CS5231 – Systems Security

Homework 3: Kernel Observability

Due date and time: **23:59 p.m.**, November 13, 2023. This homework MUST be finished independently.

1. Introduction

In this homework, you will be tinkering with eBPF, a revolutionary technology that can run sandboxed programs in the Linux kernel without changing kernel source code or loading a kernel module. Analogous to JavaScript it can manipulate an HTML site when certain HTML events (e.g. mouse clicks) happen, eBPF can be used to run user-defined programs when certain kernel events (e.g. system calls, function entry/exit, kernel tracepoints, network events) happen.

You will be given a program called `malicious_prog`, which will repeatedly do the following operations on each file under the directory `/usr/include/linux`: open the file, copy the contents of the file, and save the copied contents into a new file (with a different name) under the specified destination directory. After the operations on each file, the program pauses for 100 ms. The malicious program and the skeleton files for task 2 can be found on the following link:
https://drive.google.com/file/d/1_nKV7fVoUsUMWLZ1_3FnCLi55sSPFIQo/view?usp=sharing

Your task is to use eBPF and Linux Security Module (LSM) to implement a mandatory access control that prevents unauthorized access to sensitive files.

Environment Setup. Please reuse the Ubuntu 20.04 virtual machine of homework 1 to set up the following environment.

2. Task 1 Hello, eBPF! (5 marks)

Before we begin writing our own eBPF tool, let's see an existing eBPF tool in action and use it to monitor our system.

Preparation: Setting Up BCC – libbpftools

To begin, simply clone the BCC project to your system.

```
$ git clone https://github.com/iovisor/bcc
```

Under the `libbpf-tools` directory, you can see numerous files with `.c` and `.bpf.c`

extensions. These are the so-called eBPF programs, which you will soon write yourselves in the next task. For now, simply build and run the `opensnoop` program.

```
$ cd libbpf-tools
$ git submodule update --init --recursive
$ make opensnoop
$ sudo ./opensnoop
```

`opensnoop` traces the `open()` syscall system-wide and prints various details to the screen. To see how it works, let's go to `opensnoop.bpf.c`. You can see that at around lines 50 and 70, there are two function definitions:

```
SEC("tracepoint/syscalls/sys_enter_open")
int tracepoint__syscalls__sys_enter_open() { … }

SEC("tracepoint/syscalls/sys_enter_openat")
int tracepoint__syscalls__sys_enter_openat() { … }
```

These functions contain the code that will be executed when the specified event is triggered. The event of interest is specified just before the function definition using the `SEC()` macro. In this case, the function will be called each time an `open()` or `openat()` syscall is invoked from any user-space application.

Task Description: Generating and Analyzing Audit Logs

Using `opensnoop`, your task is to record the behavior of `malicious_program` and write the output to a file called **`task1.log`**.

```
$ sudo ./opensnoop
$ timeout 5m ./malicious_prog ./output
```

Having produced the audit logs, your task is to clean and analyze them. From the audit logs, you should write a program (using any language you prefer) to **extract the top ten most accessed files under the directory `/usr/include/linux`**. Your output should only consider the files accessed by `malicious_prog`. If there are multiple files that have the same number of accesses, you should break ties by selecting the one that appears first when sorted in lexicographical order. In your output, simply include the filename and the number of accesses, as shown below:

```
// top10.txt
/usr/include/linux/bpf.h 180
/usr/include/linux/cec.h 143
/usr/include/linux/acct.h 99
/usr/include/linux/sctp.h 99
...
```

3. Task 2: eBPF with LSM (10 marks)

Your next task is to implement your own eBPF program to implement file-level access control. To achieve this, we will be attaching our eBPF program to hooks introduced by the Linux Security Module (LSM).

During its execution, `malicious_prog` will read many files under the directory `/usr/include/linux`, and two of them are: `if.h` and `un.h`. In this task, we will mark these two header files as *sensitive* files. Your goal is to **prevent `malicious_prog` from accessing these two sensitive files** and **log the incident** if such access is detected. For the logging part, print a line *"Access to sensitive file {filename} is detected*." to `/sys/kernel/debug/tracing/trace_pipe`. **Note that you just need to detect based on the file name, instead of the full path of the file.**

3.1. Setting Up libbpf

The Linux kernel expects eBPF programs to be loaded in the form of bytecode. Since it is very hard to write bytecode by hand, eBPF programs are usually developed through projects such as BCC, Cillium, and bpftrace. In this task, we will be using libbpf, a C-based library that can generate and load the eBPF byte code into the Linux kernel.

