Rootkit of Gruppe 6

TUM
Chair for IT Security
Rootkit Programming 2011/2012

Roman Karlstetter Philipp Müller

February 7, 2012

Contents

1. Overview

This article describes a rootkit that has been implemented as a Linux Loadable Kernel Module (LKM) for the Linux kernel 2.6.32 during the Wintersemester 2011/2012 for the course Rootkit Programming at TUM supervised by Jonas Pfoh (`http://www.sec.in.tum.de/rootkit-programming-ws1112/`).

In the first part of the course, we step by step implemented a rootkit with several features listed below (detailed description of implementation: section 3). The next thing was to develop a program which is capable of detecting a rootkit of another group, you can find the basic ideas in Appendix A. In Appendix B, we shortly describe how we implemented finding the syscall table dynamically without accessing the `System.map`-file. The last challange was to adopt our own rootkit such that it does not get detected by a detector written by Gruppe 5 for our rootkit (see section 4 & section 5).

This rootkit implements several functions, that can – independently of each other – be turned on and off. These features are:

File hiding It is possible to prevent some files to be shown. Files beginning with a certain prefix are hidden from e.g. `ls`. However, these files stay accessible (subsection 3.3).

Process hiding It is possible to hide certain processes by their ID (subsection 3.4).

Module hiding It is possible to hide the rootkit module from `lsmod` (subsection 3.5).

Socket hiding The rootkit can hide certain TCP and UDP sockets from `netstat` and `ss` (subsection 3.6).

Privilege escalation You can acquire the privileges to act as root (subsection 3.7).

Covert communication All this functionality is turned on and off via a covert communication channel (subsection 3.8).

2. Building, installing & using the rootkit

2.1. Building the module

You need to have a working environment for developing kernel modules. When this is the case, `cd` to the directory with the sources and execute `make` to build the module. This will create a file called `cool_mod.ko`, which contains the compiled module.

2.2. Loading the module

The module is loaded like (probably) most other modules: using `insmod`. That is, you can just do the following (of course, you need to be root):

```
root@machine:/path/to/rootkit# insmod cool_mod.ko
```

Then, the module can be used. The module itself does quite a bit of output for the user, which can be used using `dmesg`. If you try `dmesg` directly after loading, somewhere at the end of the output you should see a line like this:

```
[   108.693480]  This is the kernel module of gruppe 6.
```

2.3. Using the module

Once the module is loaded, it can receive commands as you type them into a shell. Every command starts with the prefix `### ` (three hashes, followed by one space).

Once this sequence of characters is entered, the rootkit expects the actual command. After the actual command, there might be a parameter (separated by a space from the command). Thus, the general format for commands is simply as follows:

```
###␣command␣params␣
```

`hideproc` This command expects a parameter representing the process ID of the process to be hidden. Hides the corresponding process.

`unhideproc` This command expects a parameter representing the process ID of the process to be unhidden. Unhides the corresponding process.

`hidemodule` Prevents the module from being shown when executing `lsmod`. Note that when the module is hidden, it can not be unloaded.

`unhidemodule` Unhides the module, i.e. `lsmod` is finding the module.

`hidetcp` Hides a TCP socket from `netstat` and `ss`. Expects one argument: The port of the socket to be hidden.

`hideudp` Hides a UDP socket from `netstat` and `ss`. Expects one argument: The port of the socket to be hidden.

`unhidetcp` Disables socket hiding for a TCP socket. Expects the port as argument.

`unhideudp` Disables socket hiding for a UDP socket. Expects the port as argument.

`hidefiles` Hides files, whose filename is starting with `rootkit_`, i.e. these files are “invisible” to `ls`, but can still be accessed.

`unhidefiles` Disables file hiding. I.e. basically all files are shown then (except those that are hidden by another rootkit ;)).

`escalate` This command is most useful when invoked by a non-root-user. Once the module is loaded, *any* user can just type `### escalate` and obtains `root`-rights.

As an example, consider how to hide TCP socket 1234:

```
root@machine:/path/to/something# ###␣hidetcp␣1234␣
```

Note that you don’t need to confirm the command by pressing return. The rootkit recognizes the command and the parameter after you entered a whitespace character.

Table 1 gives a quick overview over all available commands for the rootkit.

Command	Parameter	Example
`hideproc`	PID	`###␣hideproc␣1234␣`
`unhideproc`	PID	`###␣unhideproc␣1234␣`
`hidemodule`	-	`###␣hidemodule␣`
`unhidemodule`	-	`###␣unhidemodule␣`
`hidetcp`	Port	`###␣hidetcp␣1122␣`
`hideudp`	Port	`###␣hideudp␣3344␣`
`unhidetcp`	Port	`###␣unhidetcp␣1122␣`
`unhideudp`	Port	`###␣unhideudp␣3344␣`
`hidefiles`	-	`###␣hidefiles␣`
`unhidefiles`	-	`###␣unhidefiles␣`
`escalate`	-	`###␣escalate␣`

Table 1: Commands for the rootkit

2.4. Unloading the module

To unload the module, you have to execute

```
root@machine:/path/to/rootkit# rmmod -w cool_mod.ko
```

It is necessary to specify the `-w` option, as the module makes heavy changes to the read system call which is nearly all the time in use. Therefore, we have to wait until there are no users of our module left.

