Praktikum: Rootkit-Programming
Final Writeup

Daniel Strittmatter, Christoph Baur (Group 4)

5. Februar 2013

Introducing `bROOTus`

First of all, thank you for showing your interest in the educational **Linux Kernel Rootkit** `bROOTus`.

`bROOTus` is a Linux kernel rootkit that comes as a single LKM (Loadable Kernel Module) and it is totally restricted to the Linux Kernel 2.6.32. The rootkit is dedicated to educational purposes and is intended to point out some mechanisms on how to manipulate data structures and hook functions in the kernel in order to achieve certain tasks such as file hiding, module hiding, process hiding, socket hiding, packet hiding, keylogging and privilege escalation from within the kernel. We do not provide any information on how to get the rootkit into the kernel when not being the root user already. This means this is no guide on how to find or even use exploits from user space in order to inject the module into the kernel. We assume that you already have superuser privileges and are capable of loading the module.

Setup and Usage

Setup and usage of our rootkit `bROOTus` is pretty easy. We provide some information on building the rootkit from source and on controlling the rootkit via a covert communication channel, as well as the requirements and dependencies for building and running the rootkit. This is actually all you need to know in order to get it running.

Requirements and Dependencies

In order to build and run the rootkit, please make sure your system fulfills the following requirements. You need:

- Linux Kernel 2.6.32 (vanilla, without any patches applied)
- Linux Kernel Headers (linked to by `/lib/modules/2.6.32/build`)
- Support for LKMs (Loadable Kernel Modules)
- A sane build environment, including
  - GCC version 4.4.5
  - GNU make version 3.81

Building

In order to build the rootkit, just `cd` into the root directory of the rootkit source folder and run `make`. The module should be compiled and linked without any warnings or errors.

Loading

In order to load the module, first make sure you properly compiled the rootkit. Then simply run `insmod rootkit.ko` to load the module into the kernel. Once loaded, `bROOTus` does some basic hiding by default:

- Hiding *itself* (i.e. the module) from the user
- Hiding each file or directory whose name starts with `rootkit_` from the user
- Launching the Keylogger mechanism and the mechanism that sends the logged data via UDP
- Hiding packets with the specified target or source IP address (in case it has been specified)

How the various hiding mechanisms can be enabled, disabled and configured is described in the following section.

Controlling

The behaviour of `bROOTus` can be controlled at *runtime* (i.e. after the module was loaded) through a covert communication channel. This channel is created by intercepting any read system call on `stdin`. Thus, anything you type on any virtual terminal will be intercepted and interpreted by the rootkit. The rootkit itself is capable of filtering the data read from `stdin` for certain keywords that serve as commands. Those commands all follow the same syntax rule: `command(arg1,arg2,...,argn)`. Hitting `enter` or `return` is not necessary as the command will be executed as soon as the closing parenthesis was typed in.

Commands

The currently available commands can be divided into several command groups and will be introduced hereafter.

File hiding	
`files_on()`	Enable file hiding
`files_off()`	Disable file hiding
`prefix(pfx)`	Set the prefix of the file names to be hidden to `pfx`
Process hiding	
`processes_on()`	Enable process hiding
`processes_off()`	Disable process hiding
`pids(pid1,pid2,...,pidn)`	Hide all processes with the given Process IDs
Socket hiding	
`sockets_on()`	Enable socket hiding
`sockets_off()`	Disable socket hiding
`ports(prt1,prt2,...,prtn)`	Hide all sockets which are connected or listening to one of the given ports. The syntax for the arguments is `<type><port no.>` where `type` is `T` for TCP, `U` for UDP or `A` for both (the lower case equivilants are also supported). For example `ports(t22,u111,a80)` hides TCP port 22, UDP port 111 as well as TCP and UDP port 80.
Module hiding	
`modules_on()`	Enable module hiding
`modules_off()`	Disable module hiding; unhides all prior hidden modules
`hidemod(mod)`	Hides the module named `mod`
`showmod(mod)`	Unhides the module named `mod`
`mod_hide()`	Hide the *rootkit* module
`mod_unhide()`	Unhide the *rootkit* module
Privilege Escalation	
`rootme()`	Gives the current shell superuser privileges
Keylogging	
`keylogger_on()`	Enable key logging
`keylogger_off()`	Disable key logging (also disables sending the logs via UDP)
`syslog_ip(your_ip)`	The IP address of the server you want to send the logged data to
`syslog_port(your_port)`	The port you want to send the logged data to
Packet Hiding	
`blocked_host(your_ip)`	Hide packets that are related to this IP address (source or target)

