天书夜读

（试读版）

汇编语言是一门本来是很基础很古老的语言，由于它的代码可读性可移植性较差，现在已经很少有人用。但它的优点也是显而易见的，很高的效率，不受编译器限制的随意性，对硬件的直接操作（保护模式下需要系统支持），以及逆向工程时不可或缺的反汇编调试等。随着你越来越深入的了解计算机，你会越来越觉得这个古老的东西是最重要的，是那些时髦的编程语言不可比拟的。

我们每天使用的 Windows 内核部分,几乎完全用 C 语言开发。只可惜 MS 并不公开源代码。

虽然如此，却没有谁能阻止你看汇编的代码。MS 对 Windows 既没有加壳加密，也没有花指令，洋洋洒洒上千万行代码，数千精英程序员的智慧结晶，如今就在你的电脑内。工作结束，夜晚无聊之时，难道不想读一读，这深不可测的浩瀚天书吗?

以前曾经有人把 Windows 的 dll 反汇编后改写为 C 语言的资料。后来又有 WindowsNT 和 2000 的源代码泄漏。不过到如今 vista 都已经发布了。想要自己随心所欲的阅读，还是要自己掌握 Windows 程序的汇编写法吧。获人之鱼，不如师人之渔。

本文知识初浅，并不是能用于破解，反工程，或者编写病毒等的高级技术。仅供读者业余聊以自娱之用.这本书的文字部分主要由楚狂人编写。部分代码和技术细节由 wowocock 提供.

现在本版本为完整的试读版本，包括完整的 1-3 节。今后如果需要新的章节，将按读者的意见，加入到正式版本中。

楚狂人,wowocock

2007 年 1 月 于 上海

目录

第一节 入手：基本 C 反汇编

1-1. 函数与调用栈

函数和堆栈的关系密切，这是因为：我们通过堆栈把参数从外部传入到内部。此外，我们在堆栈中划分区域来容纳函数的内部变量。

调用 push 和 pop 指令的时候，寄存器 esp 用于指向栈顶的位置。栈顶总是栈中地址最小的位置。push 执行的结果,esp 总是减少。pop 则增加。

C 语言所写的程序中，堆栈用于传递函数参数。这时称为调用栈。

写一个简单的函数如下:

```
void myfunction(int a,int b)
{
    int c = a+b;
}
```

这是标准的 C 函数调用方式.其过程是:

1) 调用方把参数反序的压入堆栈中。
2) 调用函数。
3) 调用方把堆栈复原。

而被调用函数需要做以下一些事情:

1) 保存 ebp. ebp 总是被我们用来保存这个函数执行之前的 esp 的值。执行完毕之后，我们用 ebp 恢复 esp.同时，调用我们的上层函数也用 ebp 做同样的事情。所以我们之前先把 ebp 压入堆栈。返回之前弹出，避免 ebp 被我们改动。
2) 保存 esp 到 ebp 中。

上面两步的代码如下:

```
push ebp                ;保存 ebp,并把 esp 放入 ebp 中
                        ;此时 ebp 与 esp 同。
mov ebp,esp             ;都是这次函数调用时的栈顶。
```

3) 在堆栈中腾出一个区域用来保存局部变量。这就是常说的所谓局部变量在栈空间中。方法为把 esp 减少一个数值。这样等于压入了一堆变量。日后要恢复时，只要把 esp 恢复成 ebp 中保存的数据就可以了.
4) 保存 ebx,esi,edi 到堆栈中。函数调用完后恢复。这是一个编程规范。

对应的代码如下:

```
sub esp,0cch            ;把 esp 往上移动一个范围，等于在堆栈中放出一片新
                        ;的空间用来存局部变量.
push ebx                ;下面保存三个寄存器:ebx,esi,edi,这也是 C 规范.
push esi
```

```
push edi
```

5) 把局部变量区域初始化成 0cccccccch。0cccccccch 实际是 INT 3.这是一个中断指令。因为局部变量不可能被执行，如果执行了必然程序有错，这时发生中断来提示开发者。这是 VC 编译 DEBUG 版本的特有操作。

相关代码如下:

```
lea edi,[ebp-0cch]          ;本来是要 mov edi,ebp-0cch,但是 mov 不支持-操作。所
                            ;以对 ebp-0cch 取内容，而 lea 把内容的地址也就是 ebp
                            ;-0cch 加载到 edi 中.目的是把保存局部变量的区域（从
                            ;ebp-0cch 开始的区域）初始化成全部 0cccccccch.
mov ecx,33h
mov eax,0cccccccch
rep stos dword ptr [edi]        ;拷贝字符串
```

6) 然后做函数里应该做的事情。参数的获取是 ebp+8 字节为第一个参数，ebp+12 为第二个参数，依次增加。ebp+4 字节处是要返回的地址。

7) 恢复 ebx,esi,edi ,esp,ebp,最后返回。

代码如下:

```
pop edi                     ;恢复 edi,esi,ebx
pop esi
pop ebx
mov esp,ebp                 ;恢复原来的 ebp 和 esp,让上一个调用的函数正常使用.
pop ebp
ret
```