3. Implementation Details – Initial Version

This section first gives an overview on how the module is designed and then describes the technical details of each individual part of the module.

Note that this section deals with our initial implementations. After we got the rootkit detector of gruppe 5, we revised some functions.

3.1. General Design

We have split up things in "submodules" (each with its own header and implementation file), so that our we have some nice modular architecture. This makes our module easily extendable. Everything gets glued together in `mod.ko`.

The `hook_read` submodule reads from stdin and feeds this information into the `covert_communication` submodule. In addition to this, we specifiy which commands correspond to which functions of the other submodules. This is done by the `add_command`-function of the `covert_communication` submodule. When a command (and its parameter) is recognized, the corresponding function gets called.

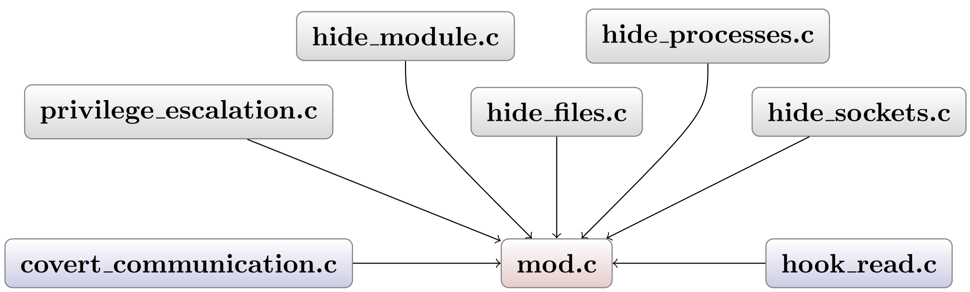


Figure 1: The general design of the module. Everything get's glued together in mod.c, where you can easily specify commands which can be used to execute functions of one of the submodules.

The file names should be self-explanatory, but we should mention some things explicitly.

The files `sysmap.h` and `sysmap.c` are generated automatically by the script `create_sysmap.sh`. This script scans the "system map"[1] of Linux. The system map is a file, where all kernel symbols are listed. All these pointers are collected and made usable to our code. The script prepends each found pointer with the prefix `ptr_`. So, if you encounter a variable starting with this prefix, chances are good that it is something from the system map.

Note that all these pointers are of type `void*`, since we do not know their type a priori. They are later casted as needed.

Remark: In the following, we present some code snippets. They are basically taken from our rootkit source code, but we shortened them a bit and describe just the crucial parts of them.

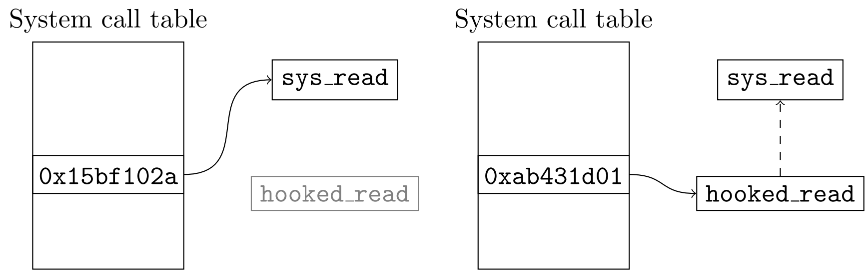
3.2. "Hooking" functions in general

A very important aspect of a rootkit is hooking of functions. This means, we "grab" an original system function (which may be called by other programs!), and replace it by our own function. This way, everytime another program tries to call the original function, it automatically calls *our* "injected" function. Figure 2 shows this mechanism as a schema.

Of course, the injected function should behave as if it was the original function, so that the other programs (and, thus, the user) do not recognize that we have manipulated the system function.

So, how is this hooking done programmatically? Let us consider how we hook the `read` system call, for example. This is done by the following code snippet:

[1] In our case, the system map resides at `/boot/System.map-2.32`. This system map needs not to exist, but luckily it is existent on our test machines.



(a) Before hooking. Pointer to the original `sys_read` system call.

(b) After hooking. Pointer to `hooked_read`, which itself can use `sys_read`.

Figure 2: Hooking a system call function

```
void hook_read(fun_void_charp_int cb){
    void** sys_call_table = (void *) ptr_sys_call_table;
    original_read = sys_call_table[__NR_read];
    make_page_writable((long unsigned int) ptr_sys_call_table);
    sys_call_table[__NR_read] = (void*) hooked_read;
    make_page_readonly((long unsigned int) ptr_sys_call_table);
}
```

Let us consider this function line by line. First, we locate the system call table, which is basically just an array of pointers to system call functions. Afterwards, we retrieve the pointer to the original `read` system call and store it in `original_read`[2]. This is necessary for several reasons: First, we want to be able to restore the original call if we unload the module. Moreover, as we will see later, we delegate the work to the original `read` function and just do some manipulations beforehand.

Let us – for now – skip the next line and directly consider the following line from the above snippet:

```
    sys_call_table[__NR_read] = (void*) hooked_read;
```

This line *modifies* the system call table in such a way that if some program is issuing a system call to `read`, it will actually invoke our own function `hooked_read`. The constant `__NR_read` is defined as a macro somewhere in the Linux kernel source code and denotes the position in the system call table, where the function pointer to the `read` system call is stored.