Limitations

The current covert channel implementation comes with backspace support. However, it does not provide a WYSIWYG behaviour which means we cannot guarantee that the rootkit "sees" the command the way the user sees it on his/her terminal. This applies especially if one uses backspaces *after* a command was executed (i.e. after a closing parenthesis was typed) in order to correct the argument, for example. In this case you have to enter the command again.

There are also (very rare) cases when a command is not recognized because the input buffer went full while the command was halfway typed in. Also, the argument parsing exits when the first closing parenthesis is detected. There is currently no possibility to escape characters. We also don't support a command history, yet.

Unloading

Unloading works by typing `rmmod rootkit` into a shell. Hit enter to execute that command and unload the module.

Note: Make sure the rootkit is not hidden when trying to unload. If it is hidden, you first have to unhide the module via the covert communication channel (e.g. by using `mod_unhide()`). Otherwise rmmod will tell you that there is no such module called "rootkit".

Implementation

Now let's dive into the technical aspects of `bROOTus`. One might ask him or herself, how the rootkit itself completes the tasks of file hiding, module hiding, socket hiding, process hiding, privilege escalation, keylogging, packet hiding and the setup of a covert communication channel.

Before we go into great detail, the following sections cover some basic technologies which are required for any rootkit implementation.

Basic technologies

Non-exported symbols

Not all variables and functions declared in the kernel are available/easily accessable from a kernel module. A prominent example is the address of the system call table within the kernel. Manipulating this data structure is essential for virtually every Linux rootkit but the corresponding variable `sys_call_table` is not resolvable in the context of a module since kernel version 2.6.

A fairly easy approach to circumvent this problem is using the `System.map` file which is generated in the kernel build process and usually put in the `/boot` directory. It contains the kernel symbols and the corresponding addresses. We used this file to generate a C header file from it which gives us access to internal kernel functions and variables.

However, this approach limits the module's scope to exactly one kernel version *and* configuration. This constitutes a very strong limitation and renders `bROOTus` unusable in almost any other environment than our testing VM. To maintain at least a little bit of compatibility and portability, we obtained the address of the previously mentioned system call table in another way.

Obtaining the system call table without the System.map

Our approach to determine the address of the system call table requires to dig deeper, on an even lower level. Our VM's kernel uses the **Intel x86 SYSENTER/SYSEXIT** instructions to perform system calls. The address to which the CPU jumps after this instruction is set in the **model-specific registers (MSRs)**. The kernel sets this value in `arch/x86/vdso/vdso32-setup.c`:

```
wrmsr(MSR_IA32_SYSENTER_EIP, (unsigned long) ia32_sysenter_target, 0);
```

The corresponding function `ia32_sysenter_target` is defined in the assembler file `arch/x86/kernel/entry_32.S`:

```
ENTRY(ia32_sysenter_target)
  CFI_STARTPROC simple
  [...]
sysenter_do_call:
  cmpl $(nr_syscalls), %eax
  jae syscall_badsys
  call *sys_call_table(,%eax,4)
  movl %eax,PT_EAX(%esp)
  [...]
```

The CALL instruction invokes the desired system call using the address of the system call table and the offset in register EAX (the syscall number) multiplied by 4. As the register content of EAX is unknown at compile time, the address of the system call table has to be somehow included in the instruction.

