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

A survey of kernel-exploitation techniques

Vincenzo Terracciano

Master Thesis

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

Università
di Genova

MSc Computer Science
Software Security Curriculum

A survey of kernel-exploitation techniques

Vincenzo Terracciano

Advisor: Giovanni Lagorio

Examiner: ....

July, 2021

Table of Contents

Chapter 1

Introduction

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

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

The reference to the memory visible only to the kernel as kernel-land memory and the memory a user process sees as user-land memory. Code executing in kernel-land runs with full privileges and can access any valid memory address on the system, while code executing in user-land is subject to all limits as describe above. This separation between hardware- and software-based is necessary to protect the kernel from accidental damage or alteration resulting from a misbehaving or malicious user-land application.

Chapter 2

Exploitation

There are various ways an attacker can behave as a *super-user*, the most excitement is generally performed with the development of an exploit. The meaning behind *exploitation* is really simple: software has bugs(collegamento con significato?), and these make the software work not correctly, or otherwise perform incorrectly a task that had to perform in an appropriate way. And all this means an advantage for the *attacker*. Not every bug is exploitable; we refer to those that are as *vulnerabilities*. Analyzing an application to establish its vulnerability is called *AUDITING*(collegamento a wiki). It entails:

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

2.1 Kernel-land exploits VS User-land exploits

Until now the kernel has been described as the entity through which many countermeasures against exploitation are realized.

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

- The kernel is the only piece of software that is strictly for the system. As long as the kernel works correctly, there is no incorrigible situation. This explains why user-land brute forcing, for example, is a feasibly choice: the only real worry we have to confront when we repeatedly crash our victim application is the noise we might create in the logs. When it comes to the kernel, this hypothesis isn't true anymore: an error at the kernel level leaves the system in an *inconsistent state*, and it's usually required to take back the machine to its appropriate functioning. If the error happens inside one of the sensible areas of the kernel, the operating system will just shut down, a condition known as panic.

- The kernel is protected from user-land via both software and hardware. Finding information about the kernel is a much more difficult job. At the same time, the number of variables that are no more under the attacker's control intensifies in an exponentially way. For example, let's consider the *memory allocator*. In a user-land exploit, the allocator is inside the process, generally connected through a shared system library. Your purpose is its only consumer and its only *affecter*. On the other side, all the processes on the system may concern the behavior and the status of a kernel memory allocator.

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

Chapter 3

Boh

3.1 Debugging

In user space we had the support of the kernel so we could easily stop processes and use gdb to inspect their behavior. GDB(collegamento wiki) allows you to see what is going on *inside* another program while it executes – or what another program was doing at the moment it crashed.

3.1.1 GDB

GDB can do four main kinds of things to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another. Using gdb as a front-end for the kernel debugger allows us to debug the kernel in the familiar and powerful debugging interface of gdb. In the kernel, in order to use gdb we need to use hypervisor like QEMU(rif wiki QEMU is a generic and open source machine emulator and virtualizer.) based hardware interfaces which are not always available. The Linux kernel provides a set of tools and debug options useful for investigating abnormal behavior.

3.2 Building the kernel

I recommended that you build and install your own kernel, rather than running the stock kernel that comes with your distribution. One of the strongest reasons for running your own kernel is that the kernel developers have built several debugging features into the kernel itself. These features can create extra output and slow performance, so they tend not to be enabled in production kernels from distributors.

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

Except where specified otherwise, all of these options are found under the "kernel hacking" menu in whatever kernel configuration tool you prefer. Note that some of these options are not supported by all architectures and if it is addes, it is not considered for the building of kernel.

CONFIG˙DEBUG˙KERNEL This option just makes other debugging options available; it should be turned on but does not, by itself, enable any features.

