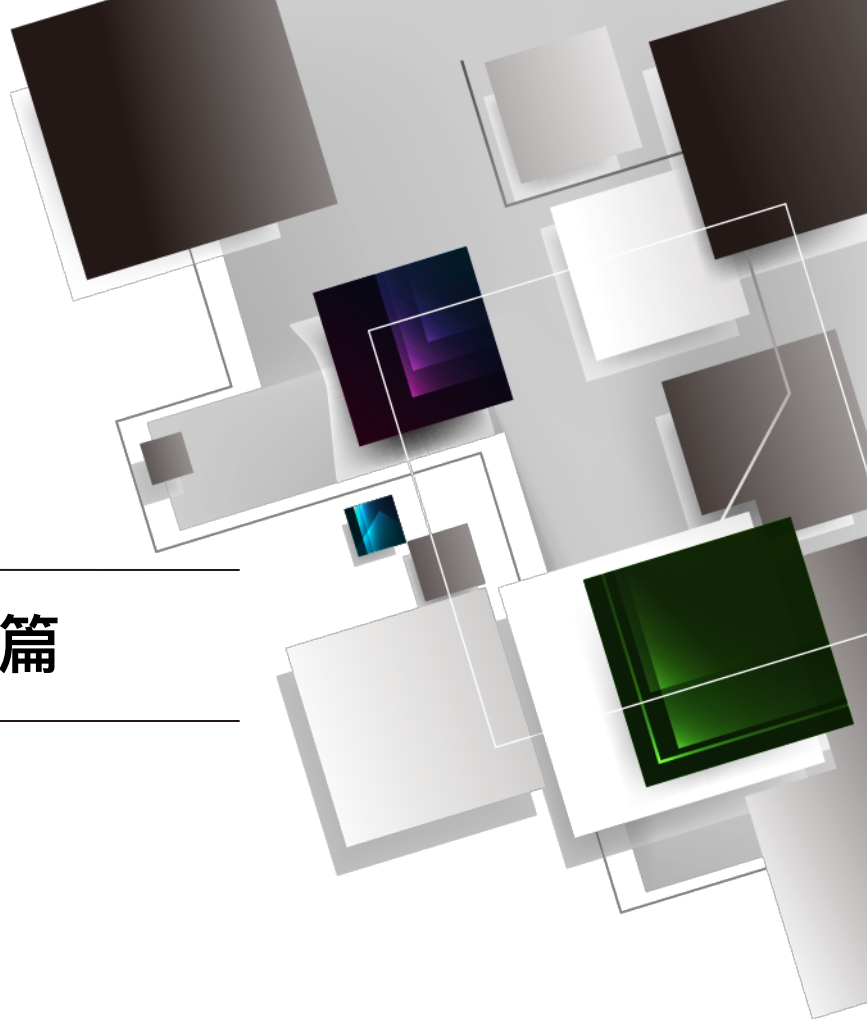


CTF 之 Linux kernel 攻击 入门篇

个人资料

姓名：林国鹏

学校：复旦大学在读（大三）

战队：sixstars

擅长：pwn

比赛经历：WCTF新锐赛亚军
XCTF final 三等奖
大学生国赛一等奖

环境准备

1. Ubuntu 16.04 虚拟机
2. Gdb 及 pwndbg插件
3. Pwntools
4. 安装muslgcc
   apt install musl-tools

常用命令

1. 查看装载驱动 lsmod
2. 查看所开保护cat /proc/cpuinfo
3. 查看内核堆块 cat /proc/slabinfo
4. 查看prepare_kernel_cred和commit_creds地址

```
grep prepare_kernel_cred  /proc/kallsyms
grep commit_creds  /proc/kallsyms
```