Let's have a look at the generated object file entry_32.o:

```
# objdump -DS entry_32.o

000000c9 <sysenter_do_call>:
  c9:   3d 51 01 00 00          cmp    $0x151,%eax
  ce:   0f 83 2f 02 00 00       jae    303 <syscall_badsys>
  d4:   ff 14 85 00 00 00 00    call   *0x0(,%eax,4)
  db:   89 44 24 18             mov    %eax,0x18(%esp)
```

We see the byte sequence `0xff 0x14 0x85` is used in the CALL instruction we are interested in. The remaining 4 bytes correspond to the address of the syscall table which the linker hasn't filled in, yet. The instruction decodes to the following:

- The first byte (`0xff`) is one of the opcodes of the `CALL` instruction
- The second byte (`0x14`) sets the `mod-reg-r/m` field to use scaled index byte addressing mode and makes EDX the destination register.
- The third byte (`0x85`) defines the scaled index byte layout:
  - Base: Displacement-only
  - Index: EAX
  - Scaling: 4
- The next 4 bytes form the displacement (i.e. the address of the syscall table).

With these information at hand we can search the code of the `ia32_sysenter_target` function for the byte sequence `0xff 0x14 0x85` because our desired instruction is the only one of this form there. The following 4 byte give us the address of the syscall table. We can obtain the address of the ia32_sysenter_target function with the `rdmsrl` macro:

```
rdmsrl(MSR_IA32_SYSENTER_EIP, syscall_handler_addr);
```

The whole mechanism is packed inside a function `get_syscall_table_addr` which is called on initialization of `bROOTus`. If the address can't be found the module refuses loading. Unfortunately, this technique has some limitations as well: This should work as long as the first `CALL` instruction of the specific type (byte sequence `0xff 0x14 0x85`) is the call to an entry in the syscall table and the `SYSENTER` instruction is used to invoke system calls. Thus this limits the scope of this method to the x86 processor familiy. Since we didn't test it with a 64-bit architecture it might be limited to 32-bit processors, too.

Alternatively, one could resolve the address of the system call table via the deprecated **Interrupt Descriptor Table**: Traditionally, when a system call is to be invoked, several registers are filled with parameters and the `int 0x80` instruction is invoked to pass control over to the kernel. The interrupt descriptor table (IDT) contains up to 256 entries where each entry has a size of 8 byte containing some flags and the address of the interrupt handler. At cell 0x80 the IDT contains a pointer to the handler `system_call`. This method may be more or less deprecated, but is still implemented in the Linux kernel, probably for compatibility reasons. So what one has to do is at first try to determine the address of the IDT which is stored in the **IDTR register**. Then you can examine cell 0x80 of the IDT and extract the address of `system_call`. This handler jumps to syscall_trace_entry at some point, which means you can determine the address of syscall_trace_entry from inside the system_call function just like we described above.

Now that we've successfully obtained the address of the system call table, we have to introduce a very important mechanism based on the system call table: system call hooking.

System Call Hooking

A very crucial mechanism for rootkits is the hooking of system calls. The basic idea is to rewrite one or more function pointers in the system call table to point at our own function(s) while keeping a copy of the original value. We need the old pointer to restore the system call table when our module is unloaded as well as for calling the original system call function. In most cases we do not want to rewrite the whole functionality of a system call but either manipulate the arguments on the way to the original function or modify a result buffer or return value after invoking the original function.

We have already resolved the initial address of the system call table as described in the previous chapter. But there is still another obstacle we have to overcome: the memory page containing the table is marked write-protected in the page tables. Fortunately, our code is executed in kernel mode which means that we have unlimited access to the CPU (and memory) and are hence able to circumvent these restrictions. One possibility is to modify the CPU's control register `cr0` by setting the 16th bit to zero which disables the write-protected mode. This approach is limited to Intel x86 processors, of course. So for compatibility reasons we chose another way.

Our method temporarily sets the page containing the system call table writable by manipulating the corresponding entry in the page tables. We achieve this by using the internal kernel function `lookup_address` which returns the page tabe entry which controls the page of the given address in form of a `pte_t`. Now we can change the permission flags to our needs while keeping the original ones. After we finished modifying the system call table the previous permissions are restored. `bROOTus` provides this functionality through the methods `syscall_table_modify_begin` and `syscall_table_modify_end` which are defined in `syscall.c`.

Now, that we know where the system call table is located and we can write to it, we can save and replace the pointers we want to. This task is addressed by the macros `HOOK_SYSCALL` and `RESTORE_SYSCALL` which are defined in `syscall.h`.

