Ways to Become EVIL
hiding a process on Linux 2.6

Zihao Chen*
ACM Honored Class, 2013
Shanghai Jiao Tong University
zihaochen1996@gmail.com

ABSTRACT
This article serves as a technical report on hiding a process on Linux 2.6 from `ps` instruction. It includes detailed steps to hide processes and some other methods as well. Readers are expected to have a basic understanding of the concepts of operating systems (e.g. system call).

Keywords
hiding processes, Linux kernel, hijacking system calls

1. INTRODUCTION
The technique of hiding processes has broad applications in computer security, especially in building building rootkits/viruses. When attacker breaks into a computer system, it is quite important to hide his processes from being detected. In this article, we present our method to hide a process from `ps` instruction by hijacking system calls and the method works well on 32-bit Ubuntu 10.04, whose Linux kernel version is 2.6.32. There are also several other ways to hide a process on Linux and some of them will be briefly introduced as well.

2. HOW PS WORKS
To figure out how to hide a process from `ps`, we must understand how `ps` works. As pointed out in the blog [4], instructions like `ps` leverage the `/proc` file system. First, the directory `/proc` is opened via the `openat()` system call. Then, the process calls `getdents()` on the opened directory, which is a system call that returns the list of files/directories contained in a specific directory(`/proc` in this case). If we run `ls /proc` , we will notice that there is a subdirectory for every running process in the system, and each directory is named by the PID of the process itself. So, `ps` will just grab the list from `getdents()`, and then iterate over a fixed set of files in each subdirectory. The output of `ps` is the information of processes, which are grabbed from `/proc/PID/stat`, `/proc/PID/status` and `/proc/PID/cmdline`.

*ID: 5130309305

3. WAYS OF HIDING PROCESSES
In this section, we will briefly introduce several ways of hiding processes on Linux and compare their advantages and disadvantages.

The first way is to grab the source code of `ps`, implement my own "`ps`" and then recompile the source. This method is direct but it needs much work and is quite time-consuming.

The second way is to modify the C standard library(`libc`). In fact, during the execution of `ps`, the process itself doesn't directly call `openat()` and `getdents()`, as those are system calls that are abstracted by `libc`. Instead, `libc` provides two functions, `opendir()` and `readdir()`, and they take care of calling the systems calls themselves. In other words, these two functions are the functions that are directly called form `ps`. Therefore, we could modify the `readdir()` function inside `libc` to hide some `/proc` files. However, the `libc` code is pretty hard to understand and recompiling `libc` is a burden.

The third way may be the most advanced way, we could hijack and modify the `getdents()` system calls. Fortunately, we do not need to recompile the kernel source — Linux provides the technique of LKM(Loadable Kernel Module) and we could simply use a custom kernel module to satisfy our needs. This is a pretty efficient way to hide processes and the technique of hijacking system calls has broad applications in computer security.

4. OUR METHOD
4.1 Overview
We took the third way of hiding processes mentioned in the above section, that it, by hijacking system calls. Briefly speaking, our program takes the following steps:

1. Finding the `sys_call_table`.
2. Obtaining the privilege of writing to read-only areas.
3. Replace our `hacked_getdents()` with the original `getdents()` system call.

Don't worry if you haven't heard about the concepts mentioned above (e.g. `sys_call_table`), we will explain them in detail in the following parts.

4.2 Finding the sys_call_table

A system call, which is a job for the operating system kernel, is identified by a number that is used as an index in a table with function addresses. These addresses point to functions that implement the respective system calls. In the Linux kernel, the table is `sys_call_table`. To hijack some specific system call, the attacker could replace the respective function address with the address for their own malicious functions.

Unfortunately, kernel versions 2.6 and 3.0 made this attack more difficult, the `sys_call_table` address is no longer defined globally and thus it is unknown to a kernel module.

However, the `sys_call_table` lies in a segment whose address is fixed and we have several indirect ways to find this address. We will present several ways below and the last way is what we took, you could directly look at the last way if you merely want to know how did we do.

The first way is to find a list of all kernel addresses on the computer and the list is located in `/boot/System.map-<kernel version>`, which represents the mapping used by the kernel form symbol to their address locations in the memory. However, this approach do not have portability as it requires to hardcode the system call address table.

The second way is to use Interrupt Descriptor Table(IDT) to find the address of `system_call()` function, which yields the address of `sys_call_table` by a little bit searching job. Briefly speaking, the attacker could first read the Interrupt Descriptor Table Register(IDTR) and then obtain the address of IDT. Next, find the address of the `system_call()` function at the index 0x80 of the IDTR. Finally, the attacker search a few bytes in memory using assembly code. This approach is much more simple than the first one and easy to implement and in fact, many programmers take this approach to find the `sys_call_table`.

The last way is the way we took, as described in the blog[5], it is not so sexy but fully functional despite architectures and cpu's being used — the technique of brute-forcing. We define a kernel memory range that the `sys_call_table` can lie in, then we compare the pointer with a symbol that is exported as `sys_close`. We choose the offset as the constant `__NR_close` and when our pointer + offset match the `sys_close` value, we could find the `sys_call_table`.

4.3 Writing to read-only areas

After we find the `sys_call_table`, we are going to modify pointer to the system call `getdents()` in the table. However, Linux kernel 2.6 uses a security system that do not allow us to write the `sys_call_table` at runtime and this is controlled by the Write Protected(WP) bit. The WP bit is controlled by the control register `cr0` and the WP bit is actually the 16-th bit of the `cr0` register. To enable writing to read-only areas, we could simply modify the 16-th bit of the `cr0` register.

