linux内核栈溢出,【Linux内核漏洞利用】2018强网杯core_栈溢出

转载

心理咨询师清晨 于 2021-05-01 08:23:03 发布 117 收藏
文章标签： linux内核栈溢出

一.linux内核漏洞利用预备知识

1.介绍

(1)ioctl ：用于与设备通信。int ioctl(int fd, unsigned long request, ...) 的第一个参数为打开设备 (open) 返回的文件描述符，第二个参数为用户程序对设备的控制命令，再后边的参数则是一些补充参数，与设备有关。

(2)struct cred：kernel 记录了进程的权限，更具体的，是用 cred 结构体记录的，每个进程中都有一个 cred 结构，这个结构保存了该进程的权限等信息(uid，gid 等)，如果能修改某个进程的 cred，那么也就修改了这个进程的权限。

(3)内核态函数

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。

printf() -> printk() 可用dmesg查看

memcpy() -> copy_from_user()/copy_to_user()

malloc() -> kmalloc()

free() -> kfree()

(4)改变权限的函数：执行 commit_creds(prepare_kernel_cred(0)) 即可获得 root 权限(root 的 uid，gid 均为 0)。两个函数的地址都可以在 /proc/kallsyms 中查看(较老的内核版本中是 /proc/ksyms，/proc/kallsyms 的内容需要 root 权限才能查看)。

图1. commit_creds与prepare_kernel_cred函数地址

(5)攻击方法

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。

(6)Mitigation缓解措施：

canary, dep, PIE, RELRO 等保护与用户态原理和作用相同

smep: Supervisor Mode Execution Protection，当处理器处于 ring0 模式，执行 用户空间的代码会触发页错误。(在 arm 中该保护称为 PXN)

smap: Superivisor Mode Access Protection，类似于 smep，通常是在访问数据时。

2.状态切换

user space to kernel space

当发生 系统调用，产生异常，外设产生中断等事件时，会发生用户态到内核态的切换，具体的过程为：

(1)通过 swapgs 切换 GS 段寄存器，将 GS 寄存器值和一个特定位置的值进行交换，目的是保存 GS 值，同时将该位置的值作为内核执行时的 GS 值使用。

(2)将当前栈顶(用户空间栈顶esp)记录在 CPU 独占变量区域里，将 CPU 独占区域里记录的内核栈顶放入 rsp/esp。

(3)通过 push 保存各寄存器值，具体的 代码 如下:

图2. 进入内核时保存寄存器

(4)通过汇编指令判断是否为 x32_abi。 5. 通过系统调用号，跳到全局变量sys_call_table相应位置继续执行系统调用。

kernel space to user space

退出时，流程如下：

(1)通过 swapgs 恢复 GS 值

(2)通过 sysretq 或者 iretq 恢复到用户控件继续执行。如果使用 iretq 还需要给出用户空间的一些信息(CS, eflags/rflags, esp/rsp 等)

3.环境搭建-QEMU

$ sudoapt-get update

$ sudoapt-get installgitfakeroot build-essential ncurses-dev xz-utils libssl-dev bc qemu qemu-system

4.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：

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。

(2)proc_create创建文件

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。

二、题目分析

1.文件系统预处理

(1)解压后含4个文件：

bzlmage：kernel映像

core.cpio：文件系统映像

start.sh：一个用于启动 kernel 的 shell 的脚本，多用 qemu，保护措施与 qemu 不同的启动参数有关

vmlinux：类比成用户态pwn中的libc文件。解压core.cpio之后core目录里也有个vmlinux，调试时用core目录的vmlinux。　最新解释：外面是个带符号表的 vmlinux?

vmlinux 未经过压缩，也就是说我们可以从 vmlinux 中找到一些 gadget，我们先把 gadget 保存下来备用。如果题目没有给 vmlinux，可以通过 extract-vmlinux 提取(命令：./extract-vmlinux ./bzlmage > vmlinux)。

(2)修改start.sh脚本：

-m 64 设置内存大小不够，改成128

-append "root=/dev/ram rw console=ttyS0 oops=panic panic=1 quiet kaslr" \ #开启了ASLR

(3)修改文件系统(解包并重打包)：

干掉定时power down：去掉init中“poweroff -d 120 -f &”

(非必须)root权限调试：将“setsid /bin/cttyhack setuidgid 1000 /bin/sh” 中的1000改为0。

解包命令

(4)解包文件系统分析

文件系统cpio解包

core.ko：含有漏洞的目标驱动文件

gen_cpio.sh方便打包的文件

init：启动后进行初始化的文件

init系统初始化脚本分析

2.调试方法

(1)如何调试

$ qemu-system-x86_64--help|grepgdb

-gdbdev　　waitforgdb connection on'dev'