The above line is the most crucial one when it comes to system call hooking, since it actually sets the function pointer to our function.

So the only point left is what the functions `make_page_writable` and `make_page_readonly` are doing.

As it turns out, Linux 2.6 has some memory protection mechanism so that you can not modify certain memory sections (e.g. and in particular the system call table) at will. Instead, one has to first remove this write protection. This is exactly what `make_page_writable` does. It is defined in `global.c` as follows:

```
void make_page_writable(long unsigned int _addr){
    unsigned int dummy;
    pte_t *pageTableEntry = lookup_address(_addr, &dummy);
    pageTableEntry->pte |=  _PAGE_RW;
}
```

[2] `original_read` is a function pointer of type `asmlinkage ssize_t (*)(unsigned int, char __user *, size_t)`. This is a pointer to a function returning a value of type `size_t` and taking three arguments, just like the original `read` system call declared in `linux/syscalls.h` in the Linux kernel.

That is, make_page_writable first retrieves the page table entry which contains a given address (_addr). Afterwards, it allows this page to be written (by ORing with _PAGE_RW, wich is a macro defined in the linux kernel).

Hooking functions is basically done in this way. Note that the functions we want to hook need not necessarily reside in the system call table, but can be at other locations as well. The mechanism of hooking, however, is equivalent.

However, as we will see later, there are other techniques to hook a system call (see subsection 5.1).

3.3. File Hiding

We hook getdents64 (resp. getdents) to hide files beginning with the prefix rootkit_. Therefore, we simply call the original getdents-function and iterate over the resulting dirents. As soon as we find an entry whose d_name begins with rootkit_, we "erase" it by moving the remaining part forward in memory. This process is shown in Figure 3.

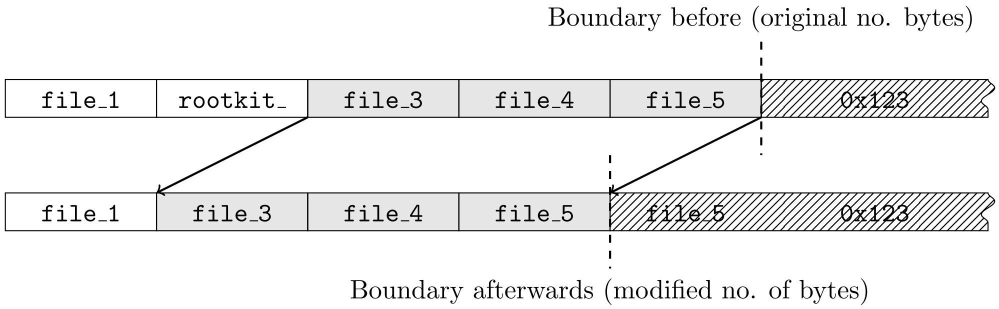


Figure 3: The file rootkit_ is overwritten by copying the remaining buffer forward. Note that the number of bytes returned are adjusted (indicated by boundaries).

Another implementation alternative would have been to manipulate the element preceeding the entry to be hidden, such that it appears to be longer and thus kind of shades the item beginning with rootkit_. Each dirent has a member d_reclen, which stores the length of the dirent itself. Thus, we also could manipulate d_reclen accordingly.

However, this alternative approach brings quite some overhead in code, because you need e.g. additional tests whether the element to be hidden is the first element (the hard case) or somewhere in the middle (that works quite well).

3.4. Process Hiding

Currently, we hook the readdir call of the proc-filesystem to hide the specified tasks (just as in subsection 3.3). Moreover, we also employ a custom filldir function[3], which actually hides the specified processes.

The hooked readdir function is quite simple:

```
int hooked_readdir(struct file *filep, void *dirent, filldir_t filldir){
    original_proc_fillfir = filldir;
    return proc_original_readdir(filep, dirent, hooked_proc_filldir);
}
```

First, we remember the original filldir function so we can use it later. Afterwards, we call the *original* readdir function, but we give our own hooked_proc_filldir as parameter to it. So, we "just change parameters" to the function.

It remains to show our hooked filldir function:

[3]filldir_t is an own type in the Linux kernel for a function used to fill the contents of readdir.

```
int hooked_proc_filldir (void *__buf, const char *name, int namelen,
                         loff_t offset, u64 ino, unsigned d_type){
    char buf[512];
    struct proc_to_hide* cur;
    for(cur = processes_to_hide; cur != 0; cur = cur->next){
        sprintf(buf, "%d", cur->pid);
        if(strcmp(buf, name)==0){
            // If we come here, we hide the process.
            return 0;
        }
    }
    // If we come here, we just delegate everything to the original function.
    return original_proc_fillfir(__buf, name, namelen, offset, ino, d_type);
}
```

As you can see, these kind of functions accept quite a bunch of arguments. But we need only one of them: `name` – it holds a PID (formatted as a array of `char`s).

All the other parameters are possibly passed to the original `filldir` function.

