École Polytechnique Fédérale de Lausanne

Effectiveness of Control-Flow Integrity in the Linux Kernel: an Empirical Study

by Sacha Kozma

Master Thesis

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Dr. sc. Anil Kurmus
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

March 15, 2021

Acknowledgments

First, I would like to thank my supervisor, Anil Kurmus, for this internship at IBM Research. His help and guidance throughout the project were very precious, and working at his side during this semester allowed me to learn a lot from his experience in software security. Also, all the software security team at IBM was very welcoming, and it was a pleasure to work with them.

I would like to thank my advisor, Mathias Payer, for his help and availability throughout the semester. I'm also very grateful for his help at finding an internship in the software security field.

A special thanks to my friends and family for all the support during this unusual period disturbed by the pandemic. In particular, a very big thanks to my sister Lydia who was always there to clear my head during these mentally-challenging days of home-office.

Finally, I thank the HexHive laboratory for providing me the template [18] of the present thesis.

Lausanne, March 15, 2021 Sacha Kozma

Abstract

Due to its wide adoption in the server and mobile world such as in Android, the Linux kernel is a target of choice for attackers. Moreover, new features are constantly added and this opens up a lot of opportunity to find new breaches. On the other hand, Linux developers try to mitigate attacks using ever-evolving protection such as a recent one called Control-Flow Integrity, which has the potential of greatly reducing the impact of current attacks. New attack strategies like data-only exploitations are thus being adopted and it is not clear how current mitigations cope with them.

In this work, we then analyze the effectiveness of Control-Flow Integrity in Linux by looking into a large quantity of bugs in the kernel to determine the proportion of bugs whose exploitation is truly protected by this mitigation. The results show that a large majority of bugs is still exploitable using modern attack strategies. We also detail the exploitation of three bugs to demonstrate the power of data-only attacks. The resulting exploits allow an unprivileged attacker to completely take over the vulnerable system with root privileges.

Contents

Acknowledgments **2**

Abstract **3**

1 Introduction **6**

2 Background and related work **9**

2.1 Kernel control-flow 9
2.2 Kernel memory management 11
2.2.1 The Buddy Allocator 11
2.2.2 The SLAB Allocator 12
2.3 Kernel exploitation 13
2.3.1 Memory corruption bugs in the kernel 13
2.3.2 Control-Flow hijack attacks 14
2.4 Control-Flow Integrity 15
2.4.1 Presentation 15
2.4.2 Implementation in the Linux Kernel 16
2.4.3 Data-Only Attacks 18

3 Method **20**

3.1 Analyzing bugs in the Linux kernel 20
3.1.1 Heap out-of-bounds write 21
3.1.2 Heap out-of-bounds read 25
3.1.3 Use-after-free 25
3.1.4 Double free 26
3.1.5 Warnings, debug information, deadlocks and others 26
3.2 Practical data-only exploitations 26

4 Results **28**

4.1 Classification 28
4.2 Attacks implementation 29
4.2.1 End-to-end exploitation of bug 557d015f 29
4.2.2 Revisiting CVE-2017-7308 40
4.2.3 Revisiting CVE-2017-11176 42

5 Discussion **46**

6 Conclusion **49**

Bibliography **50**

Chapter 1

Introduction

The kernel is a core component in operating systems: it acts as an abstraction of the hardware layer to provide a simpler interface that programmers use to interact with the machine running their software. It has essential responsibilities, such as managing the memory or scheduling the user processes, which means that the kernel is also the most privileged piece of code running on a system. It is then natural that it became a target for attackers seeking to compromise a vulnerable machine. Managing to attack the kernel could lead to a lot of powerful capabilities such as getting *root* accesses to the system, escaping containerized environments or hiding malicious activities from the user. In the Linux kernel, such attacks are often made possible by the existence of memory corruption bugs. They stem from the extensive usage of the C language in the code base, which is known for its efficiency but also for relying on the capacity of the developer to write correct programs. Ensuring memory safety is then left to the programmer and this frequently leads to errors. Moreover, the Linux kernel contains a lot of different subsystems to support a large variety of functions, hardware or protocols and new ones are constantly added, so keeping up with finding and fixing the bugs is not an easy task.

By design, the Linux kernel separates the memory available to the user processes (user-space) from the memory available to the kernel (kernel-space). In this way, the processes are prevented from directly modifying the kernel objects and instead have to use *system calls* to interact with the kernel. However, this is not enough in the presence of a memory corruption bug. Attackers have found ways to take over the vulnerable system by corrupting important data in memory, such as function pointers. In this case, calling a corrupted function pointer would divert the kernel from its original execution flow and let the attacker execute unwanted instructions on the machine. In this context, modern kernels try to block attacks with several protections: first, the execution of user-space memory regions is forbidden so that jumping to a portion of code stored there crashes the kernel. However, user-provided data are often copied to kernel space, for example when sending packets through the network, so the kernel also restricts each memory chunk to be either writable or executable, but never both at the same time. This prevents attackers from injecting their payload in kernel-space. The attackers now have to rely on more advanced techniques that re-use portions of the kernel code itself to build the malicious payload. Although the kernel tries to mitigate this attack vector by randomizing the code position in memory, the low entropy [7] used during the process and the quantity of sensitive kernel pointers that are leaked by the kernel to the user due to *information leak* bugs make KASLR often easily bypassable.

Now, a mitigation called Control-Flow Integrity (CFI) [1] is being adopted by the Linux kernel and has the potential of further limiting the quantity of kernel code available to the attacker. The main idea is to restrict the number of locations at which the kernel can jump to at each change in the control-flow. For example, with CFI the kernel will only call a function pointer if it points to the beginning of a function with the same prototype (same return type and same argument(s) type(s)) than the one used in the source code. This is a major reduction to the set of jump targets available to the attackers as they are now restricted to the start of a limited number of function while they were allowed to jump to any bytes in the kernel code before CFI. Although the adoption of CFI in the Linux kernel is relatively recent, its usage in user-space applications is well-established and has thus been extensively studied, either for its security, performance or usability aspects [6, 40]. However, the same is not true in the kernel: the literature concerning the effectiveness of CFI at blocking kernel-oriented attacks is lacking and it is not clear how useful this mitigation is in this context. This skepticism stems from the hypothesis that the kernel might be a target subject to a wide variety of *data-only attacks* [33], a class of attacks that focuses on corrupting semantic data (password, administrator flag, ...) rather than *control-data* (function pointers, return addresses, ...). Indeed, the kernel stores a lot of other sensitive data in its memory such as page tables, files permissions or processes privileges and corrupting them poses a severe security threat too.

In this thesis, we then try to give a practical analysis of the effectiveness of CFI in the kernel by looking at a large quantity of bugs found by fuzzing the kernel. We then determine the proportion of them that would allow bypassing CFI during a data-only attack. To carry this analysis out, it is important to first understand and describe the attack surface available to an attacker seeking to perform a data-only attack against the kernel. We then gather 102 bugs to assess their exploitability with respect to the collected attack strategies. Finally, we build several practical exploitations on one of these bugs as well as on some existing vulnerabilities to support the reliability of the analysis and demonstrate the power of data-only attacks. The results we obtain confirm the skepticism: the remaining attack surface is significant and mostly unprotected even in modern kernels. Hence, 80% of the vulnerabilities we evaluate are still exploitable even with the presence of Control-Flow Integrity. Also, all the exploits we implement manage to take over the vulnerable system with administrator privileges, showing the power of data-only attacks are capable attack vectors.

In summary, the core contributions of this thesis are 1) the evaluation of 102 bugs in the kernel to determine the effectiveness of Control-Flow Integrity at blocking attacks against the vulnerable ones and 2) the implementation of three data-only attacks on real vulnerabilities that bypass every modern mitigations (including KASLR and CFI) and manage to escalate the privileges of the attacker by getting *root* access to the vulnerable system.

Chapter 2

Background and related work

In this section, we describe the material required to the comprehension of the thesis. We first cover the difference between the control-flow of a user-space program and the kernel. The latter having several unique attributes, highlighting them is important to understand how exploitation and mitigations differ between the two contexts. We then describe the management of dynamic memory allocation in Linux, as it has a core role when exploiting the kernel. We also cover the basics of kernel exploitation such as how memory corruptions impact the kernel and the different manners attackers can take advantage of them, be it using control-data or data-only attacks. Finally, we introduce the Control-Flow Integrity mitigation in the Linux kernel.

2.1 Kernel control-flow

A significant difference to a user-space program is that the kernel is concurrent by nature. It is called *reentrancy* and it means that several pieces of kernel code can be executed at the same time on a single machine. This can happen under different scenarios: for example, modern processors are designed such that a single processing unit is in fact composed of multiple *cores* that run different programs in parallel. Each one of them can then independently interact with the kernel using system calls, which might end up having several cores running kernel code at the same time. It could even happen on a single core as Linux is a *preemptive* kernel: if, during a system call, the kernel cannot serve the request immediately, it will deliberately choose another process to run and will pause the current one in the mean time. In this situation, two interactions with the kernel could be interleaved. Also, in order to communicate and exchange data with the machine, the hardware connected to the system issue *interrupts* to get the attention of the kernel, and the corresponding *interrupt handlers* are then called to take the appropriate actions. These interrupts can happen at anytime, and if the CPU is already running a kernel path at that moment, it is paused to handle the request. A similar scenario can happen with exceptions: if an anomalous condition is detected while executing kernel code (for example due to a bad memory

```c
struct file_operations {
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    __poll_t (*poll) (struct file *, struct poll_table_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    (..)
};
```

Listing 2.1 – A structure used to store function pointers for each action executable on a file.

access), the current execution is paused and a procedure is called to take care of the exception. Again, in these two cases a new kernel path is executed on top of another one.

Due to these interactions, the kernel needs to be protected against concurrent accesses to shared data such as global variables. Otherwise, a kernel path could modify data that are being accessed by another thread and create a *race condition*. The kernel developers have several tools at their disposal to protect them, such as *spinlocks*, *mutexes* or *Read-copy-update* [36], but it is often not trivial to consider every possible interactions and errors are frequent. Moreover, due to their non-deterministic property, race conditions are hard to uncover and can remain buried in the source code for a long time before being found and fixed. From an exploitation point of view, race conditions are responsible for different type of bugs, such as memory corruptions which open up opportunities for attackers to compromise the kernel.

Another particularity of the Linux control-flow resides in the way it is programmed: although the code base is predominantly in C, which is not an object-oriented programming language, there is an extensive use of the *dynamic method dispatch* pattern within the kernel. More precisely, a lot of structures declared in the code contain function pointers which are dynamically set at run-time depending on the context. In this way, the kernel can operate on a lot of different formats or protocols without having to create copies of the same structure for each of them. Listing 2.1 gives an example of such a structure, which in this case stores pointers to functions related to the operations one can perform on a file. The implementation of how to read, write, open or release a file can be dynamically adapted depending on the situation, for example according to the underlying filesystem. A pointer to this structure can then be easily stored in an object describing a file in the kernel and serve to dispatch control flow to the correct function. Due to this abundance, function pointers are a privileged target for attackers who then try to corrupt them: we will see in the next sections how it is done in practice.

2.2 Kernel memory management

As in user-space, memory in the kernel can either be statically, automatically or dynamically allocated. Static memory allocations concern memory that is allocated at build time: a fixed region of memory is designated to store variables that live during the whole kernel execution. For example, variables declared globally in the code base are statically allocated. The memory region containing the static allocation is called the kernel *data section*. Otherwise, for variables with a shorter lifespan, such as local declarations in a function, the kernel automatically stores them on the *stack*, a memory region created at boot time. However, the size of the stack in the kernel is limited and kernel developers have to use it sparingly. Thus, when more complicated allocations are needed (bigger sizes, different lifespans, ...), they can rely on dynamic allocation: in this case, they are responsible for the allocation and the freeing of the memory chunk by calling the appropriate functions in the kernel. The memory region used to serve dynamic allocation requests is called the *heap*.

However, for performance reasons, the kernel does not let developers allocate memory at any location in the heap. Instead, it relies on two systems called the *Buddy Allocator* and the *SLAB Allocator* to manage the heap. We now give a highlevel explanation of the two systems.

2.2.1 The Buddy Allocator

In Linux, the basic unit used in memory management is called a page. The size of a page matches the size of a physical page, which is determined by the *Memory Management Unit* (MMU) of a given machine. For example, the most frequently seen page size on a x86-64 machine is 4096 bytes, so for the rest of the thesis, we will assume that this is the default page size. Now, the Buddy Allocator is used for memory requests which sizes are a multiple of a page size: it is then mainly used for large allocations.