CONFIG˙DEBUG˙SLAB This crucial option turns on several types of checks in the kernel memory allocation functions; with these checks enabled, it is possible to detect a number of memory overrun and missing initialization errors. Each byte of allocated memory is set to 0xa5 before being handed to the caller and then set to 0x6b when it is freed. If you ever see either of those "poison" patterns repeating in output from your driver (or often in an oops listing), you'll know exactly what sort of error to look for. When debugging is enabled, the kernel also places special guard values before and after every allocated memory object; if those values ever get changed, the kernel knows that somebody has overrun a memory allocation, and it complains loudly. Various checks for more obscure errors are enabled as well.

CONFIG˙DEBUG˙PAGEALLOC Full pages are removed from the kernel address space when freed. This option can slow things down significantly, but it can also quickly point out certain kinds of memory corruption errors.

CONFIG˙DEBUG˙SPINLOCK With this option enabled, the kernel catches operations on uninitialized spinlocks and various other errors (such as unlocking a lock twice).

CONFIG˙INIT˙DEBUG Items marked with ˙ ˙init (or ˙ ˙initdata) are discarded after system initialization or module load time. This option enables checks for code that attempts to access initialization-time memory after initialization is complete.

CONFIG˙DEBUG˙INFO This option causes the kernel to be built with full debugging information included. Including debug information in the kernel and kernel modules will make both the image and the modules larger in size, but is a required option for debugging the kernel or kernel modules with gdb. It is necessary if you want to debug

the kernel with gdb. You may also want to enable CONFIG˙FRAME˙POINTER if you plan to use gdb.

CONFIG˙DEBUG˙STACK˙USAGE CONFIG˙DEBUG˙STACKOVERFLOW to check the overflows of kernel, IRQ and exception stacks. This option will cause messages of the stacks in detail when free stack space drops below a certain limit. A sure sign of a stack overflow is an oops listing without any sort of reasonable back trace. The first option adds explicit overflow checks to the kernel; the second causes the kernel to monitor stack usage and make some statistics available via the magic SysRq key.

(oops is a deviation from correct behavior of the Linux kernel, one that produces a certain error log. The better-known kernel panic condition results from many kinds of oops, but other instances of an oops event may allow continued operation with compromised reliability. The term does not stand for anything, other than that it is a simple mistake.)

CONFIG˙KALLSYMS This option(under "General setup/Standard features") causes kernel symbol information to be built into the kernel; it is enabled by default. The symbol information is used in debugging contexts; without it, an oops listing can give you a kernel traceback only in hexadecimal, which is not very useful.

CONFIG˙IKCONFIG CONFIG˙IKCONFIG˙PROC These options(found in the "General setup" menu) cause the full kernel configuration state to be built into the kernel and to be made available via /proc. Most kernel developers know which configuration they used and do not need these options (which make the kernel bigger). They can be useful, though, if you are trying to debug a problem in a kernel built by somebody else.

CONFIG˙ACPI˙DEBUG Under "Power management/ACPI."This option turns on verbose ACPI (Advanced Configuration and Power Interface) debugging informat

CONFIG˙DEBUG˙DRIVER Under "Device drivers."Turns on debugging information in the driver core, which can be useful for tracking down problems in the low-level support code.

CONFIG˙SCSI˙CONSTANTS(OPTIONAL) This option, found under "Device drivers/SCSI device support," builds in information for verbose SCSI error messages. If you are working on a SCSI driver, you probably want this option.

CONFIG˙INPUT˙EVBUG(OPTIONAL) This option (under "Device drivers/Input device support") turns on verbose logging of input events. If you are working on a driver for an input device, this option may be helpful. Be aware of the security implications of this option, however: it logs everything you type, including your passwords.

CONFIG˙PROFILING This option is found under "Profiling support." Profiling is normally used for system performance tuning, but it can also be useful for tracking

down some kernel hangs and related problems.

CONFIG˙GDB˙SCRIPTS Adds links to the GDB helper scripts. I have found this option to be extremely useful. I find it particularly useful when debugging a kernel module, when I need to inspect the kernel log buffer or VFS mounts, or when I have to do anything with tasks (more on this later).