The function does essentially the following: It traverses the list, where we store the process IDs of the processes we want to hide. If it finds that the current PID is to be hidden, it just returns 0, indicating that the current call did not produce any output. This causes the callee just to ignore the corresponding process. If no such PID is found, it just delegates the work to `original_proc_filldir`, which then acts just like there was no interception at all.

Another way to hide processes is to remove the respective task from the hash table which is used for looking up the tasks and generating the entries in proc.

3.5. Module Hiding

To hide the module, we

- remove it from the list of modules and
- hook the `readdir` operation of the sysfs.

The first point is pretty straightforward, once you have the pointer to the list of modules. The following line accomplishes this:

```
list_del(&(THIS_MODULE->list));
```

This line relies on some macros defined in the Linux kernel sources: `list_del` removes an item from a linked list and `THIS_MODULE` is a pointer to a `struct module` representing the module it is invoked from. So, the line removes *our* module from the list of modules.

To remove the module from `/sys/module`, you have to hook also create a `filldir`-function which gets called by the hooked readdir operation (for the `sys`-filesystem this time) and returns 0 when called with this module as parameter. This mechanism is similar to the one described in subsection 3.4.

In order to be able to unload the module again, we have to make is visible, otherwise `rmmod` tells us that such a module is not loaded. So, to make it visible again, we have to insert it again into the list of modules. Fortunatly, our `sysmap.h` has an entry `ptr_modules` which points exactly to the desired list. By calling

```
list_add(&(THIS_MODULE->list), (struct list_head *) ptr_modules);
```

the module gets inserted again and we can unload it with rmmod (additionally, we of course have to remove the hook for the `sys`-filesystem).

3.6. Socket Hiding

Hiding sockets means that they are not displayed by neither `netstat` nor `ss`. `netstat` gets all its information through `/proc`, while `ss` gets the TCP-socket information through the NETLINK interface of the kernel (when this is available), the UDP information is also from `/proc`.

We implement socket hiding by hooking the `show`-operation of the `seq_file` interface (only of tcp and udp socket-files in `/proc/net`, of course). Therefore we look at the port and the protocol of the passed `struct socket` and compare it with the socket to hide.

For TCP sockets displayed by `ss`, we hook the `socketcall` function and return an error code when it gets called with a special combination of parameters. This lets `ss` also use its fallback alternative, which is `/proc` as already described.

3.7. Privilege Escalation

Privilege escalation is done using a simple mechanism: We set the user id of the current process to 0 (which corresponds to user `root`). This is done by the following code snippet:

```
void escalate(void){
    struct cred *my_cred;
    my_cred = prepare_creds();
    my_cred->uid = my_cred->euid = my_cred->suid = my_cred->fsuid = 0;
    my_cred->gid = my_cred->egid = my_cred->sgid = my_cred->fsgid = 0;
    commit_creds(my_cred);
}
```

The function `prepare_creds` creates a copy of the current threads credentials (i.e. user id, etc). We work on this copy (stored in `my_cred`), and – after modifying it – apply the changes using `commit_cred`.

While one could think it would be more convenient to directly modify the task's credentials, this is explicitly discouraged in the linux source code[4].

3.8. Covert Communication

We hook `sys_read` and intercept all input from standard in.

First, we try to match a magic activating command (this is the three hashes and a space, followed by one of the commands described in subsection 2.3). After this command, we read until a whitespace occurs. After that, we analyze what has been entered and execute the corresponding command, if

- it is available and
- it does not expect an argument.

If it does expect an argument (this is always a number), we read again until a whitespace and call the function with the parsed number as argument.

4. Gruppe 5's Rootkit Detector

We received a rootkit detector for our rootkit that detects three aspects of our rootkit. This section gives an overview about how detection works.

Hooked Read Function As mentioned in Figure 2, we hook `sys_read` by changing the function pointer in the system call table. This, however, leaves suspicious traces, since a detector can simply extract the original adress from the `system.map` file. This is exactly what the detector of gruppe 5 does. However, for this to work correctly, the detector has to be prepared *before* the rootkit gets loaded. Under the assumption that the rootkit is loaded before the detector (and that the detector cannot do any analysis

[4] File `kernel/cred.c`

or preparation thereof in a clean system), the rootkit could simply hook the `system.map` file such that the values in this file correspond to the actual situation when the rootkit is active.

Subsection 5.1 describes how we evade this detection.

Hooked socketcall In our initial version, we simply hooked the `sys_socket` function[5], and simply checked whether the arguments to this function are the ones given by the program `ss`. If so, we would simply return -1, so that `ss` uses its fallback.

Thus, the rootkit detector simply does the following:

```
result = socket(AF_NETLINK, SOCK_RAW, NETLINK_INET_DIAG);
```

Afterwards, they only have to check whether `result` is negative. If so, it indicates that our rootkit is present. Subsection 5.2 describes how we can hide sockets from `ss` using another technique.

Hidden Module Two things were done to hide the module (see subsection 3.5), the detector uses a weakness of the implementation used to remove the entry from the sysfs. Up to now, the rootkit just didn't show the corresponding entry in the `/sys/module` by overwriting the filldir function and using it in the hooked readdir function. However, the corresponding datastructures of `sysfs` were still present. What the detector now does is just the following: it iterates over the internal `sysfs_dirent`s of the list of modules in the `sysfs`-datastructures and counts the number of modules in this list. The result is the correct number of currently loaded modules, including our rootkit. Then, it compares this number with the number of lines that a simple

```
ls -1 /sys/module/
```

produces, which is the one less than the actual number, as our module doesn't show up in this list. In subsection 5.3, we describe what can be done to avoid detection this way (although you might already have a guess).