目 录

基础知识

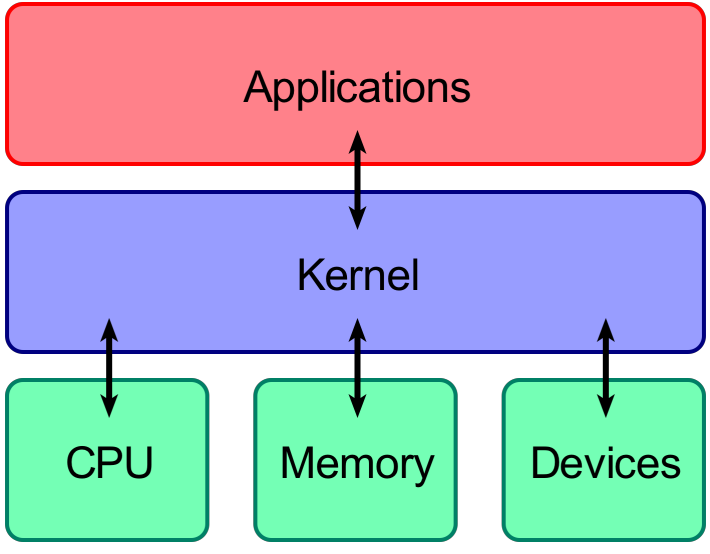
kernel 的主要功能：

1. 控制并与硬件进行交互
2. 提供 application 能运行的环境

intel CPU 将 CPU 的特权级别分为 4 个级别：Ring 0, Ring 1, Ring 2, Ring 3。

Ring0 只给 OS 使用，Ring 3 所有程序都可以使用，内层 Ring 可以随便使用外层 Ring 的资源。

Ps: 在ring0 下，可以修改用户的权限（也就是提权）



基础知识

如何进入kernel 态：

1.系统调用

2.产生异常

3.外设产生中断

等等

基础知识

进入kernel态之前会做什么?

保存用户态的各个寄存器，以及执行到代码的位置

从kernel态返回用户态需要做什么?