To begin, clone the `libbpf-bootstrap` project locally to your machine:

```
$ git clone --recurse-submodules
https://github.com/libbpf/libbpf-bootstrap
```

Then, go to the `examples/c` directory and compile the `minimal` application:

```
$ cd examples/c
$ make minimal
```

After that, verify that you can run the binary and see the output

```
$ sudo ./minimal
$ sudo cat /sys/kernel/debug/tracing/trace_pipe
```

Once you can see the trace logs, copy the skeleton code we provided (`cs5231.c, cs5231.bpf.c, Makefile`) to the `examples/c` directory.

3.2. Linux Security Module

The LSM framework works by introducing hooks into a wide variety of kernel functionalities. To use these hooks, one need to define callback functions which will return 0 if the access to the requested resource is valid and error (such as –EACCES) otherwise. Traditionally, these "modules" must be specified in the build time (i.e. compilation) of the kernel, making the development process very cumbersome. However, this changes when eBPF support was added. Now, using eBPF, we can

simply specify which LSM hook our code should attach to, and the kernel will happily run our eBPF code for us.

LSM and eBPF should be available out of the box on modern Linux distributions. However, depending on your Linux distribution, it might not be enabled by default and you need to enable it by editing the `lsm` kernel parameter. You can run the following commands to check and modify the kernel parameters during boot time:

```
$ cat /sys/kernel/security/lsm
# if you can find 'bpf', then you are already good to go
$ sudo vim /etc/default/grub
GRUB_CMDLINE_LINUX="lsm=capability,landlock,lockdown,yama,app
armor,bpf"
$ sudo grub-mkconfig –o /boot/grub/grub.cfg
$ reboot
```

At this point, you might wonder what hooks are available and how can you use them. If you are using Linux kernel prior to v6.4, the full list of the hooks exposed to security modules, along with their description can be found under `include/linux/lsm_hooks.h` in the kernel source code. Starting from v6.4 onwards, the hooks docstrings are moved to `security/security.c`. You can find the online version of the Linux kernel source code via these links:

(6.3.13) https://elixir.bootlin.com/linux/v6.3.13/source/include/linux/lsm_hooks.h

(6.5.8) https://elixir.bootlin.com/linux/latest/source/security/security.c

Your task: You fill the `BPF_PROG` function with the correct logic and attach it to the appropriate LSM hook.

To compile and run your program, you can use the same commands as compiling and running the `minimal` libbpf program:

```
$ make cs5231
$ sudo ./cs5231
$ sudo cat /sys/kernel/debug/tracing/trace_pipe
```

4. Submission

You are required to write a report for this homework. For task 1, your report should explain the steps taken by your audit log analyser program to get the top 10 most accessed files. For task 2, your report should explain: 1) which LSM hook your program attaches to; and 2) how does your eBPF program prevent `malicious_prog` (and only `malicious_prog`) from accessing the sensitive files. Additionally, you should include a screenshot of your eBPF program in action inside your report.

Your submission into Canvas is a single .zip file. Please name this file with the following format: ***your_matric_id*-hw3.zip**, e.g., A0123456X-hw3.zip. The .zip file should include **your report**, **task1.log**. **top10.txt**, **your audit log analyser**, and your eBPF program (**cs5231.c** and **cs5231.bpf.c**).

```
A0123456X-hw3
 ├── cs5231.bpf.c
 ├── cs5231.c
 ├── report.pdf
 ├── task1-analyser.c
 ├── task1.log
 └── top10.txt
```

Note that we only accept the following formats for compressed file: .zip, .tar.gz, .tar.bz, and .tar.bz2. Also, please make sure your report does not **EXCEED 3 pages**, excluding references and appendices.

For any questions, please contact the teaching team via cs5231ta@googlegroups.com. Hope you can have fun in exploring the basics in systems security.

5. Reading Materials

Below are some useful resources to get you started with eBPF and LSM:

1) https://nakryiko.com/posts/libbpf-bootstrap/
2) https://www.brendangregg.com/blog/2019-01-01/learn-ebpf-tracing.html
3) https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md
4) [Book] Learning eBPF, by Liz Rice. The code is available online at https://github.com/lizrice/learning-ebpf
5) https://docs.kernel.org/bpf/prog_lsm.html