5. Detection Evasion

As we really don't want our rootkit to be detected by the detector of Gruppe 5, we have some great ideas how to make our rootkit even better so that we finally can achieve world domination. These ideas are described in this section.

Thanks to our modular architecture, the changes could be done separately in the different submodules, such that the old code didn't have to be changed or adapted. This is true even for such fundamental things as the hooked `sys_read`, which will be described first.

5.1. Hooking `sys_read` without changing the system call table

5.1.1. Installing the hook

This subsection describes how we manipulate the `read` system call without altering the function pointer in the system call table. While in the original version, we simple redirected the call from the system call table directly to our hooked function (as illustrated in Figure 2), now, we do it another way: We directly modify the code of the original read-system call that resides in memory.

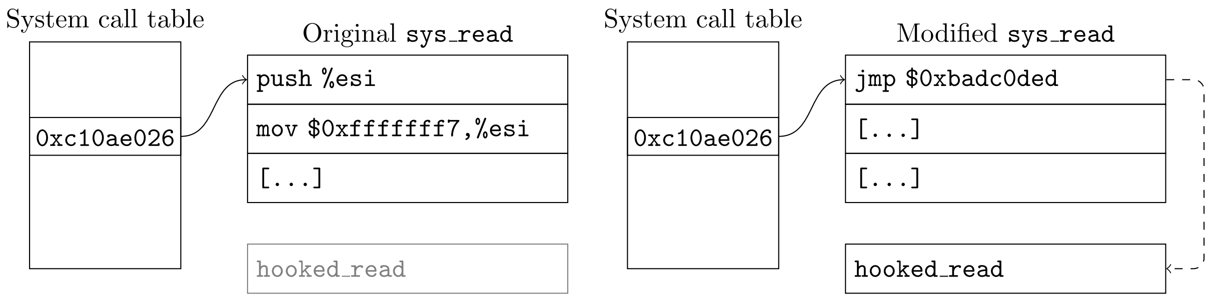
To do so, we inject some custom instructions into the original machine code for the `read` system call. These injected instructions will cause the machine to jump to *our* hooked function, even if we leave the system call table as it is. This principle is shown in Figure 4.

To understand how this works, let us first consider some disassembly of the `sys_read` function. We just want to have a look at the first some bytes[6] of this function:

[5]We actually hooked `socketcall`, because `sys_socket` isn't accessible directly, but every call to `sys_socket` is handled by the function `socketcall`.

[6]You can produce this output with the following:

```
gdb /usr/src/linux-<version-number>/vmlinux
(gdb) disass sys_read sys_read+40
```



(a) Before hooking. Pointer to the original `sys_read` system call. You see some machine instructions.

(b) After hooking. Syscall table stays untouched, but code of `sys_read` is modified, so `sys_read` itself redirects call to `hooked_read`. Note that the address of the modified `sys_read` is the same as the address for the original `sys_read`

Figure 4: Inline-modification of the read-system call

```
0xc10ae026: push   %esi
0xc10ae027: mov    $0xfffffff7,%esi
0xc10ae02c: push   %ebx
0xc10ae02d: sub    $0xc,%esp
```

We don't need to examine what these instructions do exactly, but let us consider what we need to do in here to redirect the function call to somewhere else (namely to our `hooked_read`).

```
0xc10ae026: jmp    0xbadc0ded       ;jump to the function of the rootkit
0xc10ae02b: [...]                   ;here we still have the original bytes
```

It remains to explain how we can alter the machine instructions of `sys_read` at runtime. This is done using the following code snippets[7]:

First of all, we recognize that for the Jump-instruction we want to insert, we need five bytes (one byte for the JMP, and four bytes for the address). So we do

```
#define SYSCALL_CODE_LENGTH 5
```

Furthermore, we have to do two things: We have to remember the original bytes that reside in memory, so that we can restore them when unloading the module. Moreover, we create an auxiliary array that holds the 5 bytes needed for our Jump-instruction to be inserted:

```
static char syscall_code[SYSCALL_CODE_LENGTH]; // the original bytes go here
static char new_syscall_code[SYSCALL_CODE_LENGTH] = "\xe9\x00\x00\x00\x00"; /* jmp $0 */
```

`syscall_code` will later hold the *original* bytes from the system call, while `new_syscall_code` holds the bytes representing the jump-instruction to be inserted.

Note that – up to now – we would just jump to 0. This is of course not what we want. We want to jump to `hooked_read`. To tell the program so, we do the following:

```
*(int *)&new_syscall_code[1] =
 ((int)hooked_read) - ((int)sys_call_table[__NR_read] + 5);
```

[7] Note that this is not a verbatim excerpt of the code, but just the essential parts for hooking shown together.

This sets the last 4 bytes to the corresponding address. Note that we have to compute the difference between `hooked_read` and `sys_call_table[__NR_read]`, since the jump is relative.