Module File Structure

Before we dive into the implementation details of the various hiding mechanisms, we will briefly describe the file structure of our kernel module. We split the functionality up into several header (`.h`) and implementation (`.c`) files. This way the subsystems (file hiding, socket hiding etc.) are compiled seperately and linked together at the end of the building process. Adjusting the `Makefile` is pretty straight-forward.

The following table describes each component of `bROOTus`.

`brootus.h`	Is included by every *component* (file hiding, socket hiding etc.) and defines the `BROOTUS_MODULE` macro which creates mandatory function prototypes
`kernel_functions.{h,c}`	Define commonly used non-exported kernel functions
`kernel_variables.{h,c}`	Define commonly used non-exported kernel variables
`load_magic.{h,c}`	Define a mechansism to determine if the module is still loaded
`sysmap.h`	Contains the kernel symbols and addresses; generated using the system map
`syscall.{h,c}`	Implement functions and macros needed for system call hooking
`vt_channel.{h,c}`	Implement the covert channel mechansism
`file_hiding.{h,c}`	Implement the file hiding *component*
`module_hiding.{h,c}`	Implement the module hiding *component*
`process_hiding.{h,c}`	Implement the process hiding *component*
`socket_hiding.{h,c}`	Implement the socket hiding *component*
`keylogger.{h,c}`	Implement the keylogging and especially the log-sending *component*
`packet_hiding.{h,c}`	Implement the packet hiding *component*
`rootshell.{h,c}`	Implement the privilege escalation
`main.c`	The main module file; contains init and cleanup functions

Every *component* `cmpname` has to provide the functions

- `init_cmpname`,
- `finalize_cmpname`,
- `enable_cmpname` and
- `disable_cmpname`

whose prototypes are generated by the `BROOTUS_MODULE` macro.

File Hiding

To get some first impressions how tools like `ls` obtain all the files in a directory, we made an `strace` on `ls`. This way, we already revealed the magic behind fetching the entries of a directory: The shipped ls command uses the system call `getdents64` (earlier version relies on `getdents`) to get directory entries. Thus, we hooked both system calls `getdents64` and `getdents` by replacing the original function pointers in the system call table by our own manipulated functions.

The original function (which we call at the very beginning of our hooked call) fills a buffer (pointed to by the `dirp` variable) with `linux_dirent{,64}` structs which represent directory entries. We can now iterate through these structs by using the `rec_len` field. Once we see a file that should be hidden (by comparing the `d_name` field to the predefined prefix) we copy the following structs over the current one and reduce the buffer length returned to the caller.

As a result we have an intact array of structs but with some entries removed. The files aren't listed by commands like `ls`, `find` or `tree`. However they are still accessable (readable/writeable) because `open` and `stat` are not affected by this method of file hiding.

Process Hiding

Most of the linux programs that somehow want to get information about processes and operating system statistics make use of the proc-filesystem. The proc-filesystem is a special interface for all these information.

And so it is with processes: the proc-filesystem contains entries for all processes, identified by their respective PID. Hiding (or even removing) the entries of the respective PIDs might result in hiding the existance of certain processes. One approach to hiding processes is to hook the proc_fops (file operations) and filter for PIDs. PIDs that match the provided ones are not put into the directory. That's exactly what we did in this kernel module.

We replaced the proc file operation `readdir` by our own file operation `bROOTus_readdir`. The original `readdir` calls a function called `filldir` which is responsible for filling a proc directory with entries. In our `bROOTus_readdir` we used our own filldir-function which first filters for the provided PIDs and returns 0 on match. Otherwise the original filldir-function is being executed.

Why didn't we simply hook the `filldir` function? Well, unfortunately neither the proc file operations nor the filldir-function are exported. Thus we needed to find another point of entry into the proc filesystem in order to get to those function pointers. We found the proc directory entry `proc_root` to be exported. `proc_root` itself points to the struct `proc_fops` again, which points to several functions, such as `readdir`. This way, we had anything we needed in order to hook the `readdir` function and hide certain PIDs by filtering the PIDs from within our custom `filldir` function. Voilà.