One of the main challenges for the Buddy Allocator is to prevent *fragmentation* of the kernel heap: this happens when large chunks of memory are non-continuously allocated, while the remaining "holes" are free but unusable for most requests due to their small sizes. In this scenario, the memory is not optimally managed and this results in a lot of free space being wasted. To prevent fragmentation as much as possible, the Buddy Allocator groups memory chunks by their *order*, which can range from 0 to a fixed upper limit. The size of each chunk is then computed as $2^{order} * PAGE\_SIZE$, so that it is exactly twice the size of a chunk from the lower order. Now when the kernel requests a memory chunk of a specific size, the size is rounded up to the next order and the Buddy Allocator returns a chunk from the list of available chunks for that order. These lists are called freelists. However, it is possible that after several allocations of the same order N the freelist of order N is emptied. In this case, at the next request of order N, the kernel will take a memory chunk from the freelist of order $N + 1$ and split it in half so that each half is of order N. The first half is returned to satisfy the request while the second half is added to the

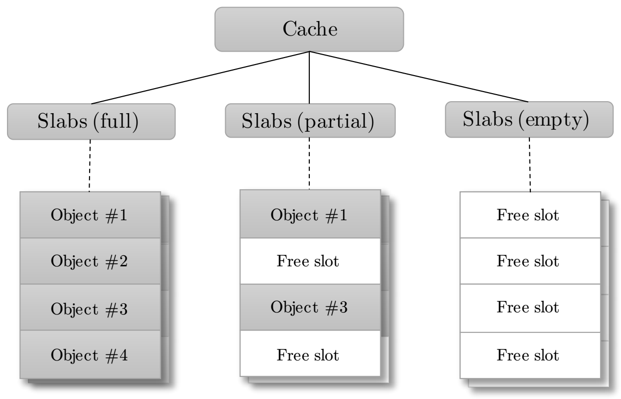


Figure 2.1 – Structure of a cache with an example of how a slab could be arranged. Here, each slab contains four slots adjacently allocated in memory.

freelist of order N.

Later, when the kernel releases that memory chunk, it simply puts the block back to the freelist of correct order. However, without additional care, the memory would eventually gets fragmented: the freelists of higher order would be exhausted as they would be split among smaller orders, and the allocator would not be able to serve requests for high order blocks. To prevent this scenario, the Buddy Allocator will try to *coalesce* the freed chunk of order N by looking for its second half in the freelist and recreate a block of order $N + 1$ if it is indeed present. It will then continue do the same thing with the chunk of order $N + 1$, and so on until the allocator cannot find a matching half anymore.

2.2.2 The SLAB Allocator

For smaller objects, typically when their sizes is less than a page size, allocating memory using the Buddy Allocator would be inefficient, as the minimum size of blocks it could return is equal to a page size, so a lot of space would be lost due to having slots that are too large compared to what is required[1]. Instead, the idea behind the SLAB Allocator is to group several objects of the same size on a contiguous memory chunk to avoid wasting space. A group of objects is called a *slab*, and each slab can be either full, partial, or empty: full slabs have all their slots allocated to objects, partial slabs have a mix of allocated and free slots and empty slabs have all their slots available. Also, several slabs together form a *cache*, and each cache can either be used to store a specific object type (*dedicated* cache) or be of general use and store a mix of different objects (*generic* slabs). For example, most of the structures in the kernel that are frequently allocated and freed have a dedicated cache, such as the structure describing a file on the system or the one storing the metadata of a process. Figure 2.1 depicts the structure of a hypothetical cache.

[1]This phenomem is called *internal fragmentation*, whereas the fragmentation problem presented in the previous section is called *external fragmentation*.

Now, when the kernel requests a slot from a cache, the SLAB Allocator first tries to find a slot in one of the partial slabs. If there is no partial slab left, it creates a new empty slab by requesting a memory chunk from the Buddy Allocator, splitting this chunk in several objects slots and returning one of them to fulfill the request. Later, if the kernel releases the object, the allocator checks if the slab which stored the object became empty and if it is the case, the associated memory chunk is released to the Buddy Allocator. To give a concrete example, the generic cache named `kmalloc-256` is used to serve every memory requests (not destined to a dedicated cache) which sizes are between 192 (not included) and 256 (included). Each slab of this cache uses a memory chunk of order 1 ($= 2^1 * PAGE\_SIZE = 8192$ bytes) so they each have $\frac{8192}{256} = 32$ slots available at creation. When all the slabs are full, the SLAB Allocator then requests memory chunks of order 1 to the Buddy Allocator. Note that `kmalloc-256` is only one example of generic slab; there exists also `kmalloc-32`, `kmalloc-64`, `kmalloc-512`, and so on.

2.3 Kernel exploitation

Now that some basic primitives of the kernel have been introduced, we will cover in this section the different steps that lead to a successful exploitation. We will first describe the origin of memory corruption bugs and the main way attackers achieve privilege escalation using them, followed by the mitigation that have been added to Linux to prevent kernel abuses. At this point, we will present the recently adopted protection called Control-Flow Integrity, as well as modern exploitation techniques that bypass it.

2.3.1 Memory corruption bugs in the kernel

On this subject, the kernel suffers from the same issues as user-space programs: since the C language does not provide memory safety, errors in the code can lead to the program corrupting its own memory. A corruption can first happen when *spatial safety* is not respected: an object (buffer, structure, ...) is accessed outside of its bounds, either for reading or writing data. This lead to bugs such as *heap out-of-bounds read*, *stack out-of-bounds write*, and so on depending on the location of the vulnerable object. This then results either in the use or the corruption of data in other objects. Errors can also happen when *temporal safety* is not respected: in this case, data are used at a moment in time which was not intended by the developer. Bugs that fall in this category are *use-after-free* (a pointer to a freed memory region is used), *uninitialized memory* (memory is not correctly initialized at allocation and used afterwards, which causes the program to use the data that were previously stored in that memory location), *double-free* (a memory region is freed twice), and so on.

```c
1 typedef int (* commit_creds)(struct cred*);
2 typedef struct cred* (* prepare_kernel_cred)(struct task_struct *);
3
4 void payload(void) {
5     struct cred *root_creds = ((prepare_kernel_cred)(PREPARE_KERNEL_CRED_ADDRESS↩
        ))(NULL);
6     ((commit_creds)(COMMIT_CREDS_ADDRESS))(root_creds);
7 }
```

Listing 2.2 – Example of a malicious payload: once run, the current process is promoted to root privileges

2.3.2 Control-Flow hijack attacks

Now if attackers trigger a memory corruption bug on a system without further preparation, the kernel will most likely overwrite random data and later crash when using them. However, by forcing the system to position sensitive data at the location of the corruption, they can take advantage of the bug to modify variables they should not have permission to access. To do this, the attackers can for example use the fact that the kernel allocates memory in a deterministic manner, as described in section 2.2: for example, due to the SLAB Allocator, several small objects of the same size or type can be allocated adjacently in memory if they are stored in the same slab. Then, if one of these object is vulnerable to an out-of-bounds write, the attackers can corrupt the objects next to the vulnerable one. Note that there exists methods which generalize to other allocators, object types or bug types: in chapter 3, we give more details on how attackers can shape memory according to their attack.

As described in section 2.1, function pointers are found in a lot of different kernel objects, so using a memory corruption bug to corrupt one of them is a common way to exploit the kernel. In this context, the attackers try to overwrite a function pointer to make it points to a memory location they control; then, when the kernel tries to call the function through the malicious pointer, the control-flow is redirected to the attacker payload which is then free to perform the attack. For example, here is how a typical control-flow hijack attack using a heap out-of-bounds write would take place in the kernel:

1. The attackers shape the kernel memory to have a victim object containing a function pointer adjacent to the vulnerable object.
2. Then, the bug is triggered and the pointer is replaced with a pointer to the attacker payload.
3. Later, the kernel calls the function so the payload of the attacker starts running.
4. Listing 2.2 shows a payload example: in the Linux kernel, each process has a *credential* objects with defines its privileges and stores metadata such as the ID of the user owning the process. Here, calling the `prepare_kernel_cred` function with a `NULL` argument (line 5) returns a newly allocated credential structure, which by default is set to the highest

(root) privileges. Then, the `commit_creds` function call (line 6) swaps the current credential structure of the process with the newly allocated one. Thus, once the payload finishes running, the attackers successfully elevates their privileges.

Now, several mitigations have been added to the Linux kernel to reduce the impact of such attacks. For example, Intel SMEP [19] (or equivalently ARM PXN [3]) restricts the kernel from executing code stored in user-space. If the kernel still tries to do so, it will crash. Also, the non-execute bit (AMD NX [2], Intel XD [19], ARM XN [3], ...) available on modern processors is used to restrain the attacker from executing code injected in the kernel memory regions storing data. Without this protection, the attackers would still have the possibility of overwriting a code pointer with a pointer to a data buffer that the kernel previously copied from user-space to kernel-space, for example while sending a packet through the network. The attackers now rely on more advanced techniques such as *returned-oriented programming* [31] or *jump-oriented programming* [5] that re-use portions of the kernel code to build the malicious payload. The attackers chain several small pieces of code (called *gadgets*) and each of them execute a small action, such as modifying the content of a register. An example attack could be a gadget chain that disables the non-execute bit on the system and then jump to a code buffer injected in kernel space. In order to make the process of creating a gadget chain harder for the attackers, the kernel implements *Kernel Address Space Randomization* (KASLR) [22], to randomize the location of the kernel code in memory. This way, at each boot the gadgets have different addresses and the payload cannot rely on hard-coded values as they will most probably point to completely different locations in the kernel memory. However, the effectiveness of KASLR in the kernel is often discussed and it is not clear how legitimate its existence is: attacks against this protection are regularly published, which for example bypass the system using its low entropy [16], transient execution [7] or information leaks [4]. Thus, recently a new mitigation called Control-Flow Integrity (CFI) is being adopted by the Linux kernel and aims at making the whole class of control-flow attacks more difficult to achieve.

2.4 Control-Flow Integrity

2.4.1 Presentation

The idea behind Control-Flow Integrity is to restrict the execution of a given program to a path in a *control-flow graph* (CFG) determined ahead of time, typically using a static analysis of the source code. Attackers that manage to hijack the control-flow can then only manipulate the execution within the bounds dictated by the CFG. To be effective, CFI has to protect *forward control-flow transfers*, such as indirect function calls, and *backward control-flow transfers*, such as jumping back to the return address saved during a function call. Abadi et al. [1] first described a CFI mechanism that limits the sets of targets allowed at each forward control-flow transfer point to the start of any function in the source code. With this design, attackers cannot jump to any bytes in the code anymore but only to the beginning of a function. For the backward edges, it implements a *shadow call stack* which stores copies of the return addresses in another stack hidden to the user. When a function returns, the return address is taken from the shadow stack instead of the standard stack so that the control-flow is not subverted, even if attackers managed to corrupt the standard stack. Note that the threat model they consider is very strong: their CFI enforces protection against powerful adversaries that have control over all the data memory of the program (i.e, not over the source code).

In subsequent works, the research community came up with numerous designs that improve different aspects of the original approach, such as the *precision* of the CFG, the performances of the checks or the compatibility with existing code bases. A powerful CFI scheme would enforce a perfect CFG (i.e, it contains only the paths that could exist in actual runs of the program) while inducing a negligent overhead and a high compatibility with existing program: this way, deploying the mechanism is doable with a reasonable amount of effort. This goal is currently not reached and current designs are making trade-offs among the different aspects to fit the usages one would consider for CFI: these trade-offs are highlighted in several papers [6, 40], which show the complexity of achieving the perfect design.

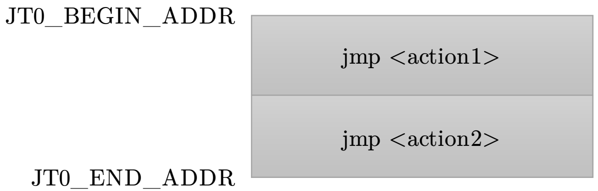
2.4.2 Implementation in the Linux Kernel

Now, although we covered CFI protections targeted to user-space software, some designs also support operating systems, which are harder to protected due to the inherent differences they have compared to traditional software: the Linux kernel, for example, have special control-flow constraints due to how interrupts work (see section section 2.1). In Linux, CFI is implemented using LLVM-CFI, which is part of the Clang compiler. The backward edges are also protected using a shadow stack called ShadowCallStack [32]: for example, on ARM the register `r18` is reserved to store the location of the alternate stack, and this pointer is never pushed to the main stack to prevent its value from being leaked to the attackers. The forward edges are protected by grouping the functions according to their prototypes (return type and argument(s) type(s)). Then, each indirect call is only allowed to jump to functions that are in the same group (or *class*) as the actual function pointer type declared in the code.

To do this, LLVM-CFI creates one *jump table* per function class, and each entry of these tables simply consists of a jump to one of the actual function address. Then, the compiler replaces every function pointers by addresses of entries in the corresponding jump tables. Finally, before jumping to a jump table entry, the kernel checks that the address at which it will go is indeed located inside the bounds of the jump table associated with the function class used at this call site. If it is not the case, the kernel crashes. This indeed enforces each indirect call to jump to the beginning of a function contained in the same jump table as the pointer type, which ensures that their prototype will match. Figure 2.2 shows a simplified example of how an indirect function call is checked and executed.

```
 1 int action1 (int arg1);
 2 int action2 (int arg1);
 3 double action3 (int arg1);
 4
 5 void demo(int arg) {
 6     int (*f) (int);
 7     (...)
 8     if (JT0_BEGIN_ADDR <= f < JT0_END_ADDR)
 9         f(arg);
10     else
11         panic();
12 }
```

(a) A basic indirect function call. The higlighted code on lines 8, 10 and 11 shows the check added by LLVM-CFI.



(b) Memory region containing the jump table for functions with prototype `int (*) (int)`.

Figure 2.2 – An example of how LLVM-CFI instrument the code to check the validity of the indirect function call. Figure 2.2a shows a code snippet augmented with the check and Figure 2.2b shows the associated jump table generated by the compiler. In this example, if the attackers manage to corrupt the function pointer before the call on line 8, they can only redirect the control-flow to either the function `action1` or `action2`, otherwise the kernel will panic.

