EXTENDED BERKELEY PACKET FILTER (EBPF) – THE NEW SWISS KNIFE FOR CYBERSECURITY EDUCATION

Si Chen and Liu Cui
Computer Science Department
West Chester University of Pennsylvania
{schen, lcui}@wcupa.edu

ABSTRACT
It is very challenging to do hands-on projects involving complex Linux kernel modification in cybersecurity courses, such as pawning a Linux kernel and hooking system calls. To solve this problem, we have developed a cybersecurity learning system named BadgerCTF+, which leverages the Extended Berkeley Packet Filter (eBPF) technology to support just-in-time kernel hooking. eBPF is a kernel technology that allows programs to run in the kernel land without changing the kernel source code or adding additional modules. With BadgerCTF+, we can develop various hands-on cybersecurity labs, including system calls and processes hijacking, attacking interrupts, and kernel tracing. While the BadgerCTF+ was initially developed for cybersecurity courses, it can also be used for network courses to learn how Internet technologies work, such as network monitoring, route tracing, Border Gateway Protocol (BGP), packet inspection, and network performance analysis.

KEY WORDS
eBPF, cybersecurity, Linux, kernel

1. Introduction

Linux has emerged as a widely used platform for teaching cybersecurity topics. Its open-source software base and widely available development tools make it easy for students to access its internals. It provides real-world code examples and thereby also enhances students' programming skills that can be immediately applied in the workforce after graduation. The kernel programming projects provide crucial hands-on experience for helping the student understand operating system concepts and advanced cybersecurity topics such as system calls, process hook, interrupt, and kernel memory management. However, developing a pedagogically-effective programming project in the context of a complex Linux kernel codebase can be a challenge [1]. This is because the Linux kernel has been developed and maintained for more than 30 years and is already settled in design. The sheer size of the kernel system makes it difficult to understand. The production operating systems are also not primarily designed as teaching tools -- any changes students make might break the whole system [2].

Therefore, designing lab activities involving the Linux kernel programming at ring 0 (e.g., the kernel land) has been a challenging problem in cybersecurity education. A common approach is to develop a modded device driver module called Loadable Kernel Module (LKM) and insert it into the Linux kernel. Although useful for demonstrating some concepts, these approaches are complicated to build and maintain. Moreover, to fully understand the modded driver's code, students need to have the background knowledge of Linux device driver APIs and Linux kernel subsystems and must be fluent in C programming.

The main contribution of our work and the main difference separating our work from other work is how we apply the eBPF technology to Linux kernel programming. We have developed a Docker-based cybersecurity learning system (similar to [3]) named BadgerCTF+, consisting of labs and hands-on projects for each essential topic of the Linux kernel, including system calls, interrupt-descriptor table (IDT), SMEP/SMAP, use-after-free (UAF) vulnerability. The construction is then compiled into container files for Docker to run on our server.

2. Background

2.1 eBPF

Berkeley Packet Filter (BPF) was first developed in 1992. It is a virtual machine (VM) in the Linux kernel, allowing a privileged user to load and run bytecode safely in the kernel and monitor some chosen events. Since version 3.18 of the Linux kernel, the BPF VM has been extended with a new name eBPF(extended BPF). In order to trigger a BPF program, one needs to attach it to one or more probes (e.g., kprobe or uprobe).

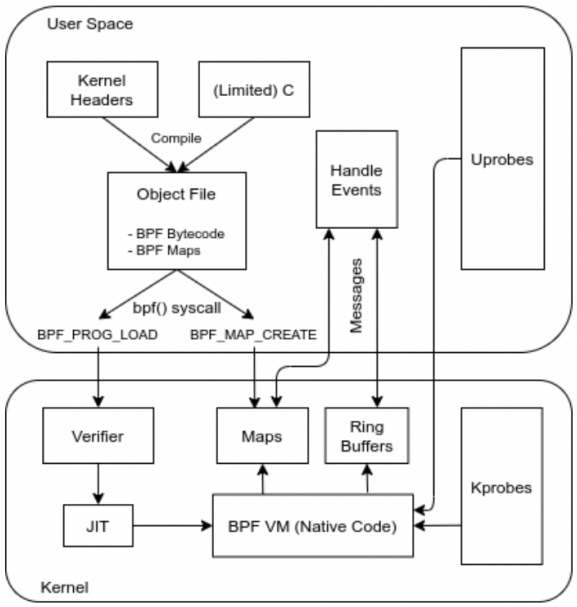

Figure 1: eBPF Components

Figure 1 shows a high-level diagram of the eBPF components. By allowing to run sandboxed programs within the operating system, teachers can run eBPF programs to add additional capabilities to the operating system at runtime. The operation system then guarantees integrity and safety as if natively compiled with the aid of a JIT (Just-in-Time) compiler and verification engine.