Module Hiding

The kernel provides detailed information about modules that reside in the kernel by two different interfaces: via `/proc/modules` and via `/sys/module`. Thus the main target is to hide a module from appearing in both interfaces.

All modules that reside in the kernel are organized in lists of `struct module`. Module structures provide almost everything we need to hide a module. In order to remove a module from showing up in `/proc/modules` we simply need to remove our module from the list of available modules (this list is used to create the content of `modules`). This can be easily done via predefined functions for operations on lists, such as `list_del` and `list_add` or `list_add_tail`. Now that we removed the module from that list, processes like `lsmod` do not list our module anymore.
However, there is another task: hiding a module from `/sys/module`. Once again, the module structure comes in handy: the module structure contains another structure that holds the so called `kobject` entry, which is actually an entry in `/sys/module`. The kernel provides some functions to remove from and add kobjects to the sys-tree. So we simply backup the original kobject of our module and remove the kobject from `/sys/module` with help of the function `kobject_del`. Now the module does not show up in `/sys/module` anymore.

Unfortunately, one cannot unload the module anymore once it is hidden. So we have to provide a mechanism to restore (*unhide*) the module again. This works simply by adding the module structure to the list of modules again, and by adding the kobject we've backuped before to the sys-tree again with help of `kobject_add`.

Socket Hiding

This is probably one of the more advanced parts of the rootkit: Hiding the existance of sockets and their ports from the user. The first task obviously is to find out, how a user can determine which ports are used by which application on which socket. We found the `proc` filesystem to be one important target again, since it contains *virtual* (so called `sequence`) files with plenty of information about sockets and ports the kernel currently knows. And then there are actually two tools that display information about sockets: `netstat` and `ss`. An `strace` on netstat shows that it simply uses the contents of the files in `/proc/tcp` and `/proc/udp`, whereas `ss` does something totally different and magic at a first glance.

So `netstat` essentially uses `/proc/tcp` and `/proc/udp` to get its socket information. One might think that hiding certain socket information is as easy as just hooking the `read file operation` of the specific files: It is not. `/proc/tcp` and `/proc/udp` are so called `sequence files` and cannot be controlled with the file operations of the proc virtual filesystem. To be more precise, those files get sequentially filled on request by the corresponding `seq functions` of `tcp` and `udp`. And of course one first has to find an access to those functions. This is rather straight forward since there is a `proc_dir_entry proc_net` from which you can search for the proc dir entries of tcp and udp. Once you found them, you can retrieve their file operations, make a backup of the `tcp_seq_show` and the `udp_seq_show` function and replace their pointers by the ones of your custom functions.

So in order to manipulate their output we first hooked the kernel functions `tcp4_seq_show` and `udp4_seq_show` by replacing the `show` function pointer in the corresponding `seq_operations` structs by our own custom functions. To get a pointer to these structs, we iterate through all net namespaces (usually there's only one) to get the `proc_dir_entry for /proc/<pid>/net`. From here we just search the entries called `tcp` and `udp` and fetch their `tcp_seq_afinfo/udp_seq_afinfo` structs through the data member. We have access to the `seq_operations` now.

The hooked functions emulate the original ones but return a length of zero if the given socket uses a hidden port as source or destination port.

We took care of netstat so far, but there is one tool left: `ss`. Some further investigation on how `ss` obtains its socket information lead us to the proc filesystem again for UDP sockets: Here `ss` also uses `/proc/udp` which we already handled. For TCP sockets, it leads us somewhere totally different: `ss` uses the kernel modules `inet_diag` and `tcp_diag` to gather socket information. So we could try to work on those modules, but it is kind of a hard work since we cannot access any of the symbols that would be useful. However, the communication with the module is done using a `netlink` socket which is created when the `inet_diag` module is loaded. In order to intercept and manipulate the transfered data one could hook the `recvmsg` system call which is called by `ss` to read from that socket. That is exactly what we did. However, on the 32-bit version of Linux 2.6.32 this function does not have its own pointer in the syscall table. Instead, there is the `socketcall` system call that multiplexes several syscalls dealing with sockets. So we hooked this one and called our version of `recvmsg` if the supplied call ID has the corresponding value.