Now, a concern that one can have regarding the way LLVM-CFI creates the function classes if that the size of each class can become large if the prototype is often used in the kernel. Indeed, some prototypes such as `void (*) (void)` or `int (*) (void)` are very basic and thus very common throughout the code. To give concrete metrics on how much the attack surface is reduced using CFI, in the Android kernel (which is based on Linux), 80% of the indirect calls site are restricted to less than 20 targets (i.e 20 functions or less are present in the jump table of the function prototype used at each call site), but 7% of the indirect calls have more than 100 targets available [12]. Although there exists techniques such as *control-flow bending* [8] to mount powerful control-flow hijack attacks that stays inside the CFG, in this work we do not focus on assessing the security of LLVM-CFI but rather the general effectiveness of a protection that would prevent all kind of control-flow hijack attacks. Thus, we perform our analysis by assuming that the control-flow of the kernel cannot be subverted at all.

Finally, note that CFI is currently not merged in the mainline Linux kernel but is already deployed in large-scale production operating systems, as it is for example enabled by default on Android starting from Google's Pixel 3 device [12]. At the time of writing, most of the required components for mainlining CFI support in the kernel have been merged [24], so it should not be long before we see CFI deployed by default.

```
 1 void spawn_privileged_shell(void) {
 2     int is_admin = 0;
 3     char username[32];
 4     char password[32];
 5 
 6     scanf("%s:%s", &username, &password);
 7     (...) // Perform permission check on username/password and set is_admin
 8     if (is_admin)
 9         execve("/bin/sh", NULL, NULL); // Spawn the shell
10 }
```

Listing 2.3 – This function is vulnerable to a data-only attack: when asking for the username and password, it does not correctly check the sizes of the inputs. Thus, when given a sufficently large username, `scanf` will write after the bounds of the `username` variable, which leads to a stack out-of-bounds write. Moreover, by writing past this buffer, the variable `is_admin` will get corrupted and later, at line 8, if this permission variable is set to anything different to 0, the program will spawn a shell. Thus, the attackers can have arbitrary code execution with this vulnerable software without knowing the administrator password and without hijacking the control-flow.

2.4.3 Data-Only Attacks

Although Control-Flow Integrity protects the software from the corruption of control data, it still leaves the remaining data used in the program unprotected. For example, local variables or data pointers can still be corrupted by an attacker without being detected by this mitigation. Then, depending on the software, these data might contain sensitive information such as access controls description, permissions or any variables that could let the attackers execute unwanted privileged actions if they get corrupted. An example of a code snippet vulnerable to a data-only privilege escalation attack is shown in Listing 2.3. Although the applicability of such technique on real-world application seems to be less likely than standard control-flow hijacking attacks, it has been showed [9] that many commonly used user-space application such as SSH, FTP or HTTP servers could be vulnerable to data-only attacks.

On the kernel side, these attacks are even more concerning due to the quantity of sensitive tasks it has to handle and the data related to them. Hence, assuming a threat model as strong as the one CFI designs in user-space do (i.e the attackers have control over the whole data memory) is not really applicable in the real-world as this would provide easy ways to bypass CFI. For example, Davi et al. [13] demonstrated a data-only attack that overwrites page table entries to later modify the kernel code in memory and remove the checks added by CFI, which effectively disable the mechanism completely. Indeed, the page tables are stored in kernel space and they take care of defining the memory permissions, such as which regions are readable, writable or executable. Tampering this data structure then allows attackers to define the memory region containing the kernel code as writable and modify it. In their work, this attack acts as a motivation to *PT-Rand*, a defense mechanism developed by the authors that protects page tables. However, it is not clear if protecting this data structure is enough to prevent attacks; for example, the attack we explained in subsection 2.3.2 can also be performed by this powerful attacker using a data-only attack: the credential structure of the current process is located by scanning the whole data memory space and corrupted in a way that simulates the calls to the `prepare_kernel_cred` and `commit_creds` function. This way, the attackers achieve the same privilege escalation without having to hijack the control-flow of the kernel, bypassing at the same time the CFI protections. With this in mind, some mitigations such as PrivGuard [29] identify several sensitive data structures in the kernel (including the credential structure in this case) and propose a protection to prevent the corruption of these important objects. However, this still does not answer the question on whether every sensitive structures have been found or if there is still room for attackers to corrupt data and gain privilege escalation. In other words, there is no proof that these workd have completely listed all the sensitive object in the kernel. Other works also tried to come up with more generic protections, for example by using *data-flow integrity*, which is similar to control-flow integrity but applied to data: a *data-flow graph* is built, and its integrity is preserved at run-time to ensure that attackers cannot corrupt other variables when a memory corruption vulnerability is present in the kernel. However, to our knowledge no such mechanism imposes an overhead low enough so that it could be implemented and used in practice in the kernel.

Similar to the present work, Federico Manuel Bento published in 2019 his thesis [14] in which he also did an analysis of the effectiveness of CFI in the kernel. Although the CFI scheme he analyzed is not LLVM-CFI but *Reuse Attack Protector* (RAP) from the PaX Team, the two schemes are very similar in essence (they both use prototype-based classes for the forward edges protections) and mostly differ on the implementation side. Moreover, he also assumes the end of code reuse attacks and presents different data-only attacks one could use to get privilege escalation in the kernel. To do this, he considers a powerful attacker (that has access to an arbitrary read and write primitive) and demonstrate the attacks using a kernel debugger to simulate such an attacker. However, it is not clear if the memory corruptions usually found in the kernel gives such capabilities to an attacker, which raises the question on whether having arbitrary control over the whole memory is a reasonable threat model. This observation motivates the use of real bugs and vulnerabilities for our analysis to make sure that the threat model we consider is compatible with real-world scenarios.

Chapter 3

Method

As stated in the introduction, we want to analyze how effective CFI in the Linux kernel is at blocking the attacks performed on memory corruption vulnerabilities. Our threat model is then a local attacker that seeks to elevate its privileges (disable security protections, get administrator access to the system, ...) by using such a bug. We assume the standard hardening features such as SMAP/SMEP, NX and so on are. However, we assume that KASLR is disabled as it can usually be bypassed using another vulnerability (see subsection 2.3.2). Now, as we want to base our analysis on concrete situations, we basically have two approaches to gather the necessary material: we can either use existing vulnerabilities that have a public exploit available check if CFI would prevent the attack, or collect kernel bugs and assess their exploitability with and without CFI. Here, we chose to do both methods: the first one is interesting as it gives the possibility to see how CFI cope with vulnerabilities which pose a proven threat as their exploitability is confirmed, however the number of existing public exploits on recent kernels is not sufficient to have a significant data set. Thus, the second method is necessary to have reliable result numbers, even if the work that has to be done is more important. In this chapter, we will then explain the method we followed to collect and analyze a large quantity of bugs present on a specific kernel version, as well as the different existing vulnerabilities we chose to work on.

3.1 Analyzing bugs in the Linux kernel

A technique which is very effective at finding bugs in large code bases is *fuzzing*: it consists of continuously sending (pseudo-)random inputs to a software until one of them hits a bug and crashes the program. Fuzzing does not require to understand the inner working of the test subject, as giving a description of the program interface is enough to start generating and feeding inputs. Otherwise, more conventional methods such as manually writing units tests are far more time-consuming. For our analyze, we use the *Syzkaller* [17] fuzzer as it is the state-of-the-art in Linux kernel fuzzing: currently, Syzkaller has reported more than 4000 bugs and 3000 of them are now fixed in the mainline kernel version [34]. Moreover, continuously running instances of Syzkaller are maintained by Google and can be consulted through a web interface called Syzbot [34], which is convenient in our case as we can directly use the bugs found by this system instead of having to run our own instance for a while.

For our analysis, we filtered the list of bugs that Syzbot outputs to only focus on the fixed bugs found in Linux 4.19[1]. Keeping only the fixed bugs allows us to remove the false positive from the list and to have access to the patch that fixed the bug, which is a precious help when trying to understand the bug more deeply. Also, restricting our data set to a specific Linux version ensures that the analysis is consistent throughout the bugs, as the set of mitigations and protections is not the same across the different Linux releases and the exploitability of a bug could then vary. Then, for each bug, we first check if it is vulnerable to a data-only attack: if it is the case, CFI cannot prevent the exploitation of this bug. If it is not the case, we check if it is vulnerable to a control-flow hijack attack: in this case, CFI is useful and would make the exploitation harder or unfeasible. Finally, if no attack is found, we classify the bug as unexploitable. It is also important to note that Syzkaller runs as a privileged process when fuzzing the kernel, so the bugs that it founds may not be triggerable by unprivileged attackers. Thus, we additionally classify each bug in one of the following category: **unprivileged** (any user on the machine can trigger the bug), **privileged setup** (some privileged steps such as mounting a specific file system are required but the steps that actually trigger the bug can be executed by any user) or **privileged** (every step requires higher privileges such as an administrator account). Note that privileged bugs can still be useful to attackers: for example, there are scenarios in which a user is administrator but does not have full access to the system, such as when seccomp [30] is restricting the number of system calls the user can actually use. Now, Syzkaller enables several debug options and sanitizers in the kernel configuration to help at finding more bugs, so the list contains different bug types which depend on the assertion that triggered the crash. For example, the *Kernel Address Sanitizer* (KASAN) [35] is enabled and track different memory errors such as use-after-frees or out-of-bounds writes. Thus, we present now the methodology we used to classify each bug type present in the list of bugs.

3.1.1 Heap out-of-bounds write

These bugs happen when the kernel writes outside of the bounds of a heap allocated object, such as a buffer or a structure. The main approach to exploit them is by manipulating the memory layout to overwrite sensitive data in neighboring structures. Thus, the two main steps to decide on their exploitability are to first find out the characteristics of the out-of-bound write, and then find a target in memory which is compatible with them. For the first step, the bugs are mainly found thanks to the KASAN sanitizer: when such an error is found, a report is produced by the tool with detailed information, such as the function call stack trace that lead to the out-of-bounds

[1]Note that as Syzbot is continuously finding new bugs, we used the list of bugs as it was the 14th of October 2020 at 6:02pm (GMT +2) so that we could work on a frozen data set.

write, the size of the written content, and so on. Using this report and any other helpful resources like the description of the patch that fixed the bug or the concerned kernel code hunk, we collect the *capabilities* of the bug, which are all the properties that characterize it. For an out-of-bounds write, these are essentially the offset at which the write begin (i.e, how far from the vulnerable object does is start), its length and its content. Most of the times these values are dynamic (they depend on the context) and we have to understand if the attackers can influence them or even control them completely. The more control the attackers have on the capabilities of the bug, the more powerful the resulting primitive is. Unfortunately, this information is not given by Syzkaller or the sanitizer, so the work has to be done manually by reading and analyzing the kernel source code.

We now have to determine if there exists a compatible target in memory: indeed, if the primitive is very restricted (for example the bug only allows to write one byte with value 0) the amount of targets that could be corrupted to get an interesting outcome will be very limited. To this end, we now give several examples of interesting objects present in the kernel heap that give a variety of choices depending on the primitive. They have either be found due to their common use in kernel exploitation, or by manually looking at the source code.

Credentials As presented in subsection 2.3.2, the credential structure (`struct cred`) stores the privilege level of the current process. The first interesting field is the ID of the user owning the process: overwriting this value with 0 raises the process to administrator privileges as `root`, the system account in Linux, has ID 0. The credential structure also stores the capabilities of the process, which define a subset of administrator privileges a process is allowed to have. For example, by maliciously adding the capability `CAP_SYS_MODULE` to the current process, an attacker can then load its own module in the kernel and execute arbitrary code in that environment.

Page tables Page tables are also stored in the heap, and they describe the properties (executable, writable and readable) of the address space for each process. Modifying this structure can then for example give the attackers the right to modify the kernel code and hence execute arbitrary code.

Process metadata In Linux, each process has a corresponding `struct task_struct` object allocated at its creation, which contains the metadata necessary to the process management. Therefore, it contains several fields that are interesting to corrupt: a pointer to the credential of the process that can be swapped with the credential of a process with more privileges, the limit between the user-space and kernel-space memory regions for this specific process (see subsection 4.2.1 for an example of how to abuse from this variable), or even data related to kernel mitigations (such as seccomp).

File system Most of the structures that deal with the underlying file system of the machine are stored in the heap: for example, the kernel maintain a cache of data chunks read from the disks (the *Page Cache*). By corrupting this cache, attackers can directly modify files as if they had write access to the file system, which they could use to modify privileged executable. Also, the kernel parses different sensitive information about the structure of the file system and stores them in different objects: one of them is the `struct inode` structure which contains the permission of each file (i.e who is allowed to read, write or execute it, if the file has to be executed on behalf of the `root` user, ...). There are also object that store directories structure, and so on. Again, tampering these metadata could let the attackers execute their malicious executable with administrator privileges.

If this is not enough, another possibility is to try improving the capabilities of the bug by creating another vulnerability in the kernel: for example, if a structure stores an array and its associated length at the same place, one can try to overwrite the size field to create an artificial out-of-bounds memory error. In the same idea, the Linux kernel uses references counting on some object, which allows it to keep track of the number of places the object is used. When the reference count drops to zero, the object is automatically freed. Thus, by tampering this field to force the object release, the attackers can create an artificial use-after-free later in the kernel execution. Another interesting possibility is to target linked-lists: these are frequently used in the kernel and by corrupting the pointers to the previous and next element in the list, an attacker can get a nice arbitrary write primitive in kernel memory. The subsection 4.2.3 gives a detailed example of how it is done in practice. Now, having arbitrary read or write in the kernel is very powerful as it allows reaching targets that are outside of the kernel heap. We can for example tamper sensitive variables in the global symbols of the kernel, stored at fixed locations in the data section (right after the kernel code in memory). We list here some examples of such targets:

User-mode helpers The kernel stores several paths pointing to user-space executable that are started with `root` privileges when it wants to delegate some choices to the system administrator: for example, when a file system with an unknown format is detected, the kernel executes one of these helpers which then has the mission of loading the correct module into the kernel to handle to new file system format. Thus, replacing the path to this helper with the path of a malicious executable gives the attacker a way to start its payload with high privileges. Examples of this attack are shown in subsection 4.2.1 and subsection 4.2.3.