2.2 Kprobes and Uprobes

Kprobes (kernel probes) and uprobes (user probes) are mechanisms in the Linux kernel to dynamically set breakpoints at any desired address. You can attach the probe in either the kernel space (ring 0) or in user space (ring 3), specifying a handler function to be invoked when the breakpoint is hit.

3. System Overview

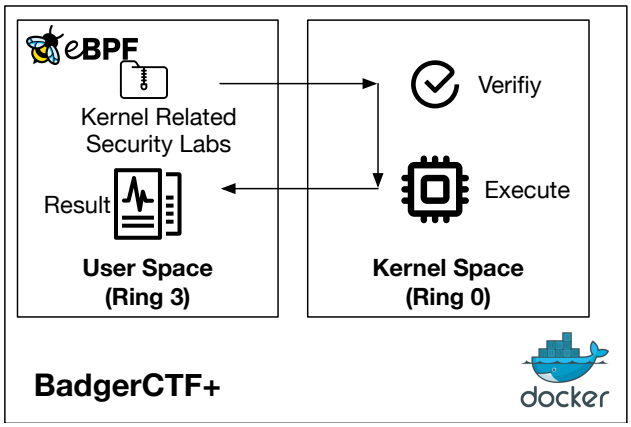

Figure 2: BadgerCTF+ System Architecture

We maintain that eBPF is a natural fit for a cybersecurity learning system, for several reasons:

- Hooking is done in the kernel space, thus is able to hook high privilege processes.
- The verifier automatically verifies the eBPF code before being inserted to the kernel, and prevent programming errors which may cause kernel crashes, hangs, or instability.
- eBPF programs can be triggered by user or kernel land probes, allowing a single mechanism to intercept all system events.
- eBPF is able to provide context about the events, including arguments, process id, user id, timestamp and more.
- eBPF supports event filtering in the kernel, saving the need to send and parse irrelevant event in the userspace (ring 3).
- It is possible to use eBPF to read and write data from kernel space to user space memory, hence arguments can be read and even changed.
- System overhead is relatively small as all the eBPF code are compiled to native code and are running inside kernel space.
- eBPF is now maintained as part of the Linux kernel and new features are constantly being added to it.

Because of the aforementioned reasons, we choose to build our cybersecurity learning system – BadgerCTF+ on top of eBPF. Figure 1 shows the system architecture of the BadgerCTF+ system. The core part of the BadgerCTF+ system is a pre-configured docker container. Users (student and teacher) can connect to the BadgerCTF+ system via a secure shell (SSH) program. Once connected via SSH, the BadgerCTF+ system will automatically spawn a new container and let the user attach to it. To make sure user's homework can be saved and synchronized, BadgerCTF+ will automatically mount user's home folder to the container. Once user disconnect via SSH, that container will be destroyed.

We use eBPF to build several kernel related security labs. These Linux kernel programming labs are developed via pseudo-C code and compiled into eBPF byte code via LLVM compiler. Student and teacher do not need to use eBPF directly, instead, they can use the Python script as an abstraction layer to interact with the built-in eBPF program.

Our lab's Python script will create a new system-level hook to gather information in the kernel land. Once the desired hook has been identified, the eBPF program can be loaded into the Linux kernel via the eBPF system call. This is typically done using one of the available eBPF libraries. As our lab program is loaded into the Linux kernel, it passes through two steps before being attached to the request hook:

1. **Verification:** As shown in Figure 2, the verification process ensures that the eBPF program is safe to run. The verifier will validate that the program meets several conditions, including:
    a. The process loading the eBPF program holds the required privileges (Note that inside the BadgerCTF+ docker container, users have the root privileges by default).
    b. The program does not crash the system.
    c. The program does not contain any infinite loop and always runs to completion.
2. **JIT Compilation:** The Just-in-Time (JIT) compilation process translates the generic bytecode of the program into the machine specific instruction set to optimize the overall running performance of the program. This makes our eBPF programs run as efficiently as natively compiled kernel code or as code inserted as a Loadable Kernel Module (LKM).