Note that – as soon as the actual jump occurs – the instruction pointer is already 5 bytes after the beginning of the jump-instruction (because this jump-instruction occupies 5 byte). This is where the 5 in the code comes from.

So, it remains to backup the original bytes and install the jump-instruction. Therefore, we wrote a small auxiliary function `_memcpy` that behaves (or at least should behave) just like the well-known `memcpy`. It takes one additional parameter, if this is 1, the function handles write-protected memory sections properly.

```
_memcpy( syscall_code, sys_call_table[__NR_read], sizeof(syscall_code), 0);
_memcpy( sys_call_table[__NR_read], new_syscall_code, sizeof(syscall_code), 1);
```

5.1.2. How the new `hooked_read` works

Now that we've installed the hook properly, we can think about how the new hooked function works. In principle, it works like the old `hooked_read`: It calls the original `sys_read`, checks its return values[8], does some processing, and returns the value received from the original `sys_read`.

The only problem arising here is that the original `sys_read` isn't original anymore! We modified it such that it automatically jumps to `hooked_read`. Therefore, we have to do some work when initializing the hook: As said before, we manipulate the first 5 bytes of `sys_read` and create a backup of them. We moreover will use these five bytes to "simulate" the original run of the `sys_read` function.

Let us for now concentrate on how the `sys_read` normally takes place on assembly level. Therefore, we consider again some disassembly for the `read` system call:

```
0xc10ae026: push   %esi                ;; gets overwritten
0xc10ae027: mov    $0xfffffff7,%esi     ;; gets overwritten
0xc10ae02c: push   %ebx
0xc10ae02d: sub    $0xc,%esp
```

When we overwrite the first 5 bytes with the jump-instruction to `hooked_read`, this affects the first two instructions of the original `read` system call (`push %esi` and `mov $0xfffffff7,%esi`). These two instructions require 7 bytes of memory. Thus, if we want to backup the first 2 instructions (not only the first 5 bytes), we can do something like follows:

```
    _memcpy( trampoline, sys_call_table[SYSCALL_NR], 7, 0);
```

This copies the first 2 instructions into an auxiliary byte array `trampoline`. That is, we have now the following situation:

- When the read system call gets called, we are redirected to `hooked_read`.
- `trampoline` stores the first 2 *original* instructions of the *original* system call.

As said above, `hooked_read` somehow needs to call the original system call so that we get the result (and – of course – the user doesn't experience any difference when typing in something). However, the system call is currently damaged since we overwrote the first 2 instructions in memory.

But we can do the following: Since we have the first 2 *original* instructions stored in `trampoline`, we can just "go there", do these two instructions, and afterwards jump to the third instruction of the original system call and continue there. This way, we "simulate" the original `sys_read` function.

We sum up how hooking the `sys_read` function by trampolining works:

- When the original `sys_read` is invoked, we issue a jump to `hooked_read` (which needs – of course – the same signature as the original `sys_read`.

[8] It doesn't actually check return value only, but the (potentially modified) arguments.

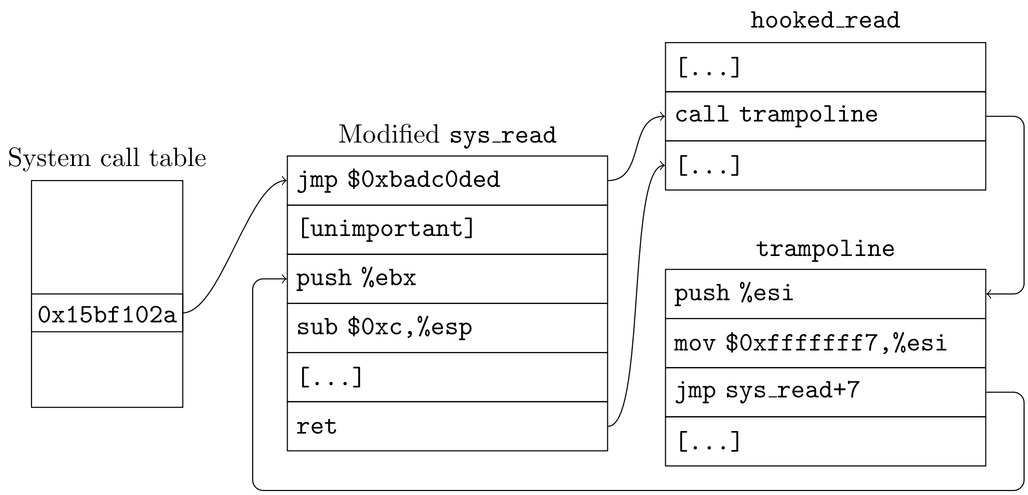


Figure 5: Mechanism of `hooked_socketcall` in connection with `trampoline`. The modified `sys_read` ensures that we jump to the rootkit's `hooked_read` function, which – for its part – issues a call to trampoline. The first two instructions of `trampoline` are equivalent to those in the original `sys_read` function and are followed by a jump to the third instruction of `sys_read`. Finally, the `ret` of the original `sys_read` makes us return to our `hooked_read`, where we continue with processing the return values.

