内核：系统调用设计

调用门

在GDT中找一空项，写入调用门

在自写的API中，返回要使用`retf`来返回并切换权限（最好用裸函数，自己平栈）

在R3中使用`call selector:0`来调用，其中选择子要符合其格式

缺点：GDT表项个数有限，使用麻烦，参数不定

改进：通过eax来传递函数的编号，调用门为总函数，将参数的地址放入寄存器中，在R0中自己拷贝参数

调用门描述符

ParamCount为参数个数，需要传参的时候要指明此字段的值，系统会自动将参数拷贝到内核栈上

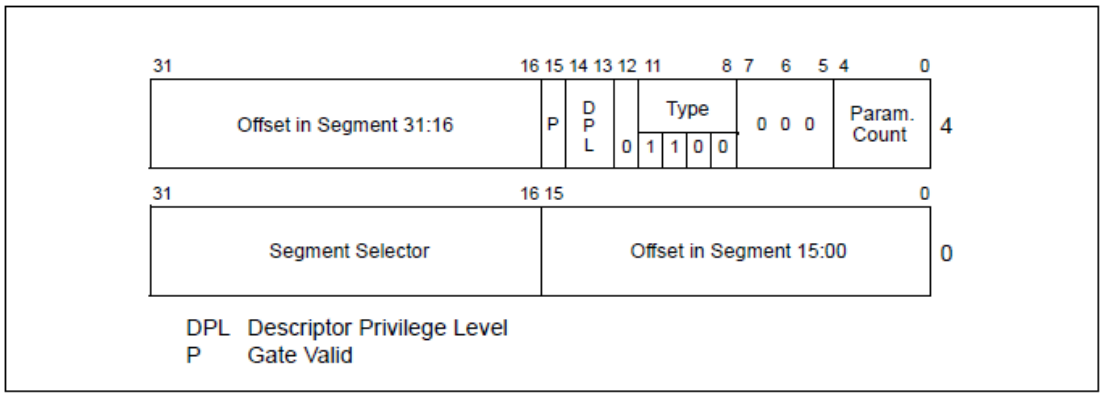

Figure 5-8. Call-Gate Descriptor

调用调用门之后栈的分布情况

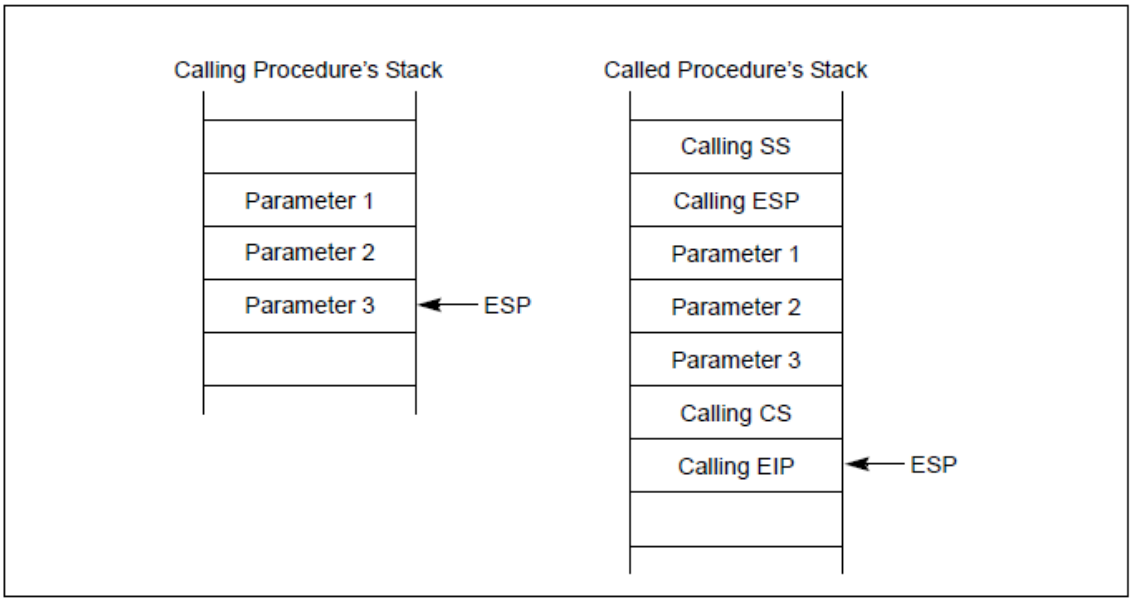

Figure 5-13. Stack Switching During an Interprivilege-Level Call

恢复环境只还原：`eip`、`cs`、`esp`、`ss`，需要手动还原 `fs`

实现步骤

R0

1. 修改中断表项（每个核都需要修改）

```
void SetCallGate(int index)
{
    //修改中断表
    char szGDT[6];
    int nShift = 1;
    GATE* pGDT = NULL;

    for (int i = 0; i < 2; i++) {
        KeSetSystemAffinityThread(nShift);  // 设置运行核心

        // 获取GDT
        __asm {
            sgdt szGDT
        }
        pGDT = *(GATE **)(szGDT + 2);

        // 设置调用门描述符
        GATE gate = { 0 };
        gate.p = 1;
        gate.dpl = 3;
        gate.type = 12;
        gate.s = 0;
        gate.sel = 8;
        gate.low  = (int)SysCall & 0xffff;
        gate.hei  = (int)SysCall >> 16;
        gate.param = 0;  //参数数量

        // 修改
```

```
        pGDT[index] = gate;

        nShift <<= 1;
    }
}
```

2. 实现系统调用总入口函数（裸函数）

```
typedef void (*SYSCALL_FUN)();
// 系统调用数组，存储函数指针
SYSCALL_FUN g_Syscall[] = { SysCall1 , SysCall2, (SYSCALL_FUN)SysCall3,
(SYSCALL_FUN)SysCall4 };
// 存储每个系统调用的参数总大小，用于从R3地址拷贝到R0上
char g_SyscallParam[] = { 0 , 0, 4, 8 };

__declspec(naked) void SysCall()
{
    __asm {
        pushad