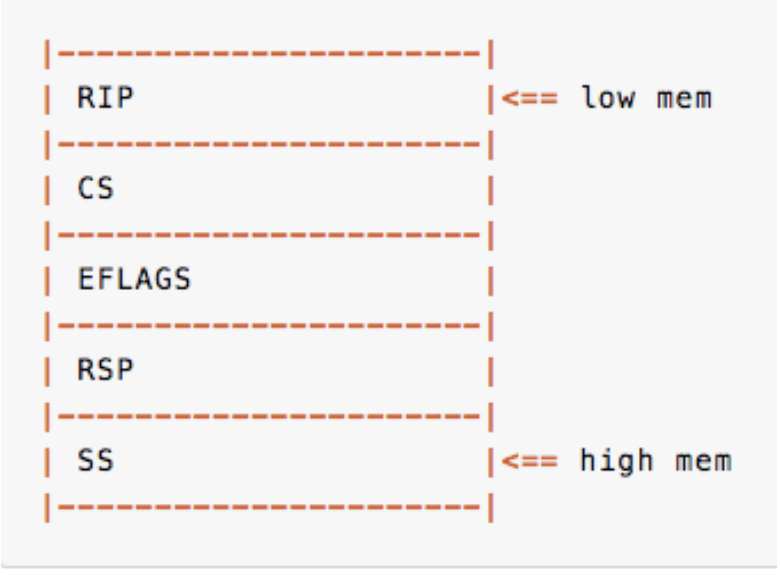
执行swapgs 和 iret 指令，当然前提是栈上需要布置好恢复的寄存器的值

基础知识

一般的攻击思路：

寻找kernel 中内核程序的漏洞，之后调用该程序进入内核态，利用漏洞进行提权，提完权后，返回用户态

返回用户态时候的栈布局：



基础知识

返回用户态时候的栈布局：

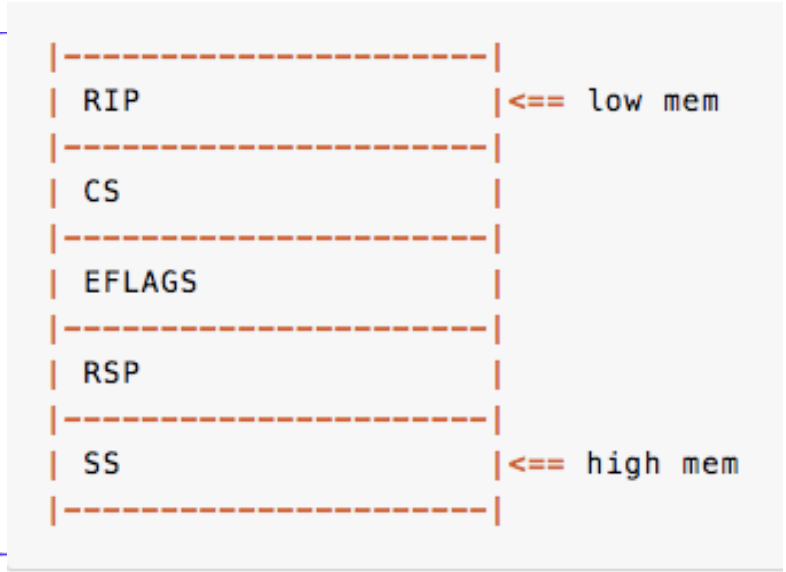
```
0x40036a    call    qword ptr [rip + 0x203cb8]

0x400370    mov     rdi, rax
0x400373    call    rbx

0x400375    mov     rsp, 0x604c60
0x40037c    swapgs
▶ 0x40037f    iretq
0x400381    pop     rbx
0x400382    ret

0x400383    nop     dword ptr [rax]
0x400386    nop     word ptr cs:[rax + rax]
0x400390    xor     edx, edx
```

```
00:0000 | rsp  0x604c60 —▸ 0x400390 ◂— 0xbf004030a5bed231
01:0008 |      0x604c68 ◂— 0x33 /* '3' */
02:0010 |      0x604c70 ◂— 0x206
03:0018 |      0x604c78 ◂— 0x7ffc4e24fba0
04:0020 |      0x604c80 ◂— 0x2b /* '+' */
05:0028 |      0x604c88 ◂— 0xd711003c42c08978
06:0030 |      0x604c90 ◂— 0x0
... ↓
```



基础知识

一般来说，题目会给出如下四个文件：

其中，baby.ko 就是有bug的程序，可以用ida 打开

bzImage 是打包的内核代码，可以用来寻找gadget

Initramfs.cpio 文件系统

startvm.sh 启动脚本

基础知识-做题前准备

1. 修改startvm.sh ， 加入gdb 调试的选项，方便之后调试

```
stty intr ^]
cd `dirname $0`
timeout --foreground 600 qemu-system-i386 \
    -m 64M \
    -nographic \
    -kernel bzImage \
    -append 'console=ttyS0 loglevel=3 oops=panic panic=1 nokaslr' \
    -monitor /dev/null \
    -initrd initramfs.cpio \
    -smp cores=1,threads=1 \
    -cpu qemu32 2>/dev/null
    -gdb tcp::1234
```

基础知识-做题前准备

2. 制作startvm.sh 的副本root.sh， Initramfs.cpio 的副本root.cpio ，方便之后从root 身份启动，查看一些信息

```
#!/bin/bash

stty intr ^]
cd `dirname $0`
timeout --foreground 600 qemu-system-i386 \
    -m 64M \
    -nographic \
    -kernel bzImage \
    -append 'console=ttyS0 loglevel=3 oops=panic panic=1 nokaslr' \
    -monitor /dev/null \
    -initrd root.cpio \
    -smp cores=1,threads=1 \
    -cpu qemu32 2>/dev/null
```

```
cd /home/pwn
setsid cttyhack setuidgid 0000 sh
```

基础知识-做题前准备

用startvm.sh 启动

```
saltedfish@Ubuntu ➜ Linux Kernel Level 0 2 ./startvm.sh
 _  __                 _          _
| |/ /___  ___ _ __ | |    __ _| |__
| ' // _ \/ _ \ '_ \| |   / _` | '_ \
| . \  __/  __/ | | | |__| (_| | |_) |
|_|\_\___|\___|_| |_|_____\__,_|_.__/

~ $
```

用root.sh 启动

```
saltedfish@Ubuntu ➜ Linux Kernel Level 0 2 ./root.sh
[    1.184871] Initramfs unpacking failed: junk in comp
 _  __                 _          _
| |/ /___  ___ _ __ | |    __ _| |__
| ' // _ \/ _ \ '_ \| |   / _` | '_ \
| . \  __/  __/ | | | |__| (_| | |_) |
|_|\_\___|\___|_| |_|_____\__,_|_.__/

/home/pwn #
```

基础知识-做题前准备

用root.sh 启动 并且查看装载的程序（使用lsmod 命令）

```
saltedfish@Ubuntu ➜ Linux Kernel Level 2 ./root.sh
[    1.080449] Initramfs unpacking failed: junk in compressed archive

 _  __                    _           _
| |/ /___  ___ _ __ | |    __ _| |__
| ' // _ \/ _ \ '_ \| |   / _` | '_ \
| . \  __/  __/ | | | |__| (_| | |_) |
|_|\_\___|\___|_| |_|_____\__,_|_.__/

/home/pwn # lsmod
baby 16384 0 - Live 0xffffffffc0002000 (POE)
```