CONFIG˙KGDB Enables the built in kernel debugger, which allows for remote debugging. Technically this option is the only one that is strictly required, but attempting to debug without debug symbols will make debugging much harder.

CONFIG˙FRAME˙POINTER Depending on the architecture of the VM you're running, you may also want to enable CONFIG˙FRAME˙POINTER. This option adds the compiler flag -fno-omit-frame-pointer and greatly improves the reliability of backtraces.

If you don't want to use menuconfig is possible to set configuration options via command line using the following

$./scripts/config - eCONFIG_{<}youroption >$

Once you have enabled all these options, you need to build the kernel. This is done from the command line:

$make - j$(nproc)

Aggiunto il 30/6/2021

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

LINUX KERNEL MITIGATION FEATURES (LIKE USERLAND)

Just like mitigation features such as ASLR, stack canaries, PIE, etc. used by userland programs, kernel also have their own set of mitigation features. Below are some of the popular and notable Linux kernel mitigation features:

Kernel stack cookies (or canaries): Stack canaries are a mitigation targeted at stack-based buffer overflow attacks. It works by exploiting one of the limitations of these kind of attacks, namely, that the attacker must overwrite all the bytes between the overflown buffer and the control data (i.e., saved registers and the return address). The idea is to put a value—the canary—between the local variables and the control data of each function stack frame. The attacker, thus, has to overwrite the canary before she can overwrite the control data. If overwriting the canary is impossible or can be detected, the attack is blocked. It is enabled in the kernel at compile time and cannot be disabled.

Kernel address space layout randomization (KASLR): Also like ASLR on userland, it is a computer security technique involved in preventing exploitation of memory corruption vulnerabilities. In order to prevent an attacker from reliably jumping to, for example, a particular exploited function in memory, ASLR randomly arranges the address space

positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries. With kernel address space layout randomization (KASLR), the kernel is loaded to a random location in memory. Loading the kernel to a random location can protect against attacks that rely on knowledge of the kernel addresses. The KASLR feature is enabled by default.

OTHER POWERFUL LINUX KERNEL MITIGATION FEATURES

Supervisor mode execution protection (SMEP): The processor introduces a new machanism that provides next level of system protection by blocking malicious software attacks from user mode code when the system is running in the highest privilege level. This feature marks all the userland pages in the page table as non-executable when the process is in kernel-mode. In the kernel, this is enabled by setting the 20th bit of Control Register CR4.

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

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

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

1. Low Entropy - there are only so many locations the kernel can fit in. This means an attacker could guess without too much trouble. 2. Knowledge of a single address can reveal the offset of the base address, exposing all other locations for a published/known kernel image. 3. Info leaks abound.

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

SET-UP QEMU-SOFTWARE

As said previously, to debug the kernel we need the qemu hypervisor. Specifically, there are some options needed in kernel analysis: -kernel ¡path¿, (where ¡path¿ means the) Path to kernel image to debug; -initrd ¡path¿, Path to initial Ram disk. In short, a RAM disk is a filesystem dynamically placed in you RAM memory in boot time that contains the basic stuff needed to get your real filesystem running with the first processes needed to get your whole system running as expected, like the init process; -gdb dev, wait for gdb connection on device dev. Typical connections will likely be TCP-based, but also UDP, pseudo TTY, or even stdio are reasonable use case; -s, shorthand for -gdb tcp::1234, i.e. open a gdbserver on TCP port 1234; -S, freeze the CPU on startup; -cpu model, select CPU model. Here we can add +smep and +smap for SMEP and SMAP mitigation features; -m [size=]megs, set virtual RAM size to megs megabytes. Default is 128 MiB. Optionally, a suffix of "M" or "G" can be used to signify a value in megabytes or gigabytes respectively. Optional pair slots, maxmem could be used to set amount of hotluggable memory slots and possible maximum amount of memory; -append, specifies additional boot options. This is also where we can enable/disable mitigation features. These options are essential for analyzing the kernel. But qemu supports other options (indicate the documentation site) which may be useful for running the system and to help the user in the analysis.