To return the result back to user space (ring 3), our system utilized a concept named eBPF maps. eBPF maps allows sharing date between our lab's eBPF kernel programs and also between kernel (ring 0) and user-space (ring 3) applications. The eBPF maps are generic data structure for storage of different data types and are treated as binary blobs. We specify the size of the key and the size of the value at map-creation time inside our lab's eBPF program. The map handles are file descriptors and multiple maps can be created and accessed by multiple users/programs at the same time.

	Monitor all system interactions	No system modification	No application modification	Easy to develop	Safe
Loadable Kernel Module (LKM)	N	N	Y	N	N
Application Modification	N	Y	N	N	N
Library Injection	N	Y	Y	Y	N
Kernel eBPF (BadgerCTF+)	Y	Y	Y	Y	Y

Table 1: Comparison of Techniques for Teaching Linux Kernel

4. Comparison of Techniques for Teaching Linux Kernel

Over the years, several techniques were suggested and implemented for teaching Linux kernel concept. While some of these approaches were shown to have good result for building class demo, they also suffer from developing student project with moderate difficulty. Table 1 provides a summary of the result of this comparison. By relying on the eBPF technology of the Linux kernel, BadgerCTF+ achieves some of the desired requirements. BadgerCTF+ is able to trace almost any part of the Linux system kernel. It can log Linux API, native library functions, system calls and internal kernel functions. By collection all context levels of an application in a unified manner, BadgerCTF+ is able to provide a bunch of programming interface which can be directly used by the student for secondary development.

5. Lab Examples: Detecting Kernel Rootkit

In this section, we demonstrate how to design a lab using eBPF technology for CSC 471: modern malware analysis class. The topics is for detecting kernel rootkit. It is difficult to teach topics like kernel rootkit due to it complex nature. Before introducing eBPF into BadgerCTF+ system, we mainly use Volatility [4] for analysing the memory dump file of the malware. The limitation of using Volatility is student can only perform static analysis on the memory dump file. With eBPF, student can perform both static analysis and dynamic analysis of the malware sample on our BadgerCTF+ system.

5.1 Kernel rootkit and hooking

A kernel rootkit is a malicious software that runs inside the kernel space to create a backdoor to a system without being detected. It can be de deployed onto a system via a worm, or an attacker can use a kernel vulnerability. Because kernel rootkit involves compromising the kernel, they can basically do anything, including avoiding detection by other software. A common technique of Linux kernel rootkit is to overwrite the function addresses inside the syscall table. Syscalls are the main way userland applications interact with the kernel and underlying hardware. By hooking (or 'altering') the syscall table, rootkit can change the data reported by the kernel to hide anything incriminating, such as its own process or a network connection to a command and control (C&C) server.

In this lab, we use an open-sourced LKM rootkit named *Diamorphine* [5] as an example. It hooks three syscalls: *Kill*, *Getdents* and *Getdents64* to hide itself.

5.2 eBPF syscall hooking program

The eBPF programs used in our cybersecurity lab are event-driven and it will be triggered when the kernel or an application passes a certain hook point. There are several pre-defined hooks which include system-calls, function entry/exit, kernel tracepoints, and network events.

```
1  int syscall__ret_execve(struct pt_regs *ctx)
2  {
3      struct comm_event event = {
4          .pid = bpf_get_current_pid_tgid() >> 32,
5          .type = TYPE_RETURN,
6      };
7
8      bpf_get_current_common(&event.comm, sizeof(event.comm));
9      comm_event.perf_submit(ctx, &event, sizeof(event));
10
11     return 0;
12 }
```

Figure 3: Creating an eBPF Hook for System Call

If a predefined hook does not exist for a particular need, we should use kernel probe (kprobe) or user probe (uprobe) to attach eBPF programs and retrieve the data (Shown in Figure 3).

In this lab, we will attach a kprobe to the *Kill* syscall function. Since eBPF has the ability to record stack traces of a function call and shows what functions were called in

both user space and the kernel space, we can therefore detect when a function or syscall has been hooked by monitoring any suspicious function insertion. Our eBPF lab program is able to record the stack trace from all *Kill* syscalls and print it out afterward.