基础知识-常见几个保护

1. Kaslr 地址随机化

在startvm.sh 中，一般本地调试时，改为nokaslr

```
#!/bin/bash

stty intr ^]
cd `dirname $0`
timeout --foreground 10000 qemu-system-x86_64 \
    -m 64M \
    -nographic \
    -kernel bzImage \
    -append 'console=ttyS0 loglevel=3 oops=panic panic=1 kaslr' \
    -monitor /dev/null \
    -initrd initramfs.cpio \
    -smp cores=1,threads=1 \
    -cpu qemu64,smep 2>/dev/null \
    -gdb tcp::1234
```

2. Smep 内核态不可执行用户态代码

root.sh 启动

cat /proc/cpuinfo

3. Smap 内核态不可访问用户态内存

root.sh 启动

cat /proc/cpuinfo

```
saltedfish@Ubuntu ➜ Linux Kernel Level 5 ./root.sh
[    1.105678] Initramfs unpacking failed: junk in compressed archive
 _  __                 _                 _
| |/ /___ ___ _ __ ___| |    __ _| |__
| ' // _ \/ _ \ '_ \ |  |   / _` | '_ \
| . \  __/  __/ | | | |__| (_| | |_) |
|_|\_\___|\___|_| |_|_____\__,_|_.__/

/home/pwn # cat /proc/cpuinfo
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 6
model           : 6
model name      : QEMU Virtual CPU version 2.5+
stepping        : 3
cpu MHz         : 1799.905
cache size      : 512 KB
physical id     : 0
siblings        : 1
core id         : 0
cpu cores       : 1
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 sysca
opl cpuid pni cx16 hypervisor lahf_lm svm 3dnowprefetch vmmcall smep smap
bugs            : fxsave_leak sysret_ss_attrs spectre_v1 spectre_v2 spec_store_bypass
```

栈溢出

```
__int64 __usercall sub_0@<rax>(__int64 a1@<rbp>, int command@<esi>)
{
  __int64 src; // rdx
  __int64 dst; // [rsp-88h] [rbp-88h]
  __int64 v5; // [rsp-8h] [rbp-8h]

  ((void (*)(void))_fentry__)();
  if ( command != 0x6001 )
    return 0LL;
  v5 = a1;
  return (signed int)copy_from_user(&dst, src, 0x100LL);
}
```