-sshorthandfor-gdbtcp::1234

可以通过 -gdb tcp:port 或者 -s 来开启调试端口，-s即为“-gdb tcp::1234”缩写

(2)加载驱动 core.ko 的符号表

命令： add-symbol-file core.ko textaddr ， textaddr是core.ko加载地址，用cat /sys/module/core/sections/.text获取

(3)流程

调试流程

3.代码分析：

首先，core.ko开了canary和nx(checksec core.ko)，内核开了kaslr没有开smep

init_module()注册了/proc/core

exit_core()删除 /proc/core

core_ioctl()

定义了三条命令，分别调用 core_read()，core_copy_func() 和设置全局变量 off

core_read()

从 v4[off] 拷贝 64 个字节到用户空间，但要注意的是全局变量 off 使我们能够控制的，因此可以合理的控制 off 来 leak canary 和一些地址。

core_copy_func()

从全局变量 name 中拷贝数据到局部变量中，长度是由我们指定的，当要注意的是 qmemcpy 用的是 unsigned __int16，但传递的长度是 signed __int64，因此如果控制传入的长度为 0xffffffffffff0000|(0x100) 等值，就可以栈溢出了。

core_write()

向全局变量 name 上写，这样通过 core_write() 和 core_copy_func() 就可以控制 ropchain 了。

三、漏洞及利用分析—ROP方法

1.利用思路

(1)获取 commit_creds()，prepare_kernel_cred() 的地址：/tmp/kallsyms 中保存了这些地址，可以直接读取，同时根据偏移固定也能确定 gadgets 的地址。

(2)通过 ioctl 设置 off，然后通过 core_read() leak 出 canary

(3)通过 core_write() 向 name 写，构造 ropchain

(4)通过 core_copy_func() 从 name 向局部变量上写，通过设置合理的长度和 canary 进行 rop

(5)通过 rop 执行 commit_creds(prepare_kernel_cred(0))

(6)返回用户态，通过 system("/bin/sh") 等起 shell

2.返回用户态栈布置

进内核态之前做的事情：swapgs、交换栈顶、push保存各种寄存器，我们只需要关注进内核时前5个push(因为第5个是ip寄存器)。

进入内核时保存的前5个寄存器

在着陆时(返回用户态时)执行swapgs; iretq，之前说过需要设置 cs, rflags 等信息，可以写一个函数保存这些信息。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！

exp中保存关键寄存器的asm汇编代码

3.找gadget方法

找到rop gadget之后根据/tmp/kallsyms 中读取的函数地址加上固定偏移即可得到gadget地址。

a. ROPper—Ropper

ropper --file ./vmlinux --nocolor > g1

b. ROPgadget—ROPgadget

ROPgadget --binary ./vmlinux > g2

c. IDA

vmlinux放进IDA；search->text；勾选Match Case/ Search Up/ Find all occurences；搜swaps和iretq。

找到一个可用的swaps：swaps; popfq; retn。

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。

4.exploit见exploit_rop.py

编译并打包运行

成功执行

四、漏洞及利用分析—ret2user方法

1.exploit

见exploit_ret2user.py。

2.ret2usr 原理

原理：攻击利用了 用户空间的进程不能访问内核空间，但内核空间能访问用户空间 这个特性来定向内核代码或数据流指向用户控件，以 ring 0 特权执行用户空间代码完成提权等操作。

方法：攻击程序中定义函数commit_creds(prepare_kernel_cred(0))，内核溢出时直接执行用户代码，不需要在构造内核的ROP。

注意：get_root()提权函数，函数指针的正确声明！

直接commit_creds(prepare_kernel_cred(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；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。

3.ROP与ret2user异同

(1)通过读取 /tmp/kallsyms 获取 commit_creds 和 prepare_kernel_cred 的方法相同，同时根据这些偏移能确定 gadget 的地址。

(2)leak canary 的方法也相同，通过控制全局变量 off 读出 canary。

(3)与 kernel rop 做法不同的是 rop 链的构造

a.kernel rop 通过 内核空间的 rop 链达到执行 commit_creds(prepare_kernel_cred(0))以提权目的，之后通过 swapgs; iretq 等返回到用户态，执行用户空间的 system("/bin/sh") 获取 shell

b.ret2usr 做法中，直接返回到用户空间构造的 commit_creds(prepare_kernel_cred(0))(通过函数指针实现)来提权，虽然这两个函数位于内核空间，但此时我们是 ring 0 特权，因此可以正常运行。之后也是通过 swapgs; iretq 返回到用户态来执行用户空间的 system("/bin/sh")

从这两种做法的比较可以体会出之所以要 ret2usr，是因为一般情况下在用户空间构造特定目的的代码要比在内核空间简单得多。