        movzx ecx, g_SyscallParam[eax]  // 获取参数总大小，用于拷贝，eax相当于系统
调用号
        sub esp, ecx    // 分配参数空间
        mov esi, edx    // edx存储参数在R3中的地址
        mov edi, esp
        rep movsb       // 拷贝到R0上
        call g_Syscall[eax * 4]     // 调用相应的系统调用函数
        popad
        retf
    }
}
```

3. 实现各个系统调用函数

```
void SysCall1()
{
    KdPrint(("[51asm] SysCall1\n"));
}

void SysCall2()
{
    KdPrint(("[51asm] SysCall2\n"));
}

void SysCall3(int p1)
{
    KdPrint(("[51asm] SysCall3:%d\n", p1));
}

void SysCall4(int p1, int p2)
{
    KdPrint(("[51asm] SysCall4 p1=%d p2:%d\n", p1, p2));
}
```

R3

1. 为每个系统调用提供在R3的封装（裸函数）

```
__declspec(naked) void Syscall()
{
    __asm {
        // call selector:0的硬编码
        _emit 09ah
        _emit 00
        _emit 00
        _emit 00
        _emit 00
        _emit 04bh
        _emit 00

        // 恢复fs，此值需要提前获取
        mov ax, 3bh
        mov fs, ax
        ret
    }
}

__declspec(naked) void Syscall1()
{
    __asm {
        mov eax, 0
        lea edx, [esp + 4]  // 获取参数的地址
        jmp Syscall
    }
}

__declspec(naked) void Syscall2()
{
    __asm {
        mov eax, 1
        lea edx, [esp + 4]  // 获取参数的地址
        jmp Syscall
    }
}

__declspec(naked) void Syscall3(int p1)
{
    __asm {
        mov eax, 2
        lea edx, [esp + 4]  // 获取参数的地址
        jmp Syscall
    }
}

__declspec(naked) void Syscall4(int p1, int p2)
{
    __asm {
        mov eax, 3
        lea edx, [esp + 4]  // 获取参数的地址
        jmp Syscall
    }
}
```