程序从src中拷贝0x100 个字节到dst ， 存在溢出，因此可以控制程序的返回地址

Ps: src 是用户态的地址， dst 是内核函数中栈上的地址

栈溢出

一般思路（什么保护都没有）：

1. 利用溢出调用commit_creds(prepare_kernel_cred(0))进行提权

2. 返回用户态

栈溢出

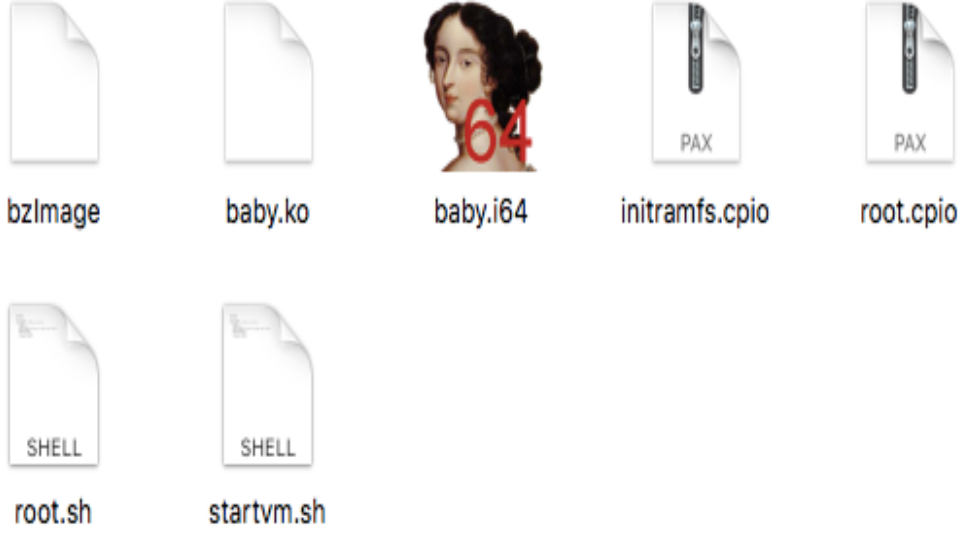
例题：

level 1

栈溢出

例题：level1

1. 制作root.cpio 和 root.sh



栈溢出

例题：level1

2. 查看装载的驱动

```
saltedfish@Ubuntu ➜ Linux Kernel Level 2 ./root.sh
[    1.508372] Initramfs unpacking failed: junk in compressed archive
 _  __                    _           _
| |/ /___  ___ _ __ | |    __ _| |__
| ' // _ \/ _ \ '_ \| |   / _` | '_ \
| . \  __/  __/ | | | |__| (_| | |_) |
|_|\_\___|\___|_| |_|_____\__,_|_.__/

/home/pwn # lsmod
baby 16384 0 - Live 0xffffffffc0002000 (POE)
```

栈溢出

例题：level1

3.查看prepare_kernel_cred和commit_creds的地址

```
/home/pwn # grep prepare_kernel_cred  /proc/kallsyms
ffffffff810b9d80 T prepare_kernel_cred
ffffffff821e0300 r __ksymtab_prepare_kernel_cred
ffffffff821efc91 r __kstrtab_prepare_kernel_cred
/home/pwn # grep commit_creds  /proc/kallsyms
ffffffff810b99d0 T commit_creds
ffffffff821db378 r __ksymtab_commit_creds
ffffffff821efccd r __kstrtab_commit_creds
```