- In `hooked_read`, we cast `trampoline` to the same function type as the original `sys_read`, and call it. Since `trampoline` stores the first 2 original instructions of `sys_read` and jumps afterwards to the third original instruction of the original `sys_read`, the result is the same as if we simply called the original `sys_read` function.
- So, in `hooked_read`, we get the return values of the original `sys_read` function, and can process them as needed.

This technique is illustrated in Figure 5. The most important code snippets of `hooked_read` can be seen here:

```
asmlinkage ssize_t hooked_read(unsigned int fd, char __user *buf, size_t count){
    // declarations, some sanity checks

    // now we treat trampoline like a function and call it
    // note that it has now the same "signature" as the original sys_read
    retval =
      ((asmlinkage ssize_t (*)(unsigned int,
                               char __user*,
                               size_t))trampoline)(fd, buf, count);
    // appropriate processing of values
    return retval;
}
```

Note that we actually issue a *call* to `trampoline`, even it is – per se – not declared as a function! But since it contains the first two instructions of the original `sys_read` and a jump the third instruction of the original `sys_read`, we can just call it and simulate the original `sys_read` function.

So, in this way, we are able to hook `sys_read` without changing the function pointer in the syscall table. Note that this technique heavily depends on the architecture (we implemented it for x86) and probably needs to be adjusted properly when porting it to another architecture.

5.2. Hiding sockets from ss another way

As mentioned before, we can hide sockets from `ss` using another technique. We still hook the `socketcall` function, but we implement another behaviour. In order to understand how this works, let us shortly examine how `ss` actually produces its output. We can track down the code of `ss` to the following essentials (irrelevant things ommited):

```
status = recvmsg(fd, &msg, 0); // receive information
[...]                          // buf is somewhere "inside" msg
h = (struct nlmsghdr*)buf;     // buf holds the information for ss
while (NLMSG_OK(h, status)) {  // "parse" information and output
  tcp_show_sock(h, NULL);      // generate one output line
  h = NLMSG_NEXT(h, status);   // fetch next item
}
```

We see that obviously `recvmsg` is called, and the resulting information is used to generate the output. `buf` is a buffer that contains several `nlmsghdr`s in sequence. These are processed one after another to generate the output.

So, if we are able to modify this sequence of `nlmsghdr`s that is generated by `recvmsg`, we are basically done. As we know, the `recvmsg` system call is internally handled by the `socketcall` function. We hook this function (by modifying the syscall table) and inject in this way our `hooked_socketcall`.

`hooked_socketcall` works as follows: First, we issue a call to the original `socketcall` function so that we get the *original* result. After that, we are going to modify this result appropriately. Since `recvmsg` returns the number of bytes stored in the sequence, we also have to remember and adjust this number. It is stored in `retval`.

We are going to modify the result using a similar technique to that described in subsection 3.3: If we find an entry to be hidden, we take all remaining entries and move them one position forward (see Figure 6 for a visualization, but remember that we are here talking about `nlmsghdr`s). Therefore, we iterate over all the entries and check for each entry if it shall be hidden.

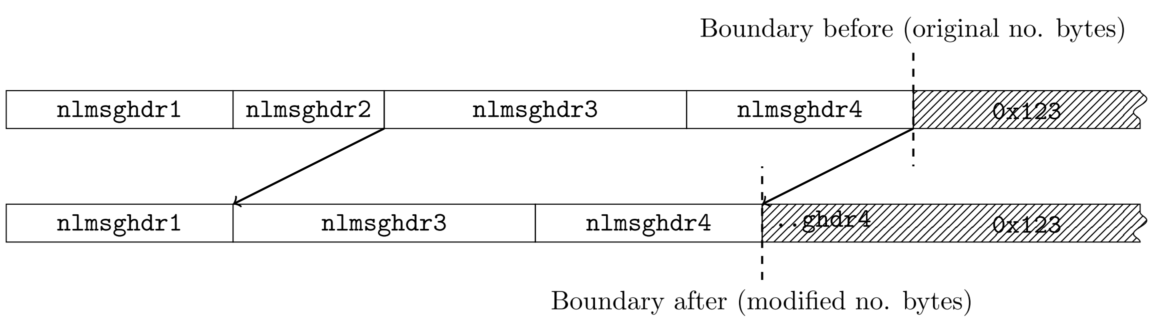


Figure 6: The second `nlmsghdr` is hidden by copying the remaining buffer forward. Note that the number of bytes returned are adjusted (indicated by boundaries).

Note that we have to take care of how long the current `nlmsghdr` is. This length (in bytes) can be computed using `NLMSG_ALIGN((h)->nlmsg_len)`, where `h` is the current `nlmsghdr`. So, the following code is basically used for modifying the buffer:

```
currhdr = (char*)h; // current entry
if (checkport(h)){  // shall we hide this entry?
    // overwrite the current entry by shifting
    for (i=0; i<status; ++i){
        currhdr[i] = currhdr[i + NLMSG_ALIGN((h)->nlmsg_len)];
    }
    // adjust return value (no. bytes)
```

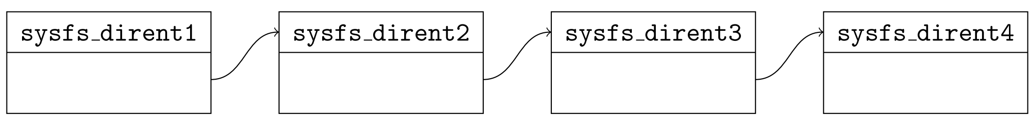
```
    retval = retval - NLMSG_ALIGN((h)->nlmsg_len);
}
```

With this new technique, the rootkit detector is not able to detect if there are sockets running, since it only checks whether `socket(AF_NETLINK, SOCK_RAW, NETLINK_INET_DIAG)` returns `-1`.

