Introduction to Information Security Systems (CS-457)
Assignment 2

Assigned: 9/4/2021
Tutorial: 9/4/2021
Due: 14/5/2021

Office Hours: 13/4, 15/4, 20/4, 22/4, 11/5, 13/5 14:00-16:00 at
https://bbb.tbm.tudelft.nl/b/eva-fbi-q3j-fp2

Part 1: Access Control System

Introduction

In this assignment, you are going to develop an access control system for system calls, using the C programming language. Your system will restrict an executable (tracee) from invoking a large number of system calls within a short period of time. Your system will receive as input a text file with instructions, as well as an executable to monitor. The instructions file will contain the names of the restricted system calls along with the number of times they can be called within one second. Access control should be enabled only if a sequence of system calls is found anywhere in the executable (tracee). The purpose of this assignment is to provide you with the opportunity to get familiar with the ptrace system call.

Implementation Details

For this assignment, you are going to use *ptrace*, a system call with which a process can observe and control the execution of another process. More information about ptrace can be found here: https://man7.org/linux/man-pages/man2/ptrace.2.html . You will need to use the correct parameters in order to trace the executable and its system calls. Your implementation should use fork in order to execute and monitor the executable given as an input.

The instructions file, given as an input to your system, will contain the names of system calls and the number of times they can be called within one second. For example, if the input file contains the line “gettid 5” then the tracee is not allowed to call the *gettid* system call more than 5 times per second. The first line of the instructions file will be a sequence of system calls. This sequence will be used as a trigger in order to enable the access control system. Your system will constantly monitor system calls but the access control component will be enabled only if this sequence is found anywhere in the tracee. An example of an instructions file can be found in the next section. There is no need to develop a parser for the instructions file. You can safely assume that the input file is always correct and with valid information.

You are free to decide the action that will be taken when the tracee violates the number of times it is allowed to perform a system call. For example, you can simply print a message indicating the violation or you can make the tracee application wait for a specific amount of time.

Please notice that the system call numbers and their symbolic names may be different across architectures. You are free to develop and test your system in your own workstation. However, your final implementation should be compatible with the department's workstations.

Sample Instructions File

The following text file is an example of an instructions file which can be supplied to your system. Each system call should be in a separate line. The instructions of this file are:

- If the tracee program calls *mprotect, mprotect* and *munmap* in that exact sequence at any point in time, then enable the access control system.
- When the access control system has been enabled, the tracee program is not allowed to call *clone* more than 5 times in one second, *mmap* more than 6 times in one second and *gettid* more than 3 times in one second.

mprotect mprotect munmap
clone 5
mmap 6
gettid 3

Implementation Steps

1. Develop a module which is able to read from the instructions file and store the information that you need (e.g. sequence, system calls and their numbers). You are free to implement any data structures that you may need.
2. Develop a module which receives an executable as a CLI argument and executes it using fork().
3. Develop a module which is able to trace the system calls of the tracee using ptrace().
4. Implement the needed functionality in order to restrict the tracee from invoking a system call more than X times within one second.
5. Change your implementation so your system can restrict the tracee based on the instruction file.
6. Change your implementation so the the access control system is enabled only after the sequence, defined in the first line of the instructions file, has been found.

Part 2: Access Control via System Call Modification

In this assignment you are going to modify the *write* and *close* system calls in order to keep track of the last user who changed each file. The kernel will keep this information by generating a file with the following format: <filename>.<last_user>. If this is not the first user who has changed the file, but it is the first time that this user has changed the file, the system will raise an alert. The purpose of this assignment is to give you the opportunity to get familiar with kernel modifications that might be required in order to implement security critical functionality.

Implementation Details

In order to implement this assignment, you first need to locate the source code of the *write* and *close* system calls in the kernel's source tree. Once you get familiar with their implementation, you need to modify their source code in order to keep track of the file that is processed each time they are called, as well as the user ID that issued the system call. Then, you have to keep a log of such activities and raise an alert if it is the first time that the current user has access to the specific file being processed by the system call. Also, you have to modify the two system calls so that a new file will be generated using the format <filename>.<last_user>, where <filename> is the name of the file being processed by the system call (*write* or *close*) and <last_user> is the name of the user that accessed it last. For example, if the user "user1" is the last one to perform a *write* system call on a file named *foo.txt*, then the kernel will generate a file named foo.txt.user1.