栈溢出

例题：level1

一般采用ioctl(driver_fd,command,&struct)与驱动程序进行交互

SMEP bypass

一般思路：

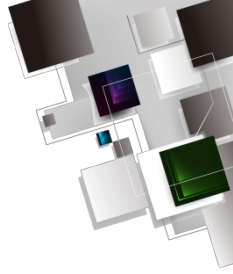
1. ROP

既然不允许执行用户态的代码，那么就直接使用内核态代码吧！

2. 修改cr4寄存器

系统根据 CR4 寄存器的值判断是否开启 smep 保护，当 CR4 寄存器的第 20 位是 1 时，保护开启；是 0 时，保护关闭。

CR4 寄存器是可以通过 mov 指令修改的，一般修改为0x6f0



SMEP bypass

例题：level2

Double fetch

从漏洞原理上属于条件竞争漏洞，是一种内核态

与用户态之间的数据访问竞争。（来自 CTF WIKI）

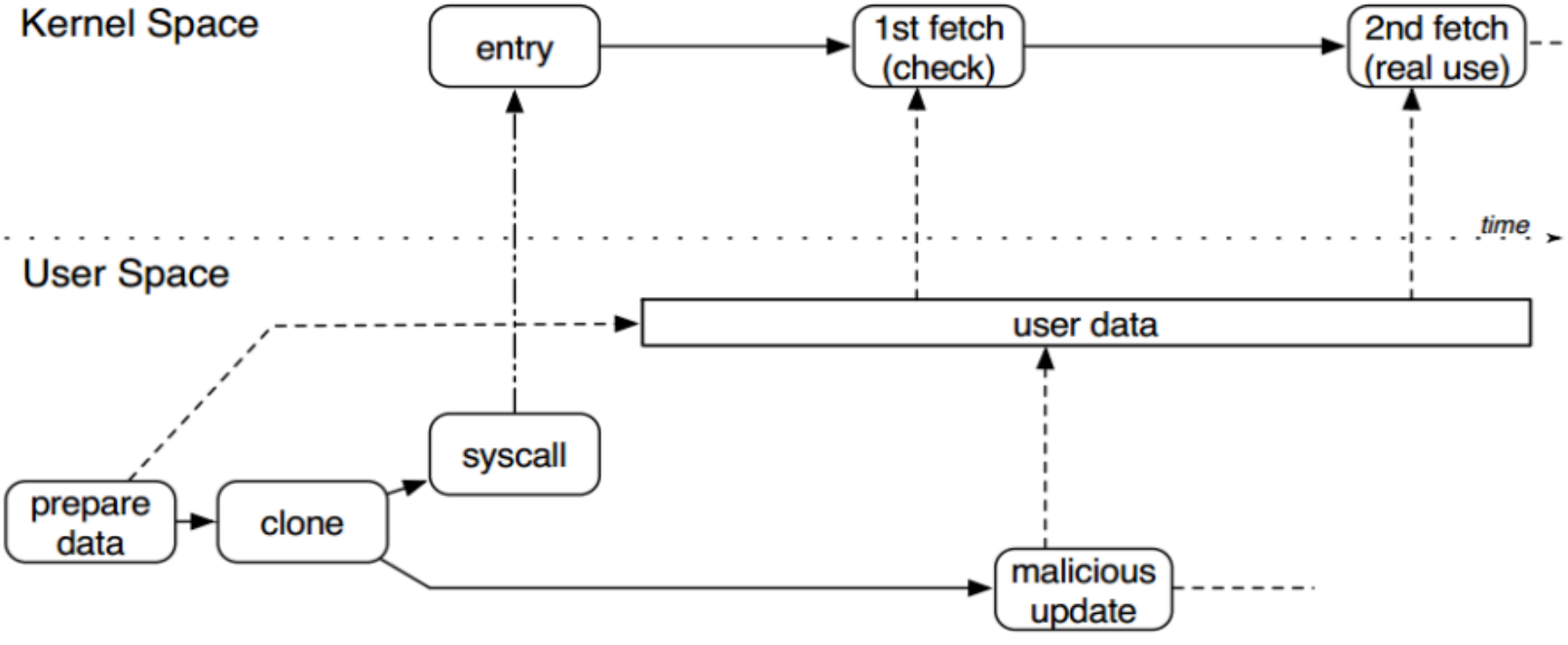
Double fetch

在 Linux 等现代操作系统中，虚拟内存地址通常被划分为内核空间和用户空间。内核空间负责运行内核代码、驱动模块代码等，权限较高。而用户空间运行用户代码，并通过系统调用进入内核完成相关功能。通常情况下，用户空间向内核传递数据时，内核先通过通过 copy_from_user 等拷贝函数将用户数据拷贝至内核空间进行校验及相关处理，但在输入数据较为复杂时，内核可能只引用其指针，而将数据暂时保存在用户空间进行后续处理。此时，该数据存在被其他恶意线程篡改风险，造成内核验证通过数据与实际使用数据不一

Double fetch

一个典型的 Double Fetch 漏洞原理如下图所示，一个用户态线程准备数据并通过系统调用进入内核，该数据在内核中有两次被取用，内核第一次取用数据进行安全检查（如缓冲区大小、指针可用性等），当检查通过后内核第二次取用数据进行实际处理。而在两次取用数据之间，另一个用户态线程可创造条件竞争，对已通过检查的用户态数据进行篡改，在真实使用时造成访问越界或缓冲区溢出，最终导致内核崩溃或权限提升。（来自 CTF WIKI）

Double fetch



Double fetch

例题：level3

UAF

与用户态的UAF相似，都是在某一个结构体释放后忘记将指针清空

常见三种情况：

1. 结构体A本身不存在函数指针，但是free 后可以edit,那么需要让另外一个含有函数指针的结构体B放入相同位置，之后修改函数指针

2.结构体A本身不存在函数指针，但是free 后可以edit,那么让cred结构体分配到相同位置，修改cred结构体进行提权

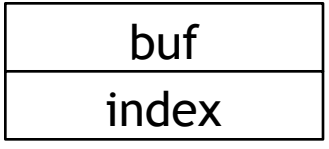
3. 结构体A本身有函数指针，但是free 之后无法edit,这时候需要分配一个可以edit 的结构体B放入相同相同位置，修改函数指针

一般结构体A 是题目提供的，而结构体B可以调用linux操作系统自

UAF

例题：level4

Iotcl(fd,command,&info)



UAF

例题：level4

基本思路：

释放堆块后，调用send_msg(),内核会将释放的堆块重新分配，而且此时的内容可控，即为想要传输的msg

参考链接：1. (https://invictus-security.blog/2017/06/15/linux-kernel-heap-spraying-uaf/)

2. (https://duasynt.com/blog/linux-kernel-heap-spray)

UAF

例题：level5

参考链接：https://www.anquanke.com/post/id/86490

例题：level5

大致思路：

free 完结构体后，打开ptmx 设备，则内核会在之前free 的位置分配另外一个结构体B，并且这个结构体B有一个指针指向了另外一个结构体C，结构体C上有一个函数指针，我们通过伪造结构体C，并且修改B的指针指向我们伪造的结构体C来劫持程序执行流

谢谢观看！