Lab4: OS Security Lab

The goal of this lab is in two folds. The first is giving you an experience of writing an in-kernel access control module though a super simple one, to let you understand that what really is possible is which way. The second is to let you implement a demonstration of a kernel-level malware so that you can realize the power of an attacker who has the full control of the operation system kernel.

Please note that all the materials are developed by the instructor (Hyungon) and you must not redistribute neither contents not the answers for the fairness of future students though the details will be changing.

0. Setup

In this lab, we need to use our own kernel that we built from the source. It typically is pretty handy task, so please carefully follow the instruction.

> It took about 6min for me to up the VM exluding the time taken to grep the box from Vagrant Cloud.

Copy the lab files to the VM local storage for faster kernel builds.

```
cp -r /vagrant/lkm .
cp -r /vagrant/lsm .
cp -r /vagrant/README.md .
cp -r /vagrant/rootkit .
```

Before getting started, build the Linux kernel for the first time. It could take 10min+. To see what you really are doing, you can check `lsm/Makefile`.

```
cd lsm
time make first-build-kernel
make reboot
```

> It could take 20min+

Your VM will reboot, so open the terminal again with this command.

```
vagrant ssh
```

To check the booting log (if needed), you can uncomment the line with `vb.gui = true`.

If you booted and logged in to the VM again, make sure that you are working with the kernel that you rebuilt by

```
uname -r
```

You should see `3.16.59`, which is the version we built.

Now you are ready, and if you want to rebuild/use the kernel later (mainly for the `2. lsm`),

```
time make rebuild-kernel
```

> It took about 2min with no source modified.

0. lkm

At this step, we will take a look at and write a simple Loadable Kernel Module for Linux. Linux kernel is one of the most complex software among at least the open-sourced ones, and keeps changing. This makes it difficult to create a comprehensive tutorial of doing A with it.

This part will, thus, cover the features and what you can do with the modules that you will need to know to finish the rest of this assignment.

0.1. hello

Let's first build, insert and remove a minimal kernel module.

Enter the subdir `lkm`.

```
cd lkm
```

The `Makefile` there will build the kernel modules for you, which are `*.ko` files. Throughout this assignment you will not need to modify the file. It builds the three `*.c` files that we are using at this stage. Please check the comments in `Makefile` for a bit more details.

`hello.c` is the minimal example of kernel module to start with. Check the comments to learn more details.

After building the module by running

```
make
```

you can insert and remove the module like this.

```
sudo insmod hello.ko
sudo rmmod hello
```

> The former having `.ko` and the latter not having it is not typo.

As the comments in `hello.c` says, you can read what it prints like this.

```
dmesg
```

Here, just build, insert and remove the module provided, and check the log.

What do you see about inserting and removing the module? Why? Google Linux kernel log level.

Deliverable 1) `hello.txt` describing the output and why.

Q2) device

Later we will write a kernel module that we can communicate with, from user application. How can we create such a channel?

An example is to create and register a character device. Though the name device suggests that it is a physical device attached to the machine, but we can in fact create something that behaves like a device.

Read the comments in `device.c` which implements a simple device driver creating a file `/dev/hello_device`.

Build and test the device with these commands.

```
make
sudo insmod device.ko
sudo chown vagrant /dev/hello_device
./test-device
sudo rmmod device
```

Referring to the example, modify the module so that the module keeps at most 10 strings that a user program wrote. You need to add another entry and function (`.write`) to do this. Also try to print the file path used to open the file, which will be `/dev/hello_device`, and check if it really is correct using `strncmp`. Our kernel has its own implementation of `strncmp`, and you can refer to how the file struct look like from `lsm/linux-`

source-3.16/include/linux/fs.h and use d_path function in lsm/linux-source-3.16/fs/dcache.c.

Deliverable 1) device.c modified as instructed above, and contains comments explaining the behavior.

Q3) filp_open

We will later want to get an inode object, which represents a file, from its name, or a pathname (e.g., /home/vagrant/malicious). How can we do that? There are many ways but here let's use a simple way.

You can find filp_open function at fs/open.c under the Linux kernel's source tree. What does the comment say? Let's just use it here as we want to make our life easy.

The function is just like the user-level open, but returns a pointer to struct file. Among the members of file, you can find f_dentry, which again contains what we are looking for, f_inode.

With this, let's take a look at which functions are called when opening/closing/reading/writing a file.

First create a file:

```
echo "asd" > /home/vagrant/temp
```

Then modify the empty module file.c to get the inode for the temp file and print the address of functions (the entries of the table i_fop) for open, release, read, and write.