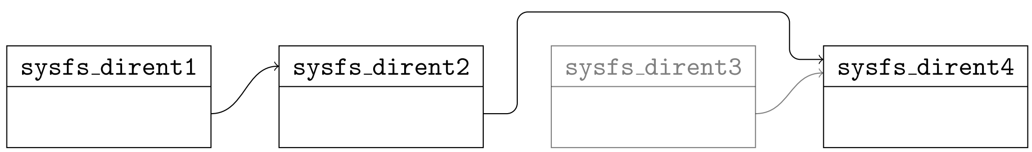
5.3. Removing module from sysfs-datastructures

This part of the detection evasion is probably the most straightforward one. As the detector iterates over the `sysfs_dirent`s and counts them, the obvious thing to do is to remove (and when the module should be added again: re-add) the respective `sysfs_dirent` from this siblings-list. What makes our live even easier is the fact that there are functions in the `sysfs`-implementation that do exaclty this: `sysfs_unlink_sibling` and `sysfs_link_sibling`. As these functions are `static`, we have to copy them (and the necessary `struct`-definitions) to our module. After that, when hiding the module, we just use the `sysfs_unlink_sibling` function to remove it from the list (see figure 7).

It is a remarkable fact that removing the module from all these datastructures does not affect the functionality of the rootkit.



(a) Original list of `sysfs_dirents`



(b) Modified list of `sysfs_dirents`. `sysfs_dirent3` is now hidden, since – even if it still contains a pointer to the next `sysfs_dirent` – it is not reachable when traversing the list.

Figure 7: Removing a `sysfs_dirent` from the list

One thing to note: While removing is done just by adapting the pointers in the list, adding them is a little bit more tricky, as the `sysfs_dirent`s are stored ordered by inode. If you just insert the `sysfs_dirent` at the beginning of the list, this will in most cases lead to unintentional behaviour which may prevent to reload the module.

When the detector now iterates over the internal datastructures to count the modules and our module is loaded and hidden, it won't see any difference compared to the output by `ls` and therefore cannot detect our module.

6. Conclusion

As you've seen, there's already quite bit the rootkit can do. However, as always with such projects, one could add more and more features. For example, one might want to aggravate the rootkit detection by "going one level deeper", e.g. hide files in the vfs-layer of the kernel.

On the other hand, also the detector programmers could do the same things to enhance their detectors.

One last remark: This module has been developed for educational purposes only. Do *not* use it to hack CIA or other similar institutions.

A. How *our* detector worked

For completeness, we will shortly describe how our detector (tool to detect the rootkit of gruppe 1) worked.

`read` **system call** Like our own rootkit, the gruppe 1 rootkit hooked the `read` system call by simply exchanging the function pointer such that it points to another function. We recognized that this hooked function differed in the first three bytes from the original `sys_read`. Thus, we were able to detect a modified `read` system call by comparing the first three bytes.

Moreover, we added a *heuristic* (i.e. this is not a one hundret percent sure indicator) that is based on the actual address for the `read` system call in the system call table. If it was modified, it is probable that it differs much from the surrounding addresses. As said, however, this is just a heuristic.

Hidden processes Gruppe 1 rootkit hid processes by simply setting their process ID to 0. Thus, it was quite easy to scan through all processes and look for an item whose PID is 0. If we find such a process, it is probable that the gruppe 1 rootkit is active.

Hidden TCP sockets In order to detect hidden TCP sockets, we use some little trick: Our detector hooks the (potentially already hooked) `socketcall` system call, and does the same checks that were performed by the gruppe 1 rootkit. The gruppe 1 rootkit basically only alters the output of this `socketcall` for one very specific combination of parameters. We check if these particular parameters are the case, and if so, we alter them *slightly*.

If the `socketcall` is hooked, this slight modification of the parameters causes the rootkit to leave the parameters as they are, which – on the other hand – results in the fact that, basically, `ss` can display the sockets the gruppe 1 rootkit originally wanted to hide.

B. Finding the system call table without System.map

As mentioned in subsection 3.1, the System.map file helps us to locate the system call table. However, it may well be the case that this file is not present or accessible to us. Therefore, we also implemented a routine that is capable of finding the system call table on itself.

Therefore, it uses a quite straightforward approach. We simply iterate over the memory (starting at `&lock_kernel`, until `&loops_per_jiffy`. We check in each step, whether for the current position `h` the following holds:

```
h[__NR_close] == sys_close
```

If the above holds, it is highly probable that `h` is the location of the system call table. It *might* be by incident that this condition holds for other memory locations in the specified section as well. However, we got quite good results, so we didn't bother checking more conditions, which might of course be done.