CSC/ECE 574 - Project 3*
Systems Security
Assigned October 23, 2019; Due 11:59pm on November 11, 2019

Total Points: 100

Prof. William Enck

Goal: The goal of this project is to learn how to build an access control system for Linux using the Linux Security Modules (LSM) interface. Specifically, students will build an LSM similar to PinUP, which "pins" files such that they can only be accessed by specific applications. In creating this LSM, students will learn concepts of access control, as well as the complexities of writing a Linux kernel module. Students will turn in functional code, as well as a report describing the challenges and answers to a few experimental questions.

Collaboration: Each student must submit their own solution. That said, students are encouraged to discuss the project with each other. Linux kernel programming is particularly tricky, and class discussion will facilitate understanding. Copying code from one another is **explicitly forbidden**. Finally, if you discuss the project or any insights with another student, **you must list their name** on your solution PDF in a clearly denoted "Acknowledgements" section.

Environment Setup

In recent Linux kernels, LSMs cannot be unloaded and reloaded after boot. This complicates LSM development: you need to reboot every time you make a code change. To simplify development, we will use a (very) old version of the Linux kernel that allows you to `insmod` and `rmmod` your LSM.

Step 1: Install a VM environment: VMWare and VirtualBox are two popular VM environments. VirtualBox is open source and free for download. VMWare is available to all CSC students. Download and install one of these. Note that in the past, some (but not all) students reported issues getting a custom kernel to boot on VirtualBox.

Step 2: Setup Ubuntu VM: Download the ISO image for Ubuntu 8.04 LTS. Install the ISO image into your VM environment. I selected half the main system's RAM and 20GB disk is plenty.[1]

*Last revised on October 31, 2019.

[1]I encountered a problem with VMWare Fusion's easy install of Ubuntu 8.04 with respect to the setup of the keyboard. If your arrow keys do not work in the terminal, go to System > Preferences > Keyboard > Layouts. Select Keyboard Model. In my case this was Apple > Apple.

After you install the Ubuntu system, boot it inside a VM. Inside the Ubuntu system you should have access to the network, and be able install a couple of packages that are not part of the initial install. Since Ubuntu 8.04 is no longer maintained, you need to update your sources.list:

Note: Following standard convention, commands starting with $ denote execution by an unprivileged user and commands starting with # denote execution by a privileged user (root).

```
$ sudo sed -i \
  -e 's/archive.ubuntu.com\|us.archive.ubuntu.com\|security.ubuntu.com/old-releases.ubuntu.com/g' \
  /etc/apt/sources.list
```

Next, update the sources and install necessary packages

```
$ sudo apt-get update
$ sudo apt-get install build-essential libncurses-dev
```

Step 3: Build Custom Kernel: From within the VM, download Linux kernel source from kernel.org. Download version 2.6.23 from source. We need a specific version in order to load security modules.

There are many kernel build instructions, but we need to make some small tweaks to ensure that the LSM security options are enabled. The following commands assume you are running as root (use sudo as appropriate).

```
# cd /usr/src ; tar zxvf /path/to/linux-2.6.23.tar.gz
# cd /usr/src/linux-2.6.23
# cp /boot/config-<latest-version> .config
# make menuconfig
```

Under "security options" only "Enable different security models" and "Socket and Networking Security Hooks" should be enabled (as "built-in" or "*"). Save your kernel configuration.

```
# make -j4 all
```

This could take several hours, or faster, depending on your machine.

```
# make modules_install
# make install
```

It would be a good idea to save the state of your VM here.

```
# cd /boot
# update-initramfs -u -k 2.6.23
# update-grub
# vim /boot/grub/menu.lst
```

Make sure the kernel with version 2.6.23 is the first option in `menu.lst`. The configuration should look something like this:

```
title       Ubuntu 8.04.4 LTS, kenrel 2.6.23 Default
root        (hd0,0)
kernel      /boot/vmlinuz-2.6.23 root=UUID=... ro quiet
splash
initrd      /boot/initrd.img-2.6.23
quiet
```

Note: Make sure the first entry contains the *initrd /boot/...* line. It is possible you have duplicate entries for 2.6.23 in this file and the correct one may be at the bottom of the file.

Make sure your configuration is correct. Otherwise, you may get an error such as "kernel panic-not syncing: VFS: unable to mount root fs on unknown block(0,0)." If you get this error, double check your menu.lst file. By pressing ESC during a reboot you can boot back into your old version to fix this file.

Step 4: Compile and Load the Sample LSM: Get the module code from the Mini-Projects page. This code contains two files: (1) sample.c, which contains a Linux Security Module that authorizes file access based on labels associated with files and executables (place in directory `linux-2.6.23/security`) and (2) Makefile, which enables sample.c to be compiled for the kernel (place in `linux-2.6.23/security`).

Note: Make sure to reboot and boot into 2.6.23 before trying load the kernel module.

Once these files are in place, you should be able to run "`make all`" from the root of the kernel source (this will be quick, because all that needs to be built is `sample.c`). This will create `sample.ko` in the `linux-2.6.23/security` directory. You can load the module with "`insmod ./sample.ko`", confirm it is loaded with "`lsmod | grep sample`", and unload it with "`rmmod sample`".

Question 1: PinDOWN LSM {70 points}

The main part of this project is to implementing a simplified version of PinUP[2] that we will call PinDOWN. Start out by reading the PinUP paper. In contrast to PinUP:

- PinDOWN identifies programs by their pathname, whereas PinUP identifies programs by the cryptographic hash of their primary binary. In addition to eliminating the need to reading files and performing hashes within the kernel, this simplification has the benefit of avoiding the need to deal with scripts (e.g., Python code) and software upgrades. However, your PinDOWN code must use the pathname of the loaded program and not the `cmdline` string in the `task_struct`, which can be manipulated by the process.
- PinDOWN file access control policy is a single program file path stored in the XAttr (extended attribute) of the protected file's inode, whereas PinUP file access control consists of lists of cryptographic hashes of programs and corresponding read and write permissions. Note that PinDOWN policy does not even specify read and write permissions, but rather leaves this distinction to the existing Unix file permissions, which is simultaneously enforced.