The hooked `recvmsg` function calls the original one at the very beginning which fills the given `msghdr` struct. Then we check if the given file descriptor points to a socket which uses the netlink address family and the `inet_diag` protocol. If so, we iterate through the `inet_diag_msg` structs (each prepended by a nlmsghdr) and compare the source and destination ports with our list of hidden ones. In order to hide an entry we copy the remaining entries over it and adjust the data length which is returned to the user.

Privilege Escalation

Once you are inside the kernel and own all the rights you can possibly have, it should be no more problem to escalate privileges on a shell. One idea to escalate privileges in an open shell is to hook a system call like `execve` and change the credentials of the process that is to be launched. Each process owns a respective `task_struct` which itself holds a struct called `cred`. This struct holds the credentials of the respective process, such as the user id. Once a process runs as root, every following process will run as root as well, absolutely anything you do is done as root (as long as you don't change the user). This means the shell inherits the user of the last process. Changing the user IDs in the `struct cred` of a process to zero does the trick. This works like a charm when doing it from a hooked sytem call. But it can be done easier: The linux kernel offers the so called `current macro`, which is a pointer to the `task_struct` of the currently running process. This way, you can change the credentials of the current process as well without having to hook a system call. And that is good because changes in the system call table can be easily detected.

Covert Communication Channel

The covert communication channel is established by hooking the `read` system call and filtering for certain inputs on `stdin`. We also have to address *concurrency* problems which occur if there are multiple active shells and even other processes writing to `stdin`[1]. So we decided to create buffers for each virtual terminal we receive input from.

Whenever our hooked `read` system call is invoked, we issue a call to the original one with unchanged parameters. After this call returns we can access the read data through a buffer the user provided and which got filled up by the original `read`. We then test if the given file descriptor is zero (which is `stdin` under Linux) and look up

[1]for example when using an anonymous pipe

the name of the file to which the `stdin` file descriptor is connected. We abort our interception if the file name is empty or a null pointer because the input does not come from a virtual terminal in this case. If the file name exists we search the buffer list for an already allocated buffer for this name and create a new one if we don't find anything. Afterwards we append the read data to the buffer and try to find one of the command keywords (which we keep in an array). If we find such a keyword we try to parse the arguments to it. If the command is complete (i.e. we find a closing parenthesis) the corresponding function is called with the parsed argument in form of a character pointer.

The covert channel subsystem provides an interface to add commands through the function `add_command` which takes the keyword and the callback function as parameters. The callback function has to have a `void` return type and must take a `char*` as parameter.

Keylogging via UDP

By means of the covert communication channel, a keylogging mechanism is already implemented. Now we enhanced the already existing keylogger by the feature of sending the logged data to another device (of your choice) in the network. We just had to take care of creating a socket, preparing the logged data for sending and finally send this data to its target via UDP. The kernel provides any of the mechanisms needed for these operations in header files such as `include/ip.h`, `include/in.h` and `include/inet.h`:

We first defined one variable of type `struct socket` and one variable of type `sockaddr_in`. On initialization of the rootkit both structs get allocated, filled, and a socket is created. The target IP address and target port you specified (or you are going to specify later on via the previously listed commands) are put into the respective `struct sockaddr_in`. Afterwards, we connect to that socket using the previously filled `struct sockaddr_in`.

Whenever a read system call from a virtual terminal is made, we forward the keys to the network device you specified by calling the function `log_keys` of the keylogger module. This function wraps the keypresses into the syslog format and finally sends the information as an UDP packet via our socket through the function `sock_sendmsg`.

But note: There is a limit to virtual addresses that is different for kernel and user space, which means when invoking system calls from the kernel, one should set this highest valid virtual address accordingly. We do so by surrounding the `sock_sendmsg` call with

```
oldfs = get_fs();
set_fs(KERNEL_DS);
...do the call...
set_fs(oldfs);
```

Although we tried to be compliant with the syslog format specified by IETF RFC 5424, syslog-ng servers do not seem to recognize the (whole) format. Our message format should look like:

```
[ VT-ID ] logged keys
```

The program name is set to `bROOTus` while the other header fields are left empty. The facility is set to "user-level messages" while the severity is set to "warning" which leads to 12 as the numerical representation.