The alert can be raised using the printk() function. Once called from kernel space, the message will appear in the kernel's message log. You can access this log by executing $ dmesg

In order to be able to identify your message within the log, you can use the following alert format "<login> ALERT user: <username> file: <filename>" and locate the message by grepping your login. For example, "csd9999 ALERT user: user1 file: foo.txt". Then, this message can be located using $ dmesg | grep csd9999

While in kernel space, you can identify the UID of the user that issued the current system call using the appropriate functions, as in user space applications. The translation between UIDs and user names can be achieved by parsing the "/etc/passwd" file. The "/etc/passwd" file contains the UID-username mappings along with extra information that you do not have to use.
The following "/etc/passwd" entry indicates that the user "csd9999" is assigned with the UID "1000"

csd9999:x:1000:985::/home/csd9999:/usr/bin/oksh

Since the *write* and *close* system calls are issued multiple times during the guest OS's normal execution, you will have to narrow down the monitoring process so that the modified code, performing the file monitoring, is triggered only for files contained in a specific directory. This

process will help you narrow down any possible bugs as well as identify if your modifications are working correctly. For this reason, you can monitor only a specific location, for example /mnt/Documents, and trigger the monitoring only when the file being processed is contained in this directory. For example, the *write* and *close* system calls will first identify the path of the file being processed and if it contains "/mnt/Documents" they will start the monitoring process. If this is not the case, your code will not be triggered.

Optionaly, you can implement the entire functionality as separate functions which you will be able to call within the *write* and *close* system calls. Also, you can implement a few helper functions that translate UIDs to usernames or functions that generate the <filename>.<last_user> files in the user space. You can also keep such files under the directory you choose to monitor or in a separate directory.

Testing the modifications

In order to test your implementation, you first need to modify, compile and execute the new kernel. Thorough instructions describing this process can be found in the following sections. Then, you need to implement a demo application that issues *write* and *close* system calls on files residing in the directory you wish to monitor and execute the demo in the guest OS. In order to assist you in this step, the guest OS contains a root user and three simple users, named "user", "user1" and "user2", each one with a different UID. To be able to perform system calls on the same files with all users, you can create a directory in the guest OS, in a path where everyone has access, and change the permissions appropriately. For example you can create /mnt/Documents, change its permissions so that everyone has access and choose to monitor this directory.

The demo program will issue *write* and *close* system calls to files residing in the directory and you will have to execute the demo program using multiple users. Upon issuing one on the system calls, if it is the first time that this user has changed the file, the system will raise the alert which you will be able to find in the kernel's log. Also, a new file will be generated, having as suffix the user's username as described above.

The entire monitoring process, such as identifying the file's path, generating the <filename>.<last_user> files, UID to username translation etc has to be performed by the modified kernel. The user space demo application is only responsible for issuing *write* and *close* system calls on files residing in the folder you choose.

Translating the UIDs to usernames can also be performed in other ways, you are free to choose whichever process you desire. Also, you can choose the folder you wish to monitor as well as the folder to keep the <filename>.<last_user> files.

Using the Man Pages

A man page describes the functionality of a program, system call or library function. You can see a man page using the *man* command. Should you want to see the man page of function *foo* you can execute $man foo.

foo(N) represents the man page that describes function *foo* under section 'N'. In order to read the man page that describes function *foo* under section *N* you can execute $man -S N foo.
For example, the man page of open(2) can be accessed via $ man -S 2 open.

The following list contains functions that you might find useful

kmalloc, kfree, dentry_path_raw, current_uid, printk, filp_open, filp_close

Linux on QEMU Emulator

Emulators are popular for many reasons. They allow us to install and run an operating system, as simple users, on a machine that runs on a different host OS without having to modify it. The host machine can be used by many users and emulators can execute on it without affecting the users. Also, emulators are particularly useful when we need to develop applications that due to a programming error drive the operating system to crash (e.g. kernel panic). In such cases, we can develop and deploy the applications inside the guest OS and reboot it if it crashes without affecting the host's operating system. For this assignment we are going to use QEMU in order to boot a virtual machine that runs on the modified kernel that we are going to implement.

The QEMU emulator is already installed on the department's hosts and you can access its man pages via $ man qemu. QEMU is able to create and read a virtual disk image where we can install a guest operating system. For the purpose of this assignment, we have installed a simple Linux OS (ttylinux) in 32-bit x86 mode on a virtual disc image which you are going to use in order to test the modified kernel. As a first step, you need to copy this virtual disk image (63MB) from the course's home folder to a directory in your home folder.

```
$ cp ~hy457/qemu-linux/hy457-linux.img ~/<path>
```

This disk image comes with ttylinux pre-installed. It also contains the root filesystem (/) in which some basic tools, like vi and gcc, are already installed. Also, it runs on Linux kernel 2.6.38.1.
Using this virtual disk you can try out the guest OS with the following instruction:

```
$ qemu-system-i386 -hda hy457-linux.img -curses
```

The -hda hy457-linux.img switch instructs QEMU to use the hy457-linux.img file as a virtual disk image, which will appear as /dev/hda in the emulated OS (guest operating system). After executing this command you will be able to see the emulated Linux OS booting up. Once the login prompt appears you can log in either as root, using the "root" user name, or as a simple

user using one of the following user logins: “user”, “user1”, “user2”. The password is “hy457” and is the same for every account.