[2]PinUP is open source, but its code will not be of much use given the simplifications.

- PinDOWN does not have a special utility to set file access control policy, whereas PinUP has special utilities for managing policy. Instead, PinDOWN uses Linux's existing `setfattr` and `getfattr` commands to set and view policy, respectively. For example:

  ```
  setfattr -n "security.pindown" -v "/usr/bin/mutt\0" /home/enck/.mutt/mutt.passwds
  ```

  The consequence of this simplification is that any program can call `setfattr` to change the policy and access the file. However, this is a course assignment and not a production system.

  Note that the "`\0`" in the above command is to null terminate the value of the xattr when reading from the kernel.

To create the PinDOWN LSM, copy `sample.c` to `pindown.c` and rename the corresponding "sample" function, struct, and variable names to "pindown" variants. The code provides the scaffolding to create PinDOWN, and you must complete the sections with "COMPLETE ME" in the comments.

Tips

1. To understand how the LSM frameworks works, it may be helpful to read the LSM Framework paper. The most important part of the LSM framework to understand is the `security_operations` struct that is passed to `register_security`. This structure defines callback functions for all of the different LSM hooks. PinDOWN will only be using a few of these hooks. For more information about what each hook does, look in the `include/linux/security.h` file of the Linux kernel source code. The comments describe each hook's purpose, how it is called, and the semantics of the arguments that are passed.

2. The process control block (PCB) in Linux of type `task_struct`. The `task_struct` of the current running process is available by accessing the `current` variable. LSMs use the `security` field of the `task_struct` to store security information for each process. `security` is a `void *`, which you must allocate (in the `task_alloc_security` hook) and deallocate (in the `task_free_security` hook) and typecast to a structure that you define. This structure should *at least* contain the pathname of the program so that you can compare it to the policy on a file.

3. You will want to capture the path name of the current running binary in the `bprm_set_security` hook, which is called when the binary program is loaded. The `linux_binprm` structure passed to the hook has a member variable, `filename`, which contains the path of the file that is being executed.

   Note: the `task_alloc_security` hook is called on `fork()` and the `bprm_set_security` hook is called on `exec()`, therefore, it is reasonable to copy the current process's `security` values into the child process in `task_alloc_security` and update it to the `exec()`'d program in `bprm_set_security`.

4. The primary access control decision is made in the `inode_permission` hook. You may ignore directories for the purposes of this assignment. If the inode does not have an xattr for

"security.pindown", you should allow access. If there is an xattr for "security.pindown", only allow access if the value string of the xattr matches the program path string you stored when the binary program was loaded.

5. Finally, while the Linux kernel is written in C, knowing how to program in C is not enough. The kernel has its own version of stdlib functions, often prefixed with the letter "k". For example, `kmalloc` instead of `malloc`. Kernel programming is also challenging because mistakes lead to system crashes. Therefore, it is recommended to use your VM snapshot features before loading your module with `insmod`. A large part of learning to program the Linux kernel is learning the naming convention. Once you can "say" the function and structure names (e.g., "`dentry`" stands for "directory entry"), reading kernel code is much easier. There are also a number of macros that perform primitive tasks. If you are having trouble getting started, look at the code for some of the other LSMs, e.g., SELinux.

Write-up for Question 1

1. For your write-up of this question, provide a short documentation for how your LSM functions. Explain when the hooks are called and why it meets the criteria specified for PinDOWN. Additionally, if you have made enhancements to the design, explain them here.
2. Include some screenshots of enforcement with and without PinDOWN.
3. PinDOWN works well in some situations, but not others. Discuss these situations.
4. Provide general observations about PinDOWN and about the project. For example, what was the most difficult part of the project. What advice would you give to future students completing the project?

Question 2: Security Evaluation {15 points}

As described above, PinDOWN is not tamperproof. In this question, we will explore the weaknesses of PinDOWN and suggest ways of addressing them.

1. Write an appropriate threat model for PinDOWN. Note that your implementation might not meet this threat model.
2. Describe how PinDOWN can be modified to meet the threat model you just described.

Question 3: Performance Evaluation {15 points}

In this final question, you will perform a small performance evaluation of your PinDOWN implementation. Find an appropriate file system benchmark and run it both with and without the PinDOWN module loaded. Note that since you are running in a VM and PinDOWN makes minor

changes to the accessing of files, there should be negligible overhead. The purpose of this question is simply to gain experience performing a performance evaluation. You might start at this (older) page describing filesystem benchmarks.

For this question, report tables or graphs that are appropriate for the benchmark that you chose.

Submission Instructions

Submit your solution as two separate files using WolfWare (one `.pdf`, one `.tar.gz`). To upload your assignment, navigate to the CSC574 course. Use the "Project 3" assignment under "Mini-Projects."

The first file should be a single PDF document with your report. **Writeups submitted in Word, PowerPoint, Corel, RTF, Pages, and other non-PDF or ASCII formats will not be accepted.** Consider using LaTeX to format your homework solutions. (For a good primer on LaTeX, see the Not So Short Introduction to LATEX.) The second file should be a tarball (`.tar.gz`) or Zip (`.zip`) of any custom tools or attack captures that are relevant to your report.

Please post questions (especially requests for clarification) about this homework to Piazza.