Mitigations With access to the kernel data section, we can toggle the flags of some Linux security modules, such as *SELinux*, a protection mechanism that manages access controls to the applications, files or processes of a system. This module is configured by writing security policies and is used by system administrators to precisely define the level of security they want. Here, an attacker could disable this mitigation to perform unauthorized actions on the system.

Process metadata The `struct task_struct` of the very first process started on the system is stored as a global symbol. As every other processes on the system are children of this task and as the list of children `struct task_struct` is stored in this object, it is possible to navigate through the tree of processes to locate the attacker process and tamper the interesting fields described below.

```c
        struct type1 {
            struct type2 *layer1;
        };

        struct type2 {
            void (*action) (void);
        };

        void example(void) {
            struct type1 *vuln_object;
            (...) // vuln_object is incorrectly freed
            vuln_object->layer1->action(); // Use-after-free access
            (...)
        }
```

Listing 3.1 – An nested function pointer example. Here, `struct type2` is nested in `struct type1` by reference. Then, during the execution of the program, if a use-after-free error happen after the allocation of the object and the attackers can control the content of `vuln_object`, they can replace the `layer1` pointer with a pointer to a user-controller memory region that counterfeits a `struct type2` object with a malicious function pointer. By doing so, the attackers successfully hijack the control-flow..

If a target that matches the bug capabilities is found, we classify the bug as vulnerable to a data-only attack, thus bypassing CFI. Otherwise, we fall back at trying a control-flow hijack attack, so we do the same steps as before but this time looking for an object in memory containing a function pointer which can be reached by our primitive. Note that we also consider as exploitable function pointers that are indirectly stored in the vulnerable object, which means they are accessed through several nested structures (see Listing 3.1). Indeed, with techniques such as *ret2dir* [21], attackers can modify the pointer to the first nested structure and redirect it to a user-controlled memory region[2] in which they counterfeit the nested structures layers. This way, they can modify the function pointer at will. If we succeed in this research, we classify the bug as vulnerable to a control-flow hijacking attack, as recent researches show that the process of going from a control-flow hijacking primitive nto arbitrary code execution with privilege escalation can be automated [38]. Otherwise, we classify the bug as unexploitable.

Note that we do not take the step of manipulating the memory layout into account in our methodology. This step should indeed take place after we have found a target: the attackers have to make the right system calls so that the targeted victim object is placed right after the vulnerable object. However, as we previously explained, the heap memory allocation in the kernel follow a deterministic pattern that can be abused to shape the heap at will. As in similar work [10], we therefore assume that this step is almost always successful for a sufficiently motivated attacker. Later, when describing the different data-only attacks we implemented, we will show several advanced heap manipulation techniques that support this claim.

[2]Note that the user-space controlled memory region is not in user-space, as the SMAP mitigation prevents the kernel from accessing it. Instead, the ret2dir technique leverages the use of the *physmap* region, which is a portion of the address space that linearly maps the entire physical memory and is not subject to the same restriction as the address space mapping the user-space. We give more details on the physmap in subsection 4.2.1.

3.1.2 Heap out-of-bounds read

These bugs happen when the kernel reads outside of the bounds of a heap allocated object, such as a buffer or a structure. Unfortunately, it is hard to give a precise methodology for these cases, as the outcome of the bug will be highly dependent on the context in which it happens. Usually, exploiting an out-of-bound read is hard as the value will typically be used for a regular computation before being thrown away, but it could also be used in more sensitive operations: suppose there is a function in the kernel that reads pointers in an array and releases them one-by-one, which could be done by the cleaning function of a module when unloaded from the kernel. If this function is not careful enough and reads past the bounds of the array, it will read a value from a neighbor memory location and free it. Now, if an attacker manages to put an object whose first field is a pointer to another heap allocated object right after the vulnerable array, the function will free this object and create an artificial use-after-free, as the kernel did not expect that object to be released. For this reason, each heap out-of-bound read as to be manually examined using the kernel source code, the bug patch and Syzkaller debug outputs. If the bug can then translate to another primitive, the corresponding methodology is used.

3.1.3 Use-after-free

These bugs happen when the kernel uses a heap allocated object (buffer, structure, . . .) that was previously freed. Thus, the object was released at an incorrect point in time. Now, to analyze the exploitability of such a bug, we assume a standard exploitation methodology: when the vulnerable object is freed, the kernel memory is sprayed with objects whose content is user-controlled. The goal is to place one of these objects at the same memory spot the vulnerable object was allocated on. This leverages the fact that the SLAB Allocator and the Buddy Allocator both serves memory chunk with a *Last In, First Out* (LIFO) policy, as they both maintain freeelists to store the released memory hunks. Thus, by allocating objects of the same size as the vulnerable object, one of them will eventually land on the targeted memory spot. Now, finding an object of the good size for which the attacker can control the content is not a problem neither as universal spray techniques that accommodate to small and large objects have already been presented [27]. With this in mind, we then assume that the step or re-allocating an attacker controlled object on top of the vulnerable object almost always succeed for a sufficiently motivated attacker, and we therefore do not take it into account in our analysis. In other words, we assume that we could control the content of the vulnerable object as soon as the execution point at which Syzkaller reports the use-after-free is reached.

Then, we consider two scenarios: either the vulnerable object is of sensitive nature, for example if it is listed in the interesting objects presented in subsection 3.1.1. In this case, we assume the bug to be exploitable as tampering the content of the object would directly give a privilege escalation. If it is not the case, we look at improving the capabilities of our bug. Thus, in the instructions following the one that triggered the use-after-free, we look if any interesting primitive is present, such as the free of a pointer stored in the vulnerable object, a deletion in a list, a write to a data pointer, and so on as in the heap-out-of-bounds write methodology. Again, if we find one of these we then classify the bug using the methodology of the augmented bug. However, if we fail to do so, we assume the bug is not vulnerable to a data-only attack and we now have to decide if a control-flow hijack attack could be executed with this bug. To do so, we again start from the instruction that triggered the use-after-free and now look for a call to a function pointer stored (directly or nested in deeper layer) in the vulnerable object. Now, if we cannot find any function call on the path after the first use-after-free report, we estimate that the bug is not exploitable.

3.1.4 Double free

These bugs happen when the kernel frees an object (buffer, structure, ...) that was already freed earlier. Usually, this type of bug is exploited by quickly reallocating a *victim* object at the same memory location right after that the vulnerable object gets freed for the first time. This way, when the second free happens, the kernel will free the victim object and create an artificial user-after-free. The exploitation then follows the methodology of a use-after-free bug described previously (reallocation with an attacker-controlled object, and so on). In other words, a double free bugs gives an opportunity to choose an object on which we want to create an artificial use-after-free. This object has to meet some constraints as having the same size as the vulnerable object, but by the quantity of different objects that exist in the kernel and the variety of primitives this could in turn give, we assume that finding a victim object for which a data-only attack exists is almost always possible for a motivated attacker, and we therefore classify the double free bugs as exploitable with a data-only attack.

3.1.5 Warnings, debug information, deadlocks and others

Syzkaller also triggers reports for other reasons, such as assertions that are violated in the code, or deadlocks detection. For these more specific cases, it is necessary to manually inspect to code to see if a more complex bug is present. However, most of the time these bugs are not related to memory corruption errors and are classified as unexploitable. If it is not the case, the underlying bug is classified according to the appropriate methodology among the ones presented above.

3.2 Practical data-only exploitations

For the present thesis, we also implement three data-only attacks that use real bugs in the kernel. The first one is a bug taken from our classification: due to the huge amount of time it would take to actually create proof-of-concept attacks for each of the classified bugs from our list, we instead had to use the methodology we described above to do the classification in a reasonable amount of time. Thus, we implement one data-only attack from scratch to support the reliability of the analysis. In order to choose the bug we would build the attack for impartially, we followed the following steps: first, we sorted the list of exploitable bugs by the ID of the patch that fix them. Then, we selected the first bug from the **unprivileged** category that was not due to a race as our candidate for the attack. We restricted our choice to the **unprivileged** bugs to produce an exploit in which an unprivileged local attacker gains access to the *root* system account, as this kind of exploits are the most popular in kernel exploitation and therefore represent well the goals of a typical attacker. Also, we eliminate the bugs that require a race condition to be triggered as their inherent probabilistic nature make the exploit development time much larger, so for timing reasons we only keep the bugs that can be deterministically triggered.

The two other attacks we implement use existing vulnerabilities: we selected two public exploits that use control-flow hijacking to elevate the attacker privileges. We choose to work on bugs of different natures that represent the majority of kernel memory errors: a heap out-of-bounds write and a user-after-free. Also, we restricted our selection to exploits that have public write-ups available and are implemented on modern kernel versions. As the quantity of public exploits that correspond to these criteria is small, we found Google Project Zero exploitation of CVE-2017-7308 [4] and LEXFO exploitation of CVE-2017-11176 [26] to be good candidates. The idea is to revisit these two exploits to reach the same goal (take over the system with *root* privileges) but using data-only attacks in order to show the similar power of these techniques.

Chapter 4

Results

4.1 Classification

The complete list of bugs as extracted from Syzbot contains 102 bugs. Among these 102 bugs, we classify 86 bugs as being non-exploitable according to our methodology. On the remaining ones, 13 are classified as exploitable using a data-only attack, while 3 are classified as exploitable only with control-flow hijack techniques (see Table 4.1). Thus, over 16 exploitable bugs, we think that Control-Flow Integrity is effective against 3 of them. We also computed the size of the function class at the vulnerability point of each of these 3 bugs, which mean that assuming prototype-based CFI is used (as LLVM-CFI does), we look at the function call at which we think the attackers would be able to corrupt the pointer and see how many functions addresses could be substituted there according to the Control-Flow Graph. For the 3 bugs, we found classes sizes of respectively 1, 9, and 396. A class size of 1 basically means the function pointer used during the call is the only one allowed; however, a class size of 396 leaves a lot of attack surface to the attackers. We also computed different statistics, such as the distribution of bug types among the exploitable set (Table 4.2) and their required privilege level (Table 4.3). These results are further discussed in chapter 5.

Bug exploitability	**Attack type**	**Count**
Exploitable	Data-only	13
	Control-flow hijack only	3
Non-exploitable		86
Total		**102**

Table 4.1 – Distribution of attack types

Type	Count
Use-after-free	11
Doube-free	2
Heap out-of-bounds write	2
Heap out-of-bounds read	1

Table 4.2 – Distribution of bug types

Level	Count
Unprivileged	7
Privileged setup	5
Privileged	4

Table 4.3 – Distribution of privilege levels

4.2 Attacks implementation

We now describe the three different data-only attack we implemented. We first describe the one we created from scratch to give the full picture of a kernel exploitation example. We then explain how we used the Syzkaller outputs to have a minimal bug reproducer, how we improved its capabilities to get arbitrary read and write in the kernel and finally how we elevated our privileges. Then, we describe the two existing exploits we revisited by first giving an explanation about the bug as well, the exploitation method used by the original exploit authors and finally how we adapted the exploit to get to the same result but using data-only techniques.

4.2.1 End-to-end exploitation of bug 557d015f

The selected bug has ID 557d015f on Syzbot [20]. It is triggered on a Linux 4.19.108 kernel, but the bug is also present on other versions: the last production kernel vulnerable to this bug is Linux 5.5.13. Note that the attack we describe here has been implemented and tested on both versions, and to the best of our knowledge, no other public exploits for this bug were released at the time of writing. The bug is accessible to unprivileged users on kernels that permit *unprivileged namespaces*[1], which is the case for several major distributions, such as Ubuntu. Finally, we assume here that every common mitigations (including KASLR) are enabled.