2. 调用系统调用

```
Syscall1();
Syscall2();
Syscall3(3);
Syscall4(3, 4);
```

陷阱门（软中断、陷阱）

windows系统调用都使用陷阱门来实现

在IDT中找一空项，写入陷阱门（因为系统调用属于软中断，又叫陷阱中断）

使用`int xxx`调用，中断返回要用`iret`，方法与之前的调用门类似

windows利用`int 2e`来调用`nt!KiSystemService`，且内部做了一张系统服务表（SSDT）

SSDT

```
typedef struct SSDT {
    unsigned int func_arr_ptr;      // api数组的指针
    unsigned int reserve;           // 保留
    unsigned int count;             // 数组元素个数
    unsigned char param_arr_ptr;    // api参数数组指针
} SSDT, ShadowSSDT;
```

- 一个SSDT（普通的API）
  - 方法一：KTHREAD结构的ServiceTable成员
  - 方法二：导出的`KeServiceDescriptorTable`全局变量（在64位上可能此变量不导出）
- 一个ShadowSSDT（UI相关API）

用途：主动防御，监控API，且有两张表，这两张表相邻

以win7 32位为例子，在windbg上查看`KeServiceDescriptorTable`

```
0: kd> dd KeServiceDescriptorTable
83fbe9c0  83ed2d9c 00000000 00000191 83ed33e4   -> KeServiceDescriptorTable：
SSDT
83fbe9d0  00000000 00000000 00000000 00000000
83fbe9e0  83f316af 00000000 020d96db 0000005d
83fbe9f0  00000011 00000100 5385d2ba d717548f
83fbea00  83ed2d9c 00000000 00000191 83ed33e4   ->
KeServiceDescriptorTableShadow：SSDT，同上一致
83fbea10  990b6000 00000000 00000339 990b702c   ->
KeServiceDescriptorTableShadow：ShadowSSDT
83fbea20  00000000 00000000 83fbea24 00000340
83fbea30  00000340 86ced518 00000007 00000000
```

https://bbs.pediy.com/thread-98909.htm

兼容任意版本

需要解析PDB文件，`ntkrnlmp.pdb`和`win32k.pdb`

MSR

MSR寄存器：模式指令寄存器，是一组寄存器，以编号来命名

相关指令

- `wrmsr`
- `rdmsr`
- `sysenter`进R0
- `sysexit`回R3

```
MOV ECX, 08BH ;IA32_BIOS_SIGN_ID
XOR EAX, EAX ;clear EAX
XOR EDX, EDX ;clear EDX
WRMSR ;Load 0 to MSR at 8BH
MOV EAX, 1
cpuid
MOV ECX, 08BH ;IA32_BIOS_SIGN_ID
rdmsr ;Read Model Specific Register
```

进R0的方式不一样，实现调用的本质还是查表拷贝参数

SYSENTER和SYSEXIT

> 对于SYSENTER，需要R0的信息：
> • CS0 — IA32_SYSENTER_CS.
> • EIP0 — IA32_SYSENTER_EIP.
> • SS0 — IA32_SYSENTER_CS + 8.
> • ESP0 — IA32_SYSENTER_ESP.
>
> 对于SYSEXIT，需要R3的信息：
> • CS — IA32_SYSENTER_CS + 16.
> • EIP — EDX.
> • SS — IA32_SYSENTER_CS + 24.
> • ESP — ECX.
>
> 所以，在R3进R0之前会将R3的EIP和ESP分别存入EDX和ECX，而后R0将ECX压栈，由于R3是call进来的，栈上有返回地址，故在R0返回R3时，执行POP ECX; POP EDX即可

在新版本中，基本上都使用此方式进入R0与回R3

在64位上，以`syscall`和`sysret`来完成进R0和回R3（在64位下对SSDT进行了加密，对于每个项拿出来右移4位+表基址，主要是为了防止SSDT Hook）

`swapgs`交换gs寄存器（在64位下代替fs寄存器）

KPP保护：对内核模块代码进行保护，修改直接蓝屏