Important Notice! Always keep backups of the files that you copy inside the guest OS. In cases where the guest OS is forcefully terminated, there is a chance that the virtual disk image gets corrupted and these files get lost. If you reach this point, you can delete the corrupted disk image and fetch a new one from the course’s home folder.

Compiling the modified Linux kernel

The next step is to modify the Linux kernel, in order to implement the assignment, recompile the kernel, generate the bzImage and start the guest OS using the new kernel instead of the one already residing in the virtual disk image. For your convenience, modify and recompile the kernel using the host OS.

First, you need a copy of the kernel’s source code in order to perform the modifications. You can find the source code under ~hy457/qemu-linux/linux-2.6.38.1.tar.bz2.
Once you copy the tar file in your working directory, you need to decompress it in order to modify the appropriate files.

The next step is configuring and compiling the kernel source. The kernel’s configuration file can be found under ~hy457/qemu-linux/.config. You will need to copy this file inside the root directory containing the kernel’s source. In order to be able to identify whether the kernel running on the guest OS is the original or your modified version you need to edit the .config file and locate the CONFIG_LOCALVERSION flag. You can change this field by adding a suffix, such as -csd9999, which is going to appear once you execute $ uname -a at the guest OS. You can also use this field in order to mark specific kernel versions. For example, a possible suffix could be “-csd9999-version_1”.

Finally, in order to compile the modified kernel you can execute $ make ARCH=i386 bzImage. The new kernel image (bzImage) will be placed under linux2.6.38.1/arch/x86/boot/bzImage

Summarizing, the steps required in order to implement and compile the modified kernel are the following, assuming that you use the /spare directories:

$ cp ~hy457/qemu-linux/linux-2.6.38.1.tar.bz2 /spare/[username]/
$ cd /spare/[username]
$ tar -jxvf linux-2.6.38.1.tar.bz2
$ cd linux-2.6.38.1
Edit kernel source code to implement the system call modifications
$ cp ~hy457/qemu-linux/.config . ! the “.” is not a typo

Edit .config, find CONFIG_LOCALVERSION="-hy457", and append to the kernel's version name your username and a revision number.
$ make ARCH=i386 bzImage

Running the modified Linux kernel using QEMU

The final step in testing your implementation is using the new kernel image (linux2.6.38.1/arch/x86/boot/bzImage) in order to start the guest OS using QEMU. In this step you are going to use the same virtual disk image (hy457-linux.img) but this time you are going to instruct QEMU to use the modified kernel. In order to do so, you can execute the following command

```
$ qemu-system-i386 -hda hy457-linux.img -append "root=/dev/hda" -kernel
  linux-2.6.38.1/arch/x86/boot/bzImage -curses
```

The -kernel linux-2.6.38.1/arch/x86/boot/bzImage switchinstructs QEMU to boot using the modified kernel image. The -append "root=/dev/hda" switch instructs QEMU to mount the root filesystem from /dev/hda, which is the virtual disk image. Once you log in the guest OS you can check the kernel's version using $ uname -a. The kernel's version should have the suffix you chose when editing the .config file.

Transferring files between the host and guest OS

In order to transfer files from the guest OS to the host OS and vice versa, you can use the *scp* command. You can access the host OS from the guest OS using its (virtual) IP address 10.0.2.2. For example, in order to transfer the file *test1.c* from the guest OS to the host OS, in your home directory, under the folder hy457 you can execute

```
$ scp test1.c [username]@10.0.2.2:~/hy457
```

from the guest OS. [username] represents the username that is assigned to you (e.g. csd9999). You will also need to insert the password that you use in order to access the department's hosts. In the same manner, in order to copy *test1.c* from the host OS, e.g. milo, inside the guest OS you can execute the following command at the guest's shell:

```
$ scp [username]@10.0.2.2:~/hy457/test1.c .
```

! Attention the «.» that follows «test.c» is not a typo and represents the current working directory!

Notice

1. This is **NOT** a group assignment. Submitted code will be tested for plagiarism using plagiarism-detection software.You have to include your name and login in every file that you are going to submit.

2. Create a README file (30 lines max) that describes your implementation.

3. Place all the required .c and .h files you created for the first part as well as the Makefile, README, the bzImage and the demo programms in a directory and submit them using
   $ turnin assignment_2@hy457 <folder_name>

4. You can use the course's mailing list for questions. However, read the previous emails first since your question might have already been answered.

5. Avoid messaging the TAs directly, using their personal email addresses, since other students might have the same question and everyone deserves the answer.

6. Do not send source code snippets, containing parts of your implementation, to the mailing lists.

7. If you decide to use the /spare directories, fix the permissions so that only you have access.

8. If you decide to use revision tools, such as git, mark the repositories as private.

9. This assignment should be implemented using C on Linux-based machines