Now, the fuzzer outputs several interesting information on the bug: the crash report, the links to the patch that fixes the bug and to the commit that introduced the bug in the kernel, and a reproducer (automatically generated in *syzlang*, the Syzkaller system calls description language, and in C). From the report, we know that the bug is an heap out-of-bounds write of size 16 bytes. More precisely, the victim is allocated with the SLAB allocator, so this is a slab out-of-bounds write. However, we will see later that depending on some parameters, the vulnerable object can also be forced to be allocated from the Buddy Allocator. The C reproducer is also a good starting point to begin understanding the bug, but unfortunately as Syzkaller automatically converts syzlang reproducer to its C equivalent, the program is not easily understandable for

[1]Namespaces is a Linux feature that allows giving to some processes their own isolated instance of a specific resource. For example, a user namespace allows creating an isolated environments in which a process has full privileges while ensuring it cannot impact the outside system. There, the process can see itself as root in the container, but cannot execute privileged action outside of this context.

```c
// Allocate memory
mmap(0x20000000ul, ...);
*(uint64_t*)0x20000280 = 0;
*(uint32_t*)0x20000288 = 0;
*(uint64_t*)0x20000290 = 0x20000180;
*(uint64_t*)0x20000180 = 0;
*(uint64_t*)0x20000188 = 0;
*(uint64_t*)0x20000298 = 1;
*(uint64_t*)0x200002a0 = 0;
*(uint64_t*)0x200002a8 = 0;
*(uint32_t*)0x200002b0 = 0;
// Send it to a netlink socket
sendmsg(socket, 0x20000280ul, 0ul);
```

(a) Syzkaller version

```c
struct msghdr hdr;
struct iovec io;
// Structure initialization
memset(&hdr, 0, sizeof(hdr));
memset(&io, 0, sizeof(io));
// Assign values
hdr.msg_iov = &io;
hdr.msg_iovlen = 1;
// Send through socket
sendmsg(socket, &hdr, 0)
```

(b) Enhanced version

Figure 4.1 – A transition from the raw reproducer to the simplified one: the two code snippets are equivalent, but the second one is much easier for humans to understand.

humans. For example, instead of allocating structures and assigning values to the field as any standard C source code, the Syzkaller reproducer allocates big memory chunks representing these structures and assign values by directly accessing the memory chunks at the rights offsets. Thus, the first step is to fall back to a more understandable reproducer by reverse engineering the different memory offsets and values to match real structure types and real arguments (such as recovering the flags names). An example conversion is shown in Figure 4.1. From this enhanced reproducer, we learn that the bug is triggered using Netlink sockets, a kernel interface that allows the exchanges of information between kernel-space and user-space. Processes can for example use this interface to configure or get indications on various network subsystems, such as routing tables or IPsec. Here, the socket is used to configure a network interface *queuing discipline*, which manage how the packets are routed to other layers. It is used for traffic controlling, for example to limit the throughput on a specific interface: this is done by adding a *filter* to the queuing discipline that defines rules to match specific packets using conditions based on packets attributes (such as the IPv4 packet header fields). Note that it is not required to understand the inner details of the system to do the exploitation: here, we take advantage of the fact that Syzkaller gives us an executable that triggers the bug to only understand how the attacker can influence the different parameters of the bug and use it to perform the attack.

Now, to trigger the bug, the reproducer first starts by creating a Netlink socket. Then, it creates a new queuing discipline on an existing interface with one particularity: it specifies a *hash table* size, which is normally used to improve the performance of the filtering. More precisely, without using a hash table, the kernel has to linearly check the packet against each filter until it finds one that matches. For a large set of filters, this becomes unfeasible, so using a hash table allows grouping the filters in buckets to reduce the lookup time. Once this is done, the reproducer updates the parameters of the discipline it defined at creation time and increases the size of the hash table. At this moment, an out-of-bounds write is reported. We now have to dig deeper in the kernel source code to find the properties (content, offset and size) of the write the attacker controls.

Finding the bug capabilities

By looking at the vulnerable function pointed by the Syzkaller report, we see that the memory corruption happens during the parameters change, in a function called `tcindex_set_parms`. In fact, this function is called twice: once when the discipline is set up and a second time when the parameters are updated using the Netlink socket. During its execution, the information related to the hash table is stored in a variable named `cp` of type `struct tcindex_data`, which has the following definition (an explanation of each field is given next to the field name):

```c
struct tcindex_data {
    struct tcindex_filter_result *perfect; // the hash table
    u32 hash; // the user-defined hash table size
    u32 alloc_hash; // the real (allocated) hash table size
    (...) // other irrelevant fields
};
```

At the beginning of the function, the size of the hash table supplied by the user is stored in `cp->hash`. Later, the hash table is allocated using that same size:

```c
cp->perfect = kcalloc(cp->hash, sizeof(struct tcindex_filter_result), ...);
```

In kernel-space, `kcalloc` is the equivalent of the user-space `calloc`, which means that the hash table is allocated as an array of `cp->hash` entries of type `struct tcindex_filter_result`. The behavior of this allocation function is the following: if the size of the requested array is less than $2 * PAGE\_SIZE = 8129$ bytes, it requests a memory chunk from the SLAB Allocator, which will then put the array in one of the generic slabs. Otherwise, for larger arrays, the allocation method will directly make its request to the Buddy Allocator by rounding the requested size up to the next memory chunk order.

Now, the problem here is that if the allocation succeeded, the `cp->alloc_hash` field should be updated with the newly allocated size. However, this is not done here, as it is shown by the patch that fixed the bug: the assignment `cp->alloc_hash = cp->hash` is missing after the hash table allocation. The `cp-alloc_hash` field will only be updated much later in the function, so during a significant period it is executed with the wrong information. When the function is run for the first time, this by chance does not cause any problem. Later, when the reproducer changes the discipline parameters, it updates the hash table with a smaller size. It also gives another parameter called `handle` whose importance will be apparent later. As before, the hash table is re-allocated with `kcalloc` without assigning the new size to `cp->alloc_hash`, which then still contains the old (bigger !) size. Later in the execution of the function, the out-of-bound is reported by the sanitizer. If we look at the faulty instruction, we find out that an assignment to one of the hash table entries is done:

```
(cp->perfect + handle)->res = cr
```

Here, the `handle` variable is the user-controlled parameter given when updating the discipline configuration. We see that if `handle` is set to a value larger than the size of the hash table (thus, larger than `cp->alloc_hash`), the resulting pointer will be out-of-bound. Normally, the kernel check the parameter validity before the assignment to prevent this scenario:

```
if (handle >= cp->alloc_hash)
    goto errout_alloc;
```

However, remember that `cp->alloc_hash` still contain here the size of the old hash table, which is bigger than the new one. Thus, as in this case the reproducer set `handle` such that `cp->hash < handle < cp->alloc_hash`, the check passes but the kernel will write into an out-of-bound entry ! This gives us a very nice versatile out-of-bounds write primitive: by changing the size of the first hash table allocation, we can choose the upper limit of the `handle` parameter. Also, by changing the size of the second hash table allocation, we can choose which allocator we target: either one of the generic slabs or the Buddy Allocator. Finally, with the `handle` parameter, we control how far from the beginning of the array we want to write; however, there are some constraints: notice that this parameter select an entry in the array, not a byte offset. Thus, we are restricted to choosing an offset which is a multiple of the size of an entry. In our case, an entry is 104 bytes wide (i.e, the size of a `struct tcindex_filter_result`). Also, the assignment writes to the `res` field, which is located at a 32 bytes offset inside the entry. In turn, we can begin our out-of-bounds write $104 * handle + 32$ bytes after the beginning of the vulnerable array.

Now, to understand what is the content of the write, we have to track down how the `cr` variable is defined. It is first initialized as an empty object (i.e, all fields are set to zero) of type `struct tcf_result`, which has the following definition:

```
struct tcf_result {
            unsigned long class;
            u32 classid;
};
```

This confirms the 16 bytes write size reported by the sanitizer: we indeed have 8 bytes for the `unsigned long`, 4 bytes for the `u32` (which is a 32-bits long unsigned type) and 4 bytes due to the padding added by the structure alignment. The `classid` field can be set by the user: in our case it is not done by the reproducer, but in the same way the hash table size and the `handle` are specified, the `classid` is one of the configuration parameters too. Without going into the details, when the discipline is created, an object called a *class* is created and attached to the discipline. It is possible to have multiple classes, so each class has a unique class ID, which is the parameters we can specify here. If we give the class ID of an existing class, the `tcindex_set_parms` will find that class and stores the pointer to that object in the `cr->class` field. Otherwise, the field is left untouched and stays null. As attackers, this is an important information as this increase the capabilities of our primitive: instead of having the 8 first bytes of the structure filled with null bytes, we can instead store a pointer in them. To do this, we need to find the class ID of the class instantiated at the discipline creation. By looking at their initialization code, we find out how the

ID is generated:

```c
int i = 0x8000;
u32 classid = 0x80000000;
do {
    classid += 0x10000;
    if (!exists(classid)) // a free classid is found
        break;
} while (--i > 0);
```

Thus, it simply starts from a fixed value and increase it with a constant increment until a free `classid` is found, so the pattern it follows is deterministic and we can deduce the class ID we target. Here, as the class of our discipline is the first created one, the loop does only one iteration. Hence, its `classid` is equal to `0x80010000`. Now, in our reproducer, if we set the `classid` to `0x800100000` too, the `cr->class` will be set to the pointer to the class, which will point somewhere in the kernel heap as the class is dynamically allocated. To summarize, we have two options for the write: we can either specifiy an arbitrary value in the `classid` parameter, in this case the write is composed of 8 null bytes, followed by 4 bytes of our arbitrary value and finally 4 null bytes again. Or we can set `classid` to `0x80010000` and in this case the write is composed of 8 bytes representing a pointer somewhere in the kernel heap, followed by 4 bytes containing the value `0x80010000` and finally 4 null bytes. Now that we have a better view of the bug capabilites, we have to find an appropriate target in the kernel memory.

Getting an (almost) arbitrary read and write primitive

In this exploit, we chose to target the `struct task_struct` object, more precisely its field `addr_limit`. As we mentioned earlier, the `struct task_struct` object stores per-process metadata such as the file it opened, its permissions in the system, and so on. But before explaining the reasons why we use this target, we first have to give a small reminder on the Linux kernel memory layout [28], as `addr_limit` plays an important role there. Note that in the following explanation, we assume a x86-64 architecture.

The pointers in Linux are not addresses that directly map the physical memory. Instead, they span a virtual address space that allows making use of the huge range of addresses offered by 64 bits pointers by dedicating parts of the virtual memory to specific purposes. For example, the lower part of the address space (`0x0000000000000000 - 0x00007FFFFFFFFFFF`) is reserved to the user-space, while the upper part (`0xFFFF800000000000 - 0xFFFFFFFFFFFFFFFF`) is used by the kernel-space. In this region, the kernels maps the different data it need: for example, the kernel code is accessible through the virtual hunk located at `0xFFFFFFFF80000000 - 0xFFFFFFFF9FFFFFFF`. Now, the *physmap* is another interesting subpart of the kernel memory region, as it linearly maps the whole available physical memory. It starts at address `0xFFFF888000000000` and ends at `0xFFFFC87FFFFFFFFF`, so accessing the address `0xFFFF888000000000` will directly points to the first **physical** address `0x0000000000000000`.

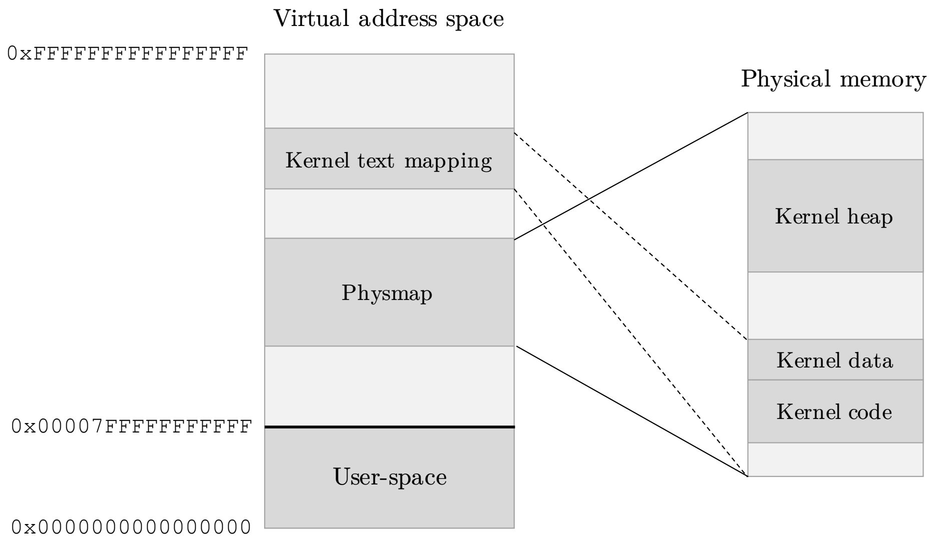


Figure 4.2 – Example of some interesting chunk in the kernel virtual address space and how they map to the physical memory. The bold limit between user-space and kernel-space is the value at which `addr_limit` is usually set. Note that the diagram is not at scale.

This is interesting as it means that some physical addresses have in fact two (alias) virtual addresses: for example, the kernel code can either be accessed through its direct mapping or through the physmap. In this context, the `addr_limit` field in the process metadata defines the limit between the user-space and the kernel-space, so by default it is set to `0x00007FFFFFFFFFFF` (Figure 4.2). This variable is then used by the kernel to verify if the user passes valid pointers to the system calls. More precisely, when the kernel has to exchange data with user-space, the user process has to give pointers to the data it wants to transmit to the kernel space. For example, when it wants to send data through a network socket, it gives the pointer to the data buffer to the kernel, which will copy the data to kernel-space before sending them to the network. In the opposite direction, when the process reads data from the network socket, it gives a pointer to an empty buffer in user-space and the kernel will write the data received from the network in that buffer. In each operation, the kernel verifies using `addr_limit` that the pointers given by the user process are indeed in user-space, otherwise it could give pointers in kernel-space, and the kernel would leak/overwrite data in its own memory region. Now, we could wonder why this limit is a per-process variable instead of being fixed, as the security impact is high. In fact, in often arises that the kernel calls a function that is intended to be used on user-space data, but invoked here on kernel-space addresses. For example, when the kernel itself wants to read from a file, the kernel uses the same `read` function it would use when reading a file for a user space-process. However, the checks are still present: as the function is intended to copy the file content into a user-space buffer, invoking it with a kernel-space destination buffer triggers an error. Therefore, in these cases, the kernel disable the check by raising temporarily the `addr_limit` variable.