4.4 Hacking the `getdents()` system call

Next we will build our own hacked `getdents()`. To do this, we should first figure out what the original `getdents()` do.

```
int getdents(unsigned int fd, struct
    linux_dirent *dirp, unsigned int count
    );
```

The system call `getdents()` reads several `linux_dirent` structures from the directory pointed at by `fd` into the memory area pointed to by `dirp`, the parameter `count` is the size of the memory area.

`linux_dirent` stands for the directory entry in Linux and is declared as follows:

```
struct dirent {
    /* inode number */
    long d_ino;
    /* offset to next dirent */
    off_t d_off;
    /* length of this dirent */
    unsigned short d_reclen;
    /* filename  */
    char d_name [NAME_MAX+1];
}
```

After we call the original system call `getdents()`, we could visit the directory entries by the pointer `dirp`. Then we search among the directory entries and if we find the `linux_dirent` of the processes we want to hide, we could simply wipe out the corresponding `linux_dirent` structures from `dirp`. In Linux, each process has its directory named by its pid under the directory `/proc`. If we tell our program the pid of the processes we want to hide, we just need to compare the pids. But if we only tell it the names of the processes, we need to get the process name with the current pid during the search. Fortunately, Linux provides some functions and we can easily obtain the task structure of the process, which contains its name:

```
struct  task_struct *task = pid_task(
    find_vpid(pid), PIDTYPE_PID);
```

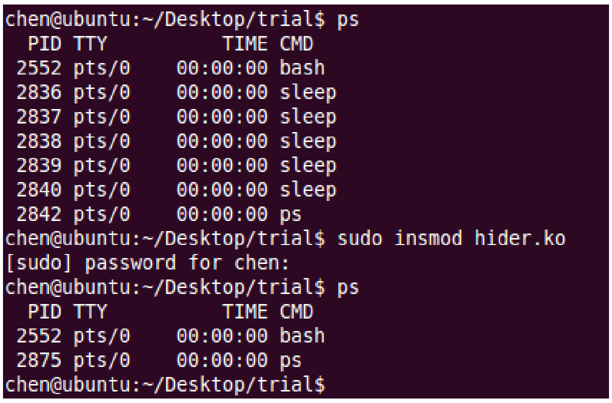
Till now, we have done most of the work for building our own `hacked_getdents()`. To make it more robust, we should not do the search if we are not in the `/proc` directory. As `/proc` is a special file system which only lies in the memory but not any other devices, it has a specific major device number 0 and minor device number 3. On top of that, it has a special inode number `PROC_ROOT_INO` of 1. Therefore, we could get whether we are in `/proc` by this:

```
struct kstat fbuf;
vfs_fstat(fd, &fbuf);
/*if the directory is not /proc we do
    nothing*/
if (!(fbuf.ino == PROC_ROOT_INO && !
    MAJOR(fbuf.dev) && MINOR(fbuf.dev)
    == 3))
     return value;
```

So far we have finished all the work of our own `hacked_getdents()`.

5. EXAMPLE

In this part we present an example of how our program works. We support the feature of hiding many processes



```
chen@ubuntu:~/Desktop/trial$ ps
  PID TTY          TIME CMD
 2552 pts/0    00:00:00 bash
 2836 pts/0    00:00:00 sleep
 2837 pts/0    00:00:00 sleep
 2838 pts/0    00:00:00 sleep
 2839 pts/0    00:00:00 sleep
 2840 pts/0    00:00:00 sleep
 2842 pts/0    00:00:00 ps
chen@ubuntu:~/Desktop/trial$ sudo insmod hider.ko
[sudo] password for chen:
chen@ubuntu:~/Desktop/trial$ ps
  PID TTY          TIME CMD
 2552 pts/0    00:00:00 bash
 2875 pts/0    00:00:00 ps
chen@ubuntu:~/Desktop/trial$
```

Figure 1: An example

with the same given name at one time. In this example, we ran several processes called *sleep*. As is shown above, after we installed the module into kernel, we successfully hid all the processes called *sleep*!

6. REMARKS AND ACKNOWLEDGMENTS

This project is the final project of the Operating System course MS110 in SJTU. Thanks to Prof.LIANG for explicit instructions, and thanks to TAs Kun WANG and Ximing TANG for providing us such a chance to have a better understanding of Linux operating system.

7. REFERENCES

[1] Tanenbaum, Andrew S. "Modern operating systems." (2009).

[2] Bovet, Daniel P., and Marco Cesati. Understanding the Linux kernel. " O'Reilly Media, Inc.", 2005.

[3] Stevens, W. Richard, and Stephen A. Rago. Advanced programming in the UNIX environment. Addison-Wesley, 2013.

[4] Hiding Linux Processes For Fun And Profit https://sysdig.com/hiding-linux-processes-for-fun-and-profit/

[5] Modern Linux Rootkits 101 http://turbochaos.blogspot.hk/2013/09/linux-rootkits-101-1-of-3.html

[6] The Linux Kernel Module Programming Guide http://www.tldp.org/LDP/lkmpg/2.6/html/index.html

[7] Linux 2.6 Hijacking System Calls and Hiding Processes (in Chinese) http://blog.csdn.net/billpig/article/details/6196163

[8] Linux Kernel Documentation https://www.kernel.org/doc/