Check the name of functions by opening the System.map file:

```
$ sudo vi /boot/System.map-3.16.59
```

Deliverable 1) file.c modified as instructed above, and contains comments explaining the behavior.

2. lsm

Linux Security Module (LSM) is a special type of Loadable Kernel Module that adds hooks to the kernel core to implement Mandatory Access Control (MAC). Despite its name, at this moment LSMs cannot be loaded after the system being booted. The Linux kernel community is discussing and working on supports for dynamic loading, but still we should rebuild the entire kernel and reboot to deploy our own LSM.

At this step, we will write a toy LSM that prevents a specific program from accessing our secret file, using the skeleton that we have.

Now let's check the skeleton at `lsm/linux-source-3.16/security/thook`. It implements and already adds a `toy hook` at file opens.

To check more complete version instead, you can refer to the one of `selinux` at `linux-source-3.16/security/selinux`.

As you can learn from the comments in `thook.c`, the module is roughly composed of the two parts. Firstly, the hooks are added at boot time so that later accesses to the corresponding resources will call them. Additionally, a device file is created so that you user program can later interact with the module.

Please note that you can rebuild the kernel with this command at `lsm`.

```
time make kernel-rebuild
```

And to reboot, use

```
make reboot
```

Q1) Count opens

Modify the `thook` module so that it counts how many times the file `/home/vagrant/secret` is opened, and allow a `root` user's program to obtain the number. You can let the device file read like this.

```
/home/vagrant/secret has been opened 10 time.
```

Also write a user-level program that reads and prints the counter values.

You may want use a function called `snprintf`

Deliverable 1) The modified thook.c @ `lsm-thook-count.c`

Deliverable 2) The user program (in C) count.c @ `lsm-count.c`

Deliverable 3) A write-up @ `lsm-report-count.txt`

Q2) Prevent opens and unlinks.

Modify further the module so that `thook` in addition prevents any attempt to open or delete the file `/home/vagrant/secret`, and counts the number of all/failed attempts to delete the file. Allow only a program that has the name bp to either open or delete the file. Also, like @ Q1, allow a user program to access the log like this.

```
Attempts to open: 10
Failed to open: 1
Attempts to unlink(delete): 10
Failed to unlink: 5
```

Deliverable 1) The modified thook.c @ `lsm-thook-prevent.c`

Deliverable 2) The user program (in C) count.c @ `lsm-prevent.c`

Deliverable 3) A write-up @ `lsm-report-prevent.txt`

3. rootkit

Everything running on this system relies on the Linux kernel. How can we alter its behavior at run time? An attacker who can insert an arbitrary kernel module can do quite a lot of things, which is why modern kernels try not to allow.

Let's see what the attackers can do by implementing simple ones.

Q1) syscall

Modify the skeleton module so that the module hijacks all calls to `sys_unlinkat` and prevent a file `/home/vagrant/malicious` from being deleted. You can write your own function that checks the file name and refuse to call the real `sys_unlinkat` and modify the kernel's system call table to achieve the goal. Modern kernels make the system call tables not writable, which introduces another step in this kind of attack. However, you can directly overwrite the system call table in this case, because our kernel is modified not to make it read-only.

Typically you also need a means to locate the system call table, but here just consult the file `/boot/System.map-3.16.59`. You will be able to find `sys_call_table` there. To be sure, check the contents using your module.

Deliverable 1) The modified syscall.c @ `rootkit-syscall.c`

Deliverable 2) A shell script (command) that tests the effect @ `rootkit-syscall.sh`

Deliverable 3) A write-up @ `rootkit-report-syscall.txt`

Q2) VFS

It seems too trivial to make the system call table read-only to prevent the earlier attack. Where else can we corrupt to do something similar? Each file in the Linux kernel is represented as a `inode` object, which has a pointer to a function table called `f_ops`. By replacing the table with yours, you can intervene the file operations without modifying the system call table. As the `inode` objects are created at run time, they cannot become read-only.

Modify the skeleton module so that the module reverts the content of the file `/home/vagrant/malicious`. Like:

```
echo "asd" > /home/vagrant/malicious
cat /home/vagrant/malicious
dsa
```

Deliverable 1) The modified vfshook.c @ `rootkit-vfshook.c`

Deliverable 2) A shell script (command) that tests the effect @ `rootkit-vfshook.sh`

Deliverable 3) A write-up @ `rootkit-report-vfshook.txt`