Now, recall that one of the value written by our out-of-bounds primitive is a pointer somewhere in the kernel heap. Also, in the **physical** memory, the memory chunk allocated for the heap is stored after the kernel code and its static data (see Figure 4.2). Thus, if we manage to overwrite `addr_limit` with this pointer, we will be able to invoke system calls with pointers to the kernel code and data without triggering the sanity checks ! This is very powerful: this lets us read and write arbitrary data to a part of the kernel memory.

Now, assuming that the `addr_limit` is indeed modified, how do we actually perform those reads and writes ? Earlier, we mentioned that the kernel exchange data with user processes when they want to communicate with the network, so this could be one usable primitive. However, a similar but simpler alternative is to use a *pipe*. In the Linux kernel, pipes are used for (unidirectional) interprocesses communication: the data written on one end of the pipe can be read from the other end of the pipe. Under the hood, the kernel acts as the intermediary: it reads the data written into pipe and copy them in the kernel space. Then, when a process reads data from the pipe, the kernel copies the data to the user process buffer. A standard use of pipes could look as follows:

```c
int ends[2]; // The pipe file descriptors
char data[] = "hello"; // The message to transmit
pipe(ends);
// From now on, ends[1] is for writing and ends[0] for reading
write(ends[1], data, 6);

// At the other end, the process allocates a buffer and reads from the pipe
char buf[6];
read(ends[0], buf, 6);
// From now on, buf contains the string "hello"
```

Here, at the beginning of the `write` system call, the kernel checks that `&data < addr_limit`. A similar check is performed for `read`. Thus, if `addr_limit` is increased due to the corruption, a process can leak kernel memory with the following construction (note that the address of the buffer the kernel has to read from is replaced by a kernel address):

```c
int ends[2];
pipe(ends);
write(ends[1], KERNEL_MEMORY_ADDRESS, 32);
// The kernel copies 32 bytes from KERNEL_MEMORY_ADDRESS to a temporary buffer
char buf[32];
read(ends[0], buf, 32); // When the process reads from the pipe, it leaks kernel↩
    data
```

Similarly, a process can write data to a kernel address (note that here, the address of the buffer the kernel has to read to is replaced by a kernel address):

```c
int ends[2];
char payload[32] = "MALICIOUS PAYLOAD";
pipe(ends);
write(ends[1], payload, 32);
// The kernel copies the payload to a temporary buffer
read(ends[0], KERNEL_MEMORY_ADDRESS , 32); // When the process reads from the pipe,↩
    the kernel will write the 32 bytes to the kernel address
```

With these two primitives, we will be able to overwrite kernel static data to elevate our privileges. But first, we need to actually overwrite the `addr_limit` of a process.

Heap memory shaping

To do this, we need to shape the kernel heap so that the `struct task_struct` of a process we control is adjacent to an array vulnerable to the out-of-bounds bug. In this way, with the correct parameters, the out-of-bounds write will reach the `addr_limit` field of the structure and corrupt it. Now, a `struct task_struct` is allocated each time a new process is created: the kernel stores them in a dedicated cache whose slabs contain 8 slots each. As the structure is 3776 bytes wide, the slabs use memory chunks of order 3 (as $2^3 * 4096 = 32768$ bytes is the nearest size of memory chunks that can hold $8 * 3776 = 30208$ bytes). For its part, recall that the vulnerable array has a variable size that we control. For example, by choosing a hash table of 315 entries, the kernel will allocate $104 * 315 = 32760$ bytes (i.e, the size of an entry times 315) using `kmalloc`, which in turn will ask the Buddy Allocator for a chunk of order 3, as this is the closest memory chunk size that can store that many bytes. Now, we could think about the following idea to get the two allocations side by side: we want to leverage the fact that the Buddy Allocator can split memory chunks in half to serve requests, which ensures that they are adjacent in memory. Thus, we first want to empty the Buddy Allocator freelist of order 3, by forcing the kernel to allocate a lot of structures of that size. Then, we trigger the allocation of the vulnerable array, and as the freelist of order 3 is empty, the Buddy Allocator will split a chunk of order 4 in two and return the first one, which will then store the vulnerable table. Finally, we create a lot of new process to force the allocation of several `struct task_struct`: their dedicated cache will eventually be full and will request a new hunk of order 3 to the Buddy Allocator, which will return the second half of the previous order 4 chunk. The new slab is then stored on that chunk. Thus, at the end of the operation, the slab is adjacent to the vulnerable array. It just remains to trigger the out-of-bounds bug to overwrite one of the neighboring `struct task_struct`.

However, this method requires us to solve two small difficulties: first, recall that we have a constraint on the offset at which we can write our pointer: it starts $104 * index + 32$ bytes after the beginning of the vulnerable array, with $index$ being attacker-controlled. Here, the field that we want to target inside the `struct task_struct`, `addr_field`, is located at a 2840 bytes offset inside the structure. Thus, the correct arguments to our out-of-bound primitive are computed by solving the following equation for $index$ and i: $104 * index + 32 = 2^3 * 4096 + i * 3776 + 2840, i \in [0, 7]$. Indeed, we want our write to first skip the whole vulnerable array $(+2^3 * 4096)$ and then overwrite the `addr_limit field` ($+2840$) of one of the eight `struct task_struct` stored in the slab ($+i * 3776$). The solution to that equation is $index = 451, i = 3$, so we will overwrite the fourth `struct task_struct` prsent in the victim slab. Thus, to summarize, the parameters given through the Netlink are the following: for the first hash table size, we give a huge value so that are out-of-bounds offset is not bounded, so for example 8000 is fine. For the second hash table size, we want to target the same chunk size order as the `struct task_struct` slabs, so we set it to 315. Finally, we set the `handle` parameter to 451 ($= index$) as it defines the out-of-bounds entry at which the write is done.

The second difficulty resides in the fact that in our method, we first allocate the vulnerable array, then the new `struct task_struct` slab and finally we trigger the bug. However, this is not feasible with our bug: recall that the out-of-bound write is done right after the table is allocated, so we do not have enough time in-between to allocate the new slab. The trick we will use here is to first allocate a placeholder object that will save the memory spot in which we want the vulnerable array to be. Thus, for a moment, we will have the placeholder object adjacent to the `struct task_struct` slab in memory. Then, we will quickly release the placeholder object and allocate the vulnerable array. As the Buddy Allocator serves the memory chunks with a LIFO policy, we will get back the same memory spot and our vulnerable array will be located right before the victim slab. This way, when the out-of-bounds write is triggered, the memory is already correctly shaped. Note that for our method we need to force the kernel allocating objects with a specific size: first when we exhaust the Buddy Allocator freelist and then when allocate the placeholder object. We then need to find how to allocate an object in the kernel with that matches the following constraints: 1) the object can be easily allocated and freed using two separates system calls, this way we can choose the lifetime of the allocation, 2) the other objects allocated by the system calls (called *side effects*) should be as few as possible to prevent influencing the memory layout and 3) the number of distinct allocations we can force is not or very slightly bounded, so we do not have a limit when spraying the memory with a lot of objects. Here, the objects we choose to use are networks sockets with *ring buffers*[2], as they give a nice primitive to allocate and free memory chunks at will, with the sizes we decide. Now, to summarize, here are the main steps of the heap memory shaping:

1. We create a lot of ring buffers whose sizes correspond to a memory chunk of order 3. This drains the Buddy Allocator freelist of order 3 until it gets empty.
2. We create one more ring buffer, called the placeholder object, which is also allocated using a memory chunk of order 3. As the corresponding freelist is empty, the allocator breaks a memory chunk of order 4 in half and the first part is used to store the dummy object. The other half is stored in the freelist of order 3.

[2]RIng buffers are an optimization that allows reading and writing to a socket faster than the traditional way: instead of using the `read` and `write` system calls, the user process can exchange data using a buffer (the ring buffer) shared between kernel-space and user-space. At the creation of the socket, the users choose the size of the ring buffer they want to use, and the kernel allocates it through the Buddy Allocator. When the socket is closed, the memory chunk is released.

3. We then create a lot of new processes: for each newly created process, the kernel requests a slot from the cache storing the `struct task_struct` structures. After a while, all the free slots are used and the cache has to create a new slab: it requests a new memory chunk of order 3, which is served from the freelist of order 3 and corresponds to the second half of the memory chunk of order 4 split earlier. At this point, the placeholder object is adjacent to the newly created slab.
4. We release the placeholder object by closing the corresponding packet socket. The memory chunk used to store the ring buffer is placed in the freelist of order 3.
5. We now trigger the bug: the vulnerable array (the hash table) is allocated, so the kernel requests a memory chunk of order 3 and the allocator returns the one we just released. At this point, the vulnerable array is adjacent to the `struct task_struct` slab in memory. Right after, the out-of-bounds write is triggered, and the `addr_limit` field of one of the process is overwritten.

Privileges escalation

Now, recall that with the tampered `addr_limit`, we have access to the kernel data region in memory. We then have to find a global variable that will give us a way to increase our privileges. Here, we choose to tamper the *core pattern* kernel parameter: in Linux, it is possible to send signals to processes to trigger specific actions. For example, one can pause or terminate a process by sending to corresponding signal. Also, some signals cause the process to exit and produce a *core dump*, which is a file containing the image of the process memory. This is useful for developers who wish to debug crashes in their program: this image can be loaded in a debugger such as `gdb` [15] to inspect the state of the program when it received the signal. For example, one such signal is `SIGSEGV`, sent when a process access to an invalid memory address. By default, the name of the produced file is `core`, but system administrators can change this behavior by changing the core pattern parameter, which is in practice done by writing the filename they want in the file `/proc/sys/kernel/core_pattern`. In addition, this filename can contain different specifiers, for example a name such as `core-%t` appends the date of the crash to the filename. Core pattern also support piping the resulting memory image directly to an executable: in this case, the administrator can specify a name such as `|/path/to/exec` and when a crash occurs, the kernel executes the `exec` programs and transmits the whole process memory image to this new instance. The important detail here is that the kernel starts the executable under the `root` account, so the process has full privileges on the system. Our goal is then to overwrite this core pattern parameter and set it so that core dumps are piped to a malicious executable we created. This way, the kernel will start on instance of our program with full privileges.

Of course, we cannot write directly into `/proc/sys/kernel/core_pattern`, as this action is restricted to privileged users. Instead, we have to corrupt its value in kernel memory. In fact, this file is not a traditional file, it is a *pseudo* file created at boot by the kernel: this means that when a user writes or reads from this file, the kernel intercepts the calls and takes the appropriate actions. In this case, when writing to /proc/sys/kernel/core_pattern, the kernel stores the value to a global variable call core_pattern. We can easily find the address of this variable by reading the /boot/System.map file, which lists every symbol in the kernel (functions, global variables, ...) with their respective address.

```
$ cat /boot/System.map | grep core_pattern
ffffffff82b21800 D core_pattern
```

As these data are statically allocated, their location inside the kernel text will not change across reboots. However, note that this address is part of the virtual address space slots that maps the kernel text (Figure 4.2); however with our corrupted addr_limit we can only reach part of the physmap. This address is then in an inaccessible spot for now. Thus, we have to compute the alias address of core_pattern in the physmap: we first compute its offset from the beginning of the physical memory (0xFFFFFFFF82B21800 - 0xFFFFFFFF80000000 = 0x2B21800). As the physmap maps the entire physical memory, we then just have to add this offset to the address of the beginning of the physmap (0x2B21800 + 0xFFFF888000000000 = 0xFFFF888002B21800). We can now build an arbitrary write using the pipe technique described earlier and overwrite the memory region at 0xFFFF888002B21800 with |/tmp/payload. We then place our malicious executable in the /tmp folder, creates a new dummy process and send the SIGSEGV to it. At this moment, the kernel starts our malicious payload with root privileges.

Breaking KASLR

Now, on modern kernels, the address of the core_pattern symbol is in fact not fixed across reboot as Linux implements KASLR, which randomizes the location of different elements in the kernel memory at boot time. In total, it randomizes six different memory regions location [37], but in our case only two of them impact our attack: the first one is the location of the kernel text in the **physical** memory. Normally, the kernel text is loaded at address 0x1000000, but KASLR adds a 2MB-aligned offset to the load address. The second one is the beginning of the physmap: here, KASLR adds a 1GB-aligned offset [25]. However, by building an arbitrary read using the pipe technique, finding the two offsets is feasible: first, we compute the physmap offset by probing every address starting from 0xFFFF888000000000 with an increment of 0x40000000 (1GB). Each address probed before reaching the physmap is not mapped in the virtual address space: thus, the read call fails when trying to read from one of these. At the first successful read, we then hit the first mapped address from the physmap, so we have found the first offset (PHYSMAP_OFFSET). Then, in order to find the KASLR offset of the kernel image, we continue to probe from this address but this time using an increment of 0x200000 (2MB). Here, all the addresses are mapped in memory, so instead we read a few bytes at each read call, and we check if the bytes correspond to the bytes of the beginning of the kernel code (note that these bytes can be computed beforehand by looking at the binary kernel image). Once we have found a matching sequence, we have the second offset (KERNEL_OFFSET). The final address of the core_pattern symbol then becomes

0xFFFF888002B21800 + PHYSMAP_OFFSET + KERNEL_OFFSET and we can overwrite it as described above.