5.3 Rootkit detection program

Student can use either Libbpf or BPF Compiler Collection (BCC) (both are powerful set of tools which uses eBPFs) for creating resourceful rootkit detection program based on kernel tracing. The main programming language for our lab is C, but student can also choose to use Lua, C++, Python or Rust. The Libbpf acts as an eBPF program loader. It can load, checks and relocates eBPF programs, sorting out maps and hooks. In our lab guidance sheet, we provide the template code using libbpf-bootstrap. Student needs to figure out how to implement the *handle_event* function to use the provided eBPF syscall hooking program and read the result.

For this lab, student will quickly notice that a new stack frame was inserted in between two expected stack traces which indicates that the syscall had been hooked by the *Diamorphine* rootkit.

6. Lab Examples: Use eBPF in network class

In this section, we demonstrate how to design a lab using eBPF technology for CSC 335: Data Communication and Networks, and CSC 302: Computer Security. Network labs are used to be a challenge for us due to the number of machines and students we have. With eBPF, connected containers are created, so more network and large scale network labs can be done.

5.1 Socket filtering

One of the current labs in CSC 335: Data Communication and Networks is socket programming where students write python programs for one server and multiple clients services.

As the next step, in CSC 302: Computer Security, students could implement their own socket programs in BadgerCTF+ and enable the socket filter manually. With eBPF, network traffic can be filtered by query name, DNS name, and address. [6] In addition, students implement eXpress Data Path (XDP) to prevent distributed denial of service (DDoS) attack. In order to do so, students provide programs of the “BPF_PROG_TYPE_XDP” type to a network interface. Then, the kernel will execute the programs on received packets before networking stack starts processing them.

5.2 TCP Congestion Control

TCP congestion control is another place to implement the lab. In CSC 335: Data Communication, TCP is introduced in detail as one protocol in the network layer. Before badgerCTF+ is implemented, the three-way handshake, re-transmission, and TCP header information are examined by students using Wireshark. The TCP congestion control is only introduced in the lecture, without a comparison among different TCP congestion control algorithms.

To include research in the class and inspire students solving everyday problems, TCP congestion control using eBPF is adopted. In this lab, TCP Tahoe algorithm is explained in detail with implementation, and other TCP congestion control algorithms such as new Reno and Vegas are provided. Students choose at least two TCP congestion algorithms to implement and compare the performance different network traffic setting. Then, students write a lab report to explain the observation.

7. Conclusion

In this paper, we showcased how eBPF, a code augmentation framework offered by the Linux kernel, can be used for cybersecurity and network education. To prove the effectiveness of the idea, we develop an eBPF based learning system named BadgerCTF+. Our lab examples indicated that eBPF can be used for teaching Linux kernel programming courses malware analysis. The in-kernel measurement via code augmentation ca be used to gather data to feed toolkits for detecting kernel rootkit. We also show that eBPF can be used for designing labs for network courses.

References:

[1] O. Laadan, J. Nieh, en N. Viennot, “Structured linux kernel projects for teaching operating systems concepts”, in *Proceedings of the 42nd ACM technical symposium on Computer science education*, 2011, 287–292.

[2] R. Hess en P. Paulson, “Linux kernel projects for an undergraduate operating systems course”, in *Proceedings of the 41st ACM technical symposium on Computer science education*, 2010, 485–489.

[3] C. E. Irvine, M. F. Thompson, M. McCarrin, en J. Khosalim, “Live lesson: Labtainers: A docker-based framework for cybersecurity labs”, in *2017 USENIX Workshop on Advances in Security Education (ASE 17)*, 2017.

[4] M. H. Ligh, A. Case, J. Levy, en A. Walters, *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory*. John Wiley & Sons, 2014.

[5] P. Krishnamurthy, H. Salehghaffari, S. Duraisamy, R. Karri, en F. Khorrami, “Stealthy rootkits in smart grid controllers”, *in 2019 IEEE 37th International Conference on Computer Design (ICCD*), 2019, bll 20–28.

[6] The Linux Kernel, “Linux Socket Filtering aka Berkeley Packet Filter (BPF)”, https://www.kernel.org/doc/html/latest/networking/filter.html, accessed on Feb 2nd, 2022.