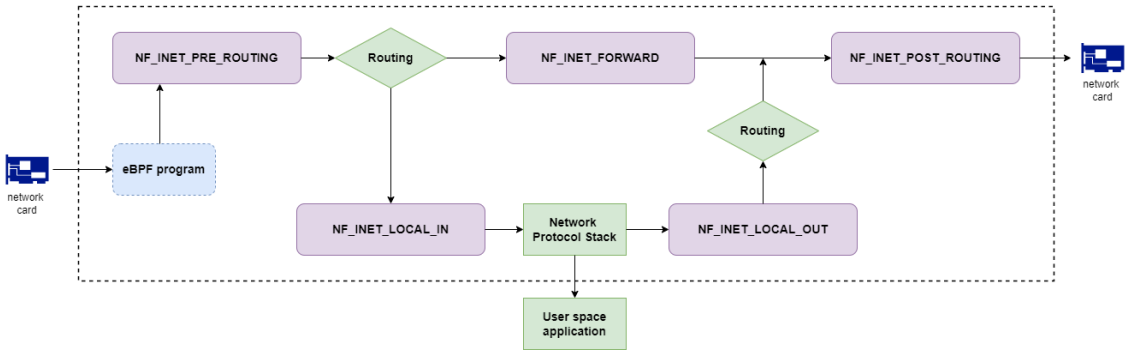


Fig. 3: Netfilter hooks in kernel.

- NF_INET_POST_ROUTING: Invoked for packets about to be transmitted out of the network interface after routing.
- NF_INET_LOCAL_OUT: Activated for outgoing packets generated by local processes.

2) Implementation: The functionality is implemented within the `icmp_cmd_executor` function, registered as a hook using Netfilter.

This function is invoked whenever an ICMP Echo Request (ping) is detected in the NF_INET_PRE_ROUTING hook. By placing our hook at NF_INET_PRE_ROUTING, we ensure that the rootkit intercepts ICMP packets as soon as they arrive at the network interface. The payload of the ICMP packet is parsed to extract a command prefixed with ”run:”. If such a command is identified, it is extracted from the payload and scheduled for execution in user mode.

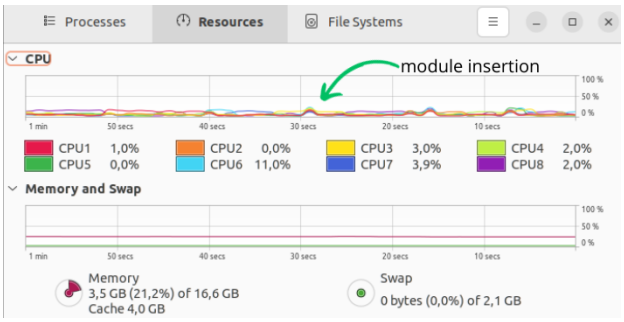
III. Performance Impact

We conducted a series of tests to evaluate the performance impact of our rootkit, specifically in terms of CPU usage. These tests were designed to measure and compare the CPU usage before and after the insertion of our rootkit module. Two scenarios were considered: one with no background applications running and another with various applications active.

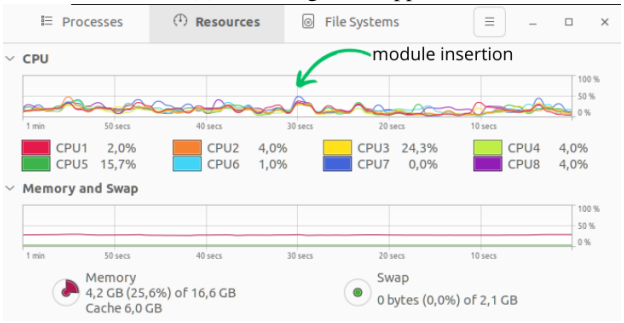
The performance impact analysis of our rootkit was conducted on a system with the following specifications:

- CPU: Intel Core i5-9300H CPU, 2.40GHz, 4 cores, 2 threads per core.
- Memory (RAM): 16GB DDR4.
- Graphics Card: GeForce GTX 1650 Ti Mobile: 4GB GDDR6.
- Operating System: Ubuntu 22.04.3 LTS, Kernel version: 6.2.0-39-generic.

The results from both scenarios indicate that our rootkit has a negligible impact on CPU usage. The small CPU usage spike can be seen at the moment of module insertion. This small performance impact is crucial for maintaining the rootkit's stealth and ensuring its persistence on the host system without arousing suspicion based on performance degradation.



(a) No background apps.



(b) With background apps.

Fig. 4: Performance impact.

The code of the project is available at https://github.com/nazar12314/Rootkit.

References

[1] Multiple ways to hook syscall(s). Available at https://foxtrot-sq.medium.com/linux-rootkits-multiple-ways-to-hook-syscall-s-7001cc02a1e6
[2] Hooking or Monitoring System calls in Linux using ftrace. Available at https://nixhacker.com/hooking-syscalls-in-linux-using-ftrace/
[3] Linux LKM Rootkit Tutorial. Available at https://www.youtube.com/watch?v=hsk450he7nI&t=12s
[4] Write a Linux firewall from scratch based on Netfilter. Available at https://levelup.gitconnected.com/write-a-linux-firewall-from-scratch-based-on-netfilter-462013202686.
[5] Linux syscall reference Available at https://syscalls64.paolostivanin.com/.
[6] Linux syscall hooking Available at https://ritcsec.wordpress.com/2020/11/22/linux-syscall-hooking/.
[7] Linux kernel communication Available at https://infosecwriteups.com/linux-kernel-communication-part-1-netfilter-hooks-15c07a5a5c4e.