4.2.2 Revisiting CVE-2017-7308

This bug was present in the kernel since 2011. It has not been found (at least publicly) until 2017 when Andrey Konovalov, a member of the Google Project Zero security team, triggered it while fuzzing the kernel using Syzkaller. The author provides an extensive explanation of the bug, as well as the exploit he wrote to gain *root* privileges on a Linux 4.8.0 kernel with all common mitigations enabled, and unprivileged namespaces authorized. The bug is due to a signedness issue, which in turn leads to a heap buffer out-of-bounds write. More precisely, the bug is located in the depths of the code that takes care of AF_PACKET sockets, which allow users to interact directly with the packets at the device driver level. With these, users can implement their own protocol directly on top of the physical layer. These sockets can also use ring buffers (described in subsection 4.2.1) to speed up the exchange of data between user-space and kernel-space. However, the check that verifies if the packet data can fit in the buffer is not correctly done: the kernel will then write data past the buffer and overwrites neighbor objects. Also, the attacker has a lot of control on the different properties of the bug: its complete capabilities are a heap out-of-bounds write of attacker-controlled length and offset (up to 64KB). Also, the attacker controls most of the content of the write as the overflown data is the content of the packet: the only uncontrolled part is a 14 bytes header at the top.

Now, in the original exploit, the author shapes the heap similarly to the method described in Listing 2.3, although here he does not target a slab filled with struct task_struct objects but rather with struct packet_sock, the structure that contains the description of a packet socket. He targets three fields in that structure:

```c
// Only the important fields are displayed here
struct packet_sock {
    struct packet_ring_buffer {
        struct tpacket_kbdq_core {
            struct timer_list {
                void (*function)(unsigned long);
                unsigned long data;
            } retire_blk_timer;
        } prb_bdqc;
    } rx_ring;
    int (*xmit)(struct sk_buff *);
}
```

The two first ones, function and data, are used by a timer the user can set on a packet socket. Whenever the timer expires, the kernel calls function(data), so this gives a nice function call primitive with controlled arguments. In the exploit, this primitive is used to disable SMAP and SMEP, the two mitigations that prevents the kernel from accessing or executing data in user-space. To disable them, the author has to tamper the CR4 control register, which contains the switches of different CPU features. In our case, the bits 20 and 21 have to be set to 0 to disable the two protections. To do this, the function field is replaced with the address of the native_write_cr4 function, used by the kernel to change the value of the CR4 register. Then, the data field is overwritten with the new tampered register value. This way, once the timer retires, the protections are disabled. Then, the exploit uses the third field, xmit, to call a malicious payload in user-space. Normally, this function is called when a packet is transmitted to the network: here, the pointer is replaced by the address of the attacker-allocated buffer in user-space containing the payload described in subsection 2.3.2. Thus, when the attacker sends a packet through the socket, the kernel will in fact jump to the attacker code and replace the credentials of the current process with full-privileged credentials.

Now, this attack would be blocked if CFI were enabled on the kernel, as substituting a kernel function pointer with a buffer in user-space is not permitted by the Control-Flow Graph computed by a LLVM-CFI. In the data-only exploit we implement, we therefore change the target: instead of corrupting a struct packet_sock object, we again corrupt a struct task_struct. This time, we focus on its cred field, which stores the credential of the process:

```
struct task_struct {
    (...)
    struct cred  *cred;
    (...)
}
```

We see that the structure does not directly embed the credential data, instead only a reference to them is stored. When a new process is created, a cred structure is allocated and filled with the right permissions: in our case, it contains the permissions of an unprivileged user, so our goal here is to replace the pointer with one that corresponds to a privileged process. Usually, these structures are dynamically allocated, so leaking the pointer to one of them is not straightforward. However, there is an exception: in Linux, the first process to start is called the init task and the other processes are just children of this task. Moreover, this task is not dynamically allocated: its struct task_struct is a static variable. In the same way, its credential structure is also a kernel symbol and its address can then be printed using the Symbol.map:

```
$ cat /boot/System.map | grep init_cred
ffffffff82a632c0 D init_cred
```

As the init task is a privileged process, we can use the out-of-bounds write to overwrite the *cred in an adjacent struct task_struct slab with the value 0xFFFFFFFF82A632C0, and we then get a privileged process.

Note that we assumed here that KASLR was disabled, as the original exploit relies on another information leak vulnerability to find the KASLR offset. However, we could also bypass this mitigation by using this bug and a data-only attack. Indeed, in the structure corrupted by the original exploit, we can find another interesting field:

```
struct packet_sock {
    (...)
    struct packet_rollover *rollover;
    (...)
}
```

which is pointer to a structure that has the following definition:

```
struct packet_rollover {
    (...)
    long num;
    long num_huge;
    long num_failed;
    (...)
} ;
```

Now, there exists a system call called `getsockopt()` which can be called on a packet socket to retrieve different informations about it. One kind of information is the "rollover stats", which when queried are read from the `num`, `num_huge` and `num_failed` field through the `packet_rollover` pointer stored in the packet socket. Thus, we can overwrite the `rollover` pointer using our out-of-bounds write with a malicious kernel pointer and then execute a `getsockopt` call. At this moment, the kernel reads three `long` values from the malicious pointer and return them to the process, which leaks the kernel memory. Hence, we have created an arbitrary read primitive. At this point, KASLR can be bypassed with the same technique as described in subsection 4.2.1.

4.2.3 Revisiting CVE-2017-11176

The second exploit we reworked with a data-only attack uses a bug present on Linux kernels up to version 4.11.9. Although he is not the author of the CVE discovery, we based our work on the public exploit of Nicolas Fabretti, from the LEXFO security cabinet. Here, the original exploit assumes that SMAP, the mitigation that prevents the kernel from reading data stored in user-space, is disabled. However, although KASLR is also assumed to be off, in our exploit we assume that both mitigations are enabled. This time, the vulnerability is a used-after-free present in the code managing the Netlink sockets. More precisely, with a race condition one of the main structure used in this subsystem gets its reference count decreased once too often, which cause the kernel to free the object. However, the structure is used afterwards, leading to a memory error. The vulnerable object is of type `netlink_sock`, a huge structure containing a lot of different fields, so finding a potential target is not difficult. Here the author chose to corrupt the following field:

```
struct netlink_sock {
    (...)
    wait_queue_head_t wait;
    (...)
} ;
```

This field is the head of a *linked-list*, which is a common data structure in the kernel used to create lists. The head is simply a pointer to the first element contained in the list. Here, this list contains elements of the following type:

```
struct __wait_queue {
    unsigned int flags;
    void *private;
    int (*func) (struct wait_queue_entry *, unsigned, int, void *);
    struct list_head task_list;
};
```

Note that the third field is a function pointer, selected as the target in the original exploit. It is not the only one present in the structure, but the author chose this one as it is easy to trigger a call to that function from userspace (using one system call) and has few side effects, which is important in order to have a reliable exploit as the vulnerable structure is in a corrupted state caused by the bug. Now, to be able to hijack the control-flow, the author first has to re-allocate the freed object with an object whose content is under his control. One of the traditional way of doing this is by using the `sendmsg()` system call, which is used to send a message through a socket: it takes as argument a buffer with user-controlled length and content which is later copied into a dynamically allocated buffer in kernel-space. Note that both `netlink_sock` and the `sendmsg` buffer are allocated using `kmalloc`, which means that if we set the `sendmsg` buffer size to the length of the `netlink_sock` structure, they will be allocated from the same generic cache. Thus, the re-allocation goes as follows: the attacker triggers the bug, so one `netlink_sock` structure is wrongly released. Here, the attacker quickly calls `sendmsg` with a buffer whose content mimics the vulnerable structure but with malicious values at the targeted fields. This way, the next time the object will be used, the kernel will read the corrupted values.

Now, the attacker leverages the fact that SMAP is disabled to set the linked-list head pointer in the victim `netlink_sock` to a faked list entry allocated in user-space. This way, it is very easy for him to change and update its content, including the `func` function pointer shown above. He can then use the system call that triggers the function call to take over the control-flow. Once the execution flow is controlled, it executes a complicated privilege escalation exploitation using return-oriented programming: basically, it forces the kernel to set its stack pointer to a malicious fake stack in user-land created by the attacker. Then, it jumps to small snippets of code in the kernel text which all ends with a `return` instruction, this way at the end of each snippet the kernel pops an address from the fake stack and jumps to the next gadget. The gadget chain is built such that at the end of the execution, the CR4 register has its SMEP bit cleared. Then, as in subsection 4.2.2, it jumps to a payload that executes `commit_creds(prepare_kernel_cred(0))`, which gives root privileges to the current process.

Again, CFI would be effective on this attack, as jumping to an arbitrary byte which is not the beginning of a function in the kernel code is not allowed by the Control-Flow Graph. However, we can find other interesting field in the `netlink_sock` structure to adapt the exploit with a data-only attack, such as this linked-list entry:

```c
struct netlink_sock {
    struct sock {
        struct hlist_node {
            struct hlist_node *next;
            struct hlist_node **pprev;
        } sk_bind_node;
    } sk;
}
```

The `sk_bind_node` field stores the linked-list metadata of an entry. Here, the `next` field points to the next entry while `pprev` stores a pointer to the previous element, so to be more precise we have here an entry of a doubly linked-list. Also, by looking at the source code that accesses to the fields of the victim object, we found that by giving the right parameters to the `setsockopt`[3] system call, we can trigger the deletion of this node from the linked-list. Now, in order to simplify some procedures in the list manipulation, note that `pprev` is not a direct pointer to the previous element but rather a pointer to the location of the previous element pointer. With this optimization, the function that removes an element from a list is implemented as following:

```c
1 void hlist_del(struct hlist_node *n)
2 {
3     struct hlist_node *next = n->next;
4     struct hlist_node **pprev = n->pprev;
5
6     *pprev = next;
7     if (next)
8         next->pprev = pprev;
9 }
```

But recall that when this function is called on the node, we control the content of the structure pointed by `n`, as it is called on the victim object field. Thus, if we set `pprev` to an arbitrary kernel address and `next` to the content we want to write, when executing `*pprev = next` (line 6) the kernel will write our value to the desired memory location. This list deletion can then be used as an arbitrary write of length 8 bytes. Using this primitive, we can for example execute the same privilege escalation technique as in subsection 4.2.3 and overwrite the `core_pattern` symbol. Also, the string `"|tmp/a"` fits in 8 bytes (including the terminating character), so one write is enough to overwrite the symbol with a path an unprivileged user has access to. Here, converting the string to a pointer type gives the value `0x00612F706D747C` (note that the bytes order is reversed due to the endianness). There is one subtlety though: the last instruction in the list deletion procedure executes the assignment `next->pprev = pprev` (line 8). However, in our case `next` is set to the path we want to write, so interpreting this string as a pointer creates an invalid address. Thus, writing here triggers a memory corruption (called a *kernel oops*) and our process is killed as the kernel does not now how to recover from the error. To circumvent this behavior, we split the write in two, so we can control the upper part of the resulting address and choose a memory region mapped in the kernel address space. For example, instead of writing `0x00612F706D747C`

[3]This system call is similar to the `getsockopt` one, except it is used to modify the socket parameters instead of reading them.

at &core_pattern, we write 0xFFFF8800706D747C at &core_pattern and 0xFFFF88000000612F 4 bytes further, at &core_pattern + 4 bytes. Now, each address points inside of a memory region mapped by the kernel [28] so accessing it is legal. Also, it might overwrite other random data, but the probability to corrupt data that crashes the kernel is very low.

Note that we can also bypass KASLR with this vulnerability, using the following field:

```
struct netlink_sock {
    (..)
    unsigned long *groups;
    (...)
}
```

Here, we can access this field using the system call getsockname. It reads the value at groups[0] and returns it to the user. Thus, be setting groups to a kernel memory address, we can build an arbitrary read of length 8 bytes. Then, we use the same technique as in subsection 4.2.1 to bypass KASLR by probing the address space to find the offset. However, note that during the attack with the pipe technique, probing an invalid address just returns an error as the address is user-supplied: to prevent badly programmed software form crashing, the kernel does not halt. Instead, here the read happens directly from kernel space, so the behavior is different and two cases are possible: if the CONFIG_PANIC_ON_OOPS flag is enabled in the kernel configuration, the kernel crashes when probing invalid address. I this case, our KASLR bypass does not work. However, this flag is rarely enabled in major distributions, as they try to recover from these problems to avoid causing denial-of-service on the system; in this case the kernel simply halts the process. Thus, before each probe, we just have to fork the current process and execute the read from there: this way, if the address is not mapped, the process crashes and the exploitation continue in the main thread.

Chapter 5

Discussion

In this chapter, we discuss different aspects of the thesis: we explain some limitations of the methodology we used, highlight some interesting results in the classification and present mitigations that could prevent the data-exploit attacks we built. We also discuss on future works which could improve our classification.

First, we observe that the current mitigations in Linux would only prevent the exploitation of a minority of the bug (3 out of 16). Thus, even if the remaining ones might also be exploited using control-flow hijacking, we think that they can at least be used to perform a data-only attack on the system, therefore bypassing CFI. We also remark that the exploitable bugs that Syzkaller founds are mainly use-after-frees. This could be explained by the fact that this type of bug is very difficult to find due to the quantity of interactions the kernel has between its different subsystems: it is then complicated to reason about the possible execution path, for example when implementing locking. In our classification, on the 11 exploitable use-after-frees, 9 of them require a race condition to trigger the bug according to their Syzkaller reproducer. Intuitively, it also makes sense that bugs that are triggered only undeterministically are harder to find using the kernel unit tests or other code verification method, while fuzzing using concurrent processes can hit these bugs more easily. It is also possible that Syzbot generated some false negative alerts as not every memory issues are reported: for example a heap out-of-bounds write which only overflows in the same object, or only in another neighbor object without corrupting the data in-between cannot be distinguished by the sanatizer from a legitimate write and no error will be reported.