Packet Hiding

Our packet hiding mechanism is based on the way `tcpdump` and `libpcap` obtain packet information. Thus, our first step was to take a look at tcpdump: by stracing tcpdump we found out it uses "packet" sockets (of type `PF_PACKET`) to retrieve traffic information:

```
socket(PF_PACKET, SOCK_RAW, 768) = 3
```

This socket is then polled every second:

```
poll([{fd=3, events=POLLIN}], 1, 1000)  = 0 (Timeout)
```

If there is data available, tcpdump writes them to STDOUT without performing any system call (for example some sort of receive) between the poll and the write. A glance at the libpcap source code revealed that it prefers to "memory map" the socket data. This means we cannot just hook the socket receive function to manipulate/hide the packets. Some further investigations lead to the assumption that the function tpacket_rcv is involved in the process of giving packet information to user space when "memory mapped" packet sockets are used. A pointer to this function (or one of its relatives) is stored in the packet_sock struct. So we could hook this function by overwriting this pointer. This has some disadvantages, though:

- It is difficult to undo this change when our module is unloaded (it might easliy lead to a kernel panic)
- We have to hook other functions (e.g. socket creation) or use a "watchdog" in order to do the hooking
- Some functions overwrite this pointer, which could lead to the same problem as above

So we decided to hook the tpacket_rcv function by *jump code injection*. This means that we copy (assembly) code at the top of the function body which leads to a jump to our (hooked) function. Whenever we need to call the original function, we have to overwrite this section with the original code again and do an ordinary call to it. So we have to be careful there and introduce a lock to prevent interleaving hooking and restoring code. This meachanism also has some disadvantages:

- It limits the scope of this rootkit to x86 processors
- It involves locking
- It requires information from the system map or heuristic strategies

Since the x86 instruction set does not provide a jump to an (absolute) immediate value, we have to come up with another solution. One possibility is the indirect jump to an address stored in a register. But this involves "destroying" a register where function arguments might be stored. So we went with the "push-ret" trick: we push the (absolute) jump address onto the stack and execute a ret instruction afterwards. The binary assembly code is given by:

```
# echo -e 'push $0x12345678\nret' | gcc -o /tmp/asm.o -xassembler -c - && \
  objdump -DS /tmp/asm.o
0:   68 78 56 34 12          push   $0x12345678
5:   c3                      ret
```

So we just have to plug in the jump address after the first byte and copy the resulting 6-byte code to the function's top.

The function hide_packet takes a socket buffer (filled with a network packet) and checks whether this packet should be hidden from the user. For this purpose we extract the IP header and check if the packet contains the blocked host's IP address in the sender or the receiver field. In order to hide our syslog traffic, we also check the UDP header (if any) for the configured port as well as the destination IP for the configured syslog host. We return a non-zero value if the packet should be hidden and zero otherwise.

The rest of the implementation deals with the hooking mechanism described above. We hook the ttpacket_rcv, packet_rcv and packet_rcv_spkt functions to also cover non-"memory mapped" sockets. These return zero if the packet should not be reported to the user and call the original function otherwise.

Final thoughts

bROOTus has got plenty of features with the purpose of gaining (almost) full control over a linux system, and it probably is an excellent source for getting a glance at many of the mechanisms used in linux kernel rootkits:

We implemented mechanisms to control the network traffic (hide certain sockets and packets), to gain root privileges in a shell, to hide processes and files, and even to hide the module itself. Furthermore there is a keylogger included that reads any keyboard input that it can get, and the rootkit is capable of sending the logged data via UDP to, for example, a syslog server. Some of the features are rather simple to implement, other mechanisms are rather advanced and require techniques such as code injection or fiddling with registers and binary code itself.

Nonetheless there are some things that can be improved and that could be faced in the future: the emphasis of this project definitely was not put on the stealthness of the rootkit, thus there are no advanced techniques to disguise and hide the rootkit (e.g. from stable and well-known anti-rootkit tools such as rkhunter). Also the rootkit is totally limited to the Linux Kernel 2.6.32. But since this rootkit is designed for educational purposes, this should be no concern at all. If you want to enhance the rootkit, feel free to do so.