Now, the major limitation of our classification is the absence of tangible proof for each bug: although we cannot provide non-exploitability proofs for bugs that we classified as harmless, an exploit can serve as a proof of exploitability and show that the bug is indeed correctly classified. However, implementing an exploit is time-consuming, in particular for kernel exploitation as the system is very complex, has a lot of interactions and a large code base. Therefore, we only implemented a working exploit on one bug drawn from our classification and otherwise relied on heuristics. Thus, it is possible than some bugs that we classify as exploitabable are in fact harmless, and conversely. For example, a use-after-free requiring a race condition might not leave a sufficient window for the attacker to re-allocate the memory spot of the vulnerable object, rendering the exploitation harder or impossible. Automating this work is then an interesting path to explore: using an *Automatic Exploit Generation* (AEG) tool that handles kernel exploitation could facilitate this kind of bug classification tasks. It could also help developers to triage bugs according to the threat they create in the kernel and prioritize releasing fixes for them: for example, in our classification only one of the bug we classified as exploitable had a CVE identifier attributed (CVE-2019-14821), contrary to the bug we used to implement our end-to-end data-only attack and which was in itself then not considered as a vulnerability. With AEG, one could also more quickly assess the effectiveness of new mitigations to see how they cope with a set of existing vulnerabilities, and conclude whether it is worth merging it in the mainline kernel or not. In this sense, several recent works published advances on exploit generation for the kernel: FUZE [39] aims at facilitating the exploitation of use-after-free bugs in the kernel, while KOOBE [10] focus on heap out-of-bounds write. Otherwise, orthogonal works such as SLAKE [11] or KEPLER [38] automates specific parts of the exploit by respectively helping to shape the slab memory at the attacker's will or by automating the privilege escalation of a given a control-flow hijack primitive. However, these tools are still at the research stage and cannot be used at a production level in their current state.

With the three data-only attacks implemented, we also remark the power of this attack type: not only we managed to reach the same goal (getting *root* privileges as an unprivileged attacker), but we also managed to bypass more mitigations than the original exploit (SMAP and KASLR). Also, we note that the attack are not necessarily more complicated than their control-flow hijack attack alternative. They require wider knowledge of the kernel objects and structures to find the appropriate sensitive data to corrupt, but the implementation of the exploitations are otherwise easier: they do not require advanced technique such as return-oriented programming to bypass existing mitigations and are therefore also more portable across different kernel builds and versions. This convenience mostly come from the fact that protections against data-only attacks are still lacking due to the difficulty of protecting such a wide attack surface with an acceptable performance impact (see subsection 2.4.3). Now, the kernel still integrates several mitigations that could prevent our data-only attacks, however all of them were disabled by default in the targeted kernels. For example, to protect sensitive targets, the `CONFIG_STATIC_USERMODEHELPER` kernel configuration toggle remove the `core_pattern` symbol and instead uses a constant path. In modern kernels (starting from Linux 5.10), the `addr_limit` is removed too and the code is adapted consequently. Also, the `CONFIG_RANDSTRUCT` flag allows randomizing the fields order in several sensitive structures, such as `struct task_struct` (although `addr_limit` is not in the randomized set) or `strcut cred`. However, this process is done at build time, so if attackers target widely distributed kernels such as the ones in major Linux distributions, they can still easily compute the offsets of their target. There are also mitigations that protect the primitives we used: the `CONFIG_DEBUG_LIST` (called `CONFIG_BUG_ON_DATA_CORRUPTION` in more recent kernels) option detects corruptions of the next and previous element pointers, which makes the usage of lists as arbitrary writes primitives more difficult. However, note that for unknown reasons this flags only protects `list` and not `hlist`, the linked-list variant used in the victim code, so the exploit presented in subsection 4.2.3 would actually still work. The kernel also proposes mitigations that makes the exploit harder to develop, such `CONFIG_SLAB_FREELIST_RANDOM` which randomizes the freelist of the slabs to make heap shaping less reliable. To ease the adoption of such flags the *Kernel Self Protection Project* (KSLP) publishes a list [23] of protection-related flags a user who wants a hardened kernel configuration can enable. This helps to block large scale attacks tailored to kernels using default parameters. However, as most of these flags require to recompile the kernel, this list is mostly destined to advanced users or server administrators, and their adoption in widely used distributions is still not earned.

Finally, note that the practical exploit implementations highlight an interesting fact about the attacker threat model: although none of the bugs we classified as exploitable is an arbitrary write primitive in itself, we showed in our exploits that increasing the capabilities up to an arbitrary read or write is common. Thus, we argue that considering an attacker with arbitrary read and write capabilities is a reasonable assumption when discussing kernel exploitation with default mitigations. Also, we showed the power of such a primitive, as it immediately gives a way to escalate our privileges. Note that we only covered privilege escalation attacks, but others are possible: for example, using the arbitrary read permitted by CVE-2017-11176 (subsection 4.2.3), we were able to read other processes memory, and as a demonstration we successfully extracted the SSH private key of a local server, which could then allow to impersonate the server. Data-only attacks are thus not only powerful but also versatile.

Chapter 6

Conclusion

In this thesis, we did an analysis of over 100 real bugs in the Linux kernel to estimate the effectiveness of current mitigations, including the latest Control-Flow Integrity protection which is on its way to being merged in Linux. The need of such classification stems from the fact that the effectiveness of this mitigation in the kernel is still not clear, as by the quantity of sensitive data stored in memory, data-only attacks could easily bypass the restrictions. Therefore, analyzing how the mitigation cope with a data set of existing bugs and vulnerabilities gives a concrete way of assessing the impact of such attacks on the kernel. The results show that current mitigations would prevent only 3 out of the 16 vulnerable bugs from being exploited. In order to give support to the reliability of the classification and show the power of data-only attaks, we also implemented three exploits on bugs that were found in the Linux kernel. One of them is drawn from the bugs we classified, while the two others are existing vulnerabilities for which a public control-flow hijacking exploit exist. For the three bugs, we managed to take over the system with *root* privileges, assuming an unprivileged local attacker. However, this does not mean that current mitigations are useless: control-flow hijacking is a powerful attack and should be considered as such, but data-only attacks is a threat which is equally powerful, and mitigations in this regard are lacking. Moreover, by the quantity of sensitive data found in the kernel memory, it is hard to come up with protections that combine completeness in the data they protect, effectiveness and performance. Thus, we think that protecting the kernel against this type of attacks will be an important challenge in future kernel security researches.

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. “Control-Flow Integrity Principles, Implementations, and Applications”. en. In: *ACM Trans. Inf. Syst. Secur.* 13.1 (Oct. 2009), pp. 1–40. ISSN: 1094-9224, 1557-7406. DOI: `10.1145/1609956.1609960`.

[2] AMD. *AMD64 Architecture Programmer’s Manual, Volumes 1-5, 40332, 24592, 24593, 24594, 26568, 26569*. en. 2020.

[3] Arm. *Arm Architecture Reference Manual Armv8, for Armv8-A Architecture Profile*. en.

[4] Ben. *Project Zero: Exploiting the Linux Kernel via Packet Sockets*. https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html. May 2017.

[5] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. “Jump-Oriented Programming: A New Class of Code-Reuse Attack”. en. In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security - ASIACCS ’11*. Hong Kong, China: ACM Press, 2011, p. 30. ISBN: 978-1-4503-0564-8. DOI: `10.1145/1966913.1966919`.

[6] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. “Control-Flow Integrity: Precision, Security, and Performance”. en. In: *ACM Comput. Surv.* 50.1 (Apr. 2017), pp. 1–33. ISSN: 0360-0300, 1557-7341. DOI: `10.1145/3054924`.

[7] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. “KASLR: Break It, Fix It, Repeat”. en. In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. Taipei Taiwan: ACM, Oct. 2020, pp. 481–493. ISBN: 978-1-4503-6750-9. DOI: `10.1145/3320269.3384747`.

[8] Nicolas Carlini, Antonio Barresi, Mathias Payer, and David Wagner. “Control-Flow Bending: On the Effectiveness of Control-Flow Integrity”. en. In: (), p. 16.

[9] Shuo Chen, Jun Xu, Emre C Sezer, Prachi Gauriar, and Ravishankar K Iyer. “Non-Control-Data Attacks Are Realistic Threats”. en. In: (Aug. 2005), p. 15.

[10] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. “KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities”. en. In: (), p. 18.

[11] Yueqi Chen and Xinyu Xing. “SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel”. en. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. London United Kingdom: ACM, Nov. 2019, pp. 1707–1722. ISBN: 978-1-4503-6747-9. DOI: 10.1145/3319535.3363212.

[12] *Control Flow Integrity in the Android Kernel.* en. https://android-developers.googleblog.com/2018/10/control-flow-integrity-in-android-kernel.html.

[13] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. “PT-Rand: Practical Mitigation of Data-Only Attacks against Page Tables”. en. In: *Proceedings 2017 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2017. ISBN: 978-1-891562-46-4. DOI: 10.14722/ndss.2017.23421.

[14] Federico Manuel Bento. *Control-Flow Integrity for the Linux Kernel:A Security Evaluation.* 2019.

[15] *GDB: The GNU Project Debugger.* https://www.gnu.org/software/gdb/.

[16] *GitHub Linux Source Code.* en. https://github.com/torvalds/linux. 2019.

[17] *Google/Syzkaller.* https://github.com/google/syzkaller. Mar. 2021.

[18] *HexHive/Thesis_template.* https://github.com/HexHive/thesis_template. Mar. 2021.

[19] Intel®. *Intel® 64 and IA-32 Architectures Software Developer’s Manual...* en.

[20] *KASAN: Slab-out-of-Bounds Write in Tcindex_set_parms.* https://syzkaller.appspot.com/bug?id=ea260693da894e7b078d18fca2c9c0a19b457534.

[21] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. “Ret2dir: Rethinking Kernel Isolation”. en. In: (), p. 16.

[22] *Kernel Address Space Layout Randomization [LWN.Net].* https://lwn.net/Articles/569635/.

[23] *Kernel Self Protection Project/Recommended Settings - Linux Kernel Security Subsystem.* https://kernsec.org/wiki/index.php/Kernel_Self_Protection_Project/Recommended_Settings.

[24] *Kernel/Git/Torvalds/Linux.Git - Kbuild: Add Support for Clang LTO.* https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=dc5723b02e52.

[25] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. “TagBleed: Breaking KASLR on the Isolated Kernel Address Space Using Tagged TLBs”. en. In: *2020 IEEE European Symposium on Security and Privacy (EuroS&P).* Genoa, Italy: IEEE, Sept. 2020, pp. 309–321. ISBN: 978-1-72815-087-1. DOI: 10.1109/EuroSP48549.2020.00027.

[26] *Lexfo’s Security Blog - CVE-2017-11176: A Step-by-Step Linux Kernel Exploitation.* https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part1.html.

[27] *Linux Kernel Universal Heap Spray - Vitaly Nikolenko.* https://duasynt.com/blog/linux-kernel-heap-spray.

[28] *Mm.Txt - Documentation/X86/X86_64/Mm.Txt - Linux Source Code (v4.19.108) - Bootlin.* https://elixir.bootlin.com/linux/v4.19.108/source/Documentation/x86/x86_64/mm.txt.

[29] Weizhong Qiang, Jiawei Yang, Hai Jin, and Xuanhua Shi. “PrivGuard: Protecting Sensitive Kernel Data From Privilege Escalation Attacks”. In: *IEEE Access* 6 (2018), pp. 46584–46594. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2866498.

[30] *Seccomp(2) - Linux Manual Page.* https://man7.org/linux/man-pages/man2/seccomp.2.html.

[31] Hovav Shacham. “The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86)”. In: *Proceedings of the 14th ACM Conference on Computer and Communications Security.* CCS ’07. New York, NY, USA: Association for Computing Machinery, Oct. 2007, pp. 552–561. ISBN: 978-1-59593-703-2. DOI: 10.1145/1315245.1315313.

[32] *ShadowCallStack — Clang 12 Documentation.* https://clang.llvm.org/docs/ShadowCallStack.html.

[33] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. “Enforcing Kernel Security Invariants with Data Flow Integrity”. en. In: *Proceedings 2016 Network and Distributed System Security Symposium.* San Diego, CA: Internet Society, 2016. ISBN: 978-1-891562-41-9. DOI: 10.14722/ndss.2016.23218.

[34] *Syzbot.* https://syzkaller.appspot.com/upstream.

[35] *The Kernel Address Sanitizer (KASAN) — The Linux Kernel Documentation.* https://www.kernel.org/doc/html/v4.19/dev-tools/kasan.html.

[36] The Linux Kernel documentation. *Unreliable Guide To Locking.* https://www.kernel.org/doc/html/latest/kernel-hacking/locking.html#.

[37] Fernando Vano-Garcia and Hector Marco-Gisbert. “KASLR-MT: Kernel Address Space Layout Randomization for Multi-Tenant Cloud Systems”. en. In: *Journal of Parallel and Distributed Computing* 137 (Mar. 2020), pp. 77–90. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2019.11.008.

[38] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. “KEPLER: Facilitating Control-Flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities”. en. In: (), p. 18.

[39] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. “FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities”. en. In: (), p. 17.

[40] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W Hamlen, and Zhiqiang Lin. “CONFIRM: Evaluating Compatibility and Relevance of Control-Flow Integrity Protections for Modern Software”. en. In: (2019), p. 18.