为了简单起见，我的函数没有返回值。如果要返回值，函数应该在返回之前，把返回值放入 eax 中。外部通过 eax 得到返回值。

所以用 VC2003 编译 Debug 版本，反汇编代码如下：

```
void myfunction(int a,int b)
{
push ebp                    ;保存 ebp,并把 esp 放入 ebp 中。此时 ebp 与 esp 同。
mov ebp,esp                 ;都是这次函数调用时的栈顶。
sub esp,0cch                ;把 esp 往上移动一个范围，等于在堆栈中放出一片新
                            ;的空间用来存局部变量.
push ebx                    ;下面保存三个寄存器:ebx,esi,edi,这也是 C 规范.
push esi
push edi
lea edi,[ebp-0cch]          ;本来是要 mov edi,ebp-0cch,但是 mov 不支持-操作。所
                            ;以对 ebp-0cch 取内容，而 lea 把内容的地址也就是 ebp
                            ;-0cch 加载到 edi 中.目的是把保存局部变量的区域（从
                            ;ebp-0cch 开始的区域）初始化成全部 0cccccccch.
mov ecx,33h
mov eax,0cccccccch
rep stos dword ptr [edi]        ;拷贝字符串
    int c = a+b;
```

```
mov eax,dword ptr [a]          ;简单的相加操作.这里从堆栈中取得从外部传入的参数。那么
add eax,dword ptr[b]           ;a 和 b 到底是怎么取得的呢，通过 ida 反汇编可以看到，其实
                               ;这两条指令是 mov eax, [ebp+8] ，add eax, [ebp+0Ch]，参数是
                               ;通过 ebp 从堆栈中取得的。这里看到的是 VC 调试器的显示结
                               ;果，为了阅读方便直接加上了参数名。
mov dword ptr[c],eax
}
pop edi                        ;恢复 edi,esi,ebx
pop esi
pop ebx
mov esp,ebp                    ;恢复原来的 ebp 和 esp,让上一个调用的函数正常使用.
pop ebp
ret
```

而这个函数的调用方式是:

```
mov eax,dword ptr[b]           ; 把 b,a 两个参数压入堆栈
push eax
mov ecx,dword ptr[a]
push ecx
call myfunction                ; 调用函数 myfunction.
add esp,8                          ; 恢复堆栈.
```

这样一来，函数调用的过程就很清楚了。

1-2. 循环

下面我把函数改得复杂一点，增加了一个循环，来看反汇编的结果:

```
int myfunction(int a,int b)
{
    int c = a+b;
    int i;
    for(i=0;i<50;i++)
    {
        c = c+i;
    }
    return c;
}
```

前面的反汇编结果和前一节的一样了，现在从 for 的地方开始反汇编，结果如下:

```
for(i=0;i<50;i++)
00412BC7   mov          dword ptr [i],0
00412BCE   jmp          myfunction+39h (412BD9h)
```

```
00412BD0   mov          eax,dword ptr [i]
00412BD3   add          eax,1
00412BD6   mov          dword ptr [i],eax
00412BD9   cmp          dword ptr [i],32h
00412BDD   jge          myfunction+4Ah (412BEAh)
 {
      c = c+i;
00412BDF   mov          eax,dword ptr [c]
00412BE2   add          eax,dword ptr [i]
00412BE5   mov          dword ptr [c],eax
 }
00412BE8   jmp          myfunction+30h (412BD0h)
 00412BEA   mov          eax,dword ptr [c]
}
```

… 后面省略的部分也和上一节相同。

可以看到循环主要用这么几条指令来实现:mov 进行初始化。jmp 跳过循环变量改变代码。cmp 实现条件判断，jge 根据条件跳转。用 jmp 回到循环改变代码进行下一次循环。所以结构.大体如下:

```
    mov <循环变量>,<初始值>              ;给循环变量赋初值
    jmp  B                               ;跳到第一次循环处
A:  (改动循环变量)                       ;修改循环变量。
    …
B:  cmp <循环变量>,<限制变量>            ;检查循环条件
    jgp  跳出循环
    (循环体)
    …
    jmp A                                ;跳回去修改循环变量
```

我们再看一下 do 循环，因为 do 循环没有修改循环变量的部分，所以比 for 循环要简单一些。

```
do {
     c = c+i;
00411A55   mov          eax,dword ptr [c]
00411A58   add          eax,dword ptr [i]
00411A5B   mov          dword ptr [c],eax
} while(c< 100);
00411A5E   cmp          dword ptr [c],64h
00411A62   jl           myfunction+35h (411A55h)
return c;
```

…

do 循环就是一个简单的条件跳转回去。只有两条指令：

```
cmp <循环变量>,<限制变量>
jl    <循环开始点>
```

下面看看 while 的情况:

```
    while(c<100){
00411A55   cmp          dword ptr [c],64h
00411A59   jge          myfunction+46h (411A66h)
        c = c+i;
00411A5B   mov          eax,dword ptr [c]
00411A5E   add          eax,dword ptr [i]
00411A61   mov          dword ptr [c],eax
    }
00411A64   jmp          myfunction+35h (411A55h)
```

```
return c;
```

你会发现 while 要更复杂一点。因为 while 除了开始的时候判断循环条件之外，后面还必须有一条无条件跳转回到循环开始的地方，共用三条指令实现：

```
A:  cmp <循环变量>,<限制变量>
    jge  B
    (循环体)
    …
    jmp A
B:  (循环结束了)
```

1-3. 判断与分支

写一个简单的 if 判断结构：

```
if(c>0 && c<10)
{
    printf("c>0");
}
else if( c>10 && c<100)
{
    printf("c>10 && c<100");
}
else
{
    printf("c>10 && c<100");
}
```

if 判断都是使用 cmp 再加上条件跳转指令。对于 if(A && B)的情况，一般都是使用否决法。如果 A 不成立，立刻跳下一个分支。依次，如果 B 不成立，同样跳下一分支。：

```
cmp <条件>
jle   <下一个分支>
```

所以开始部分的反汇编为:

```
if(c>0 && c<10)
00411A66   cmp          dword ptr [c],0
00411A6A   jle          myfunction+61h (411A81h)     ; 跳下一个 else if 的判断点
00411A6C   cmp          dword ptr [c],0Ah
00411A70   jge          myfunction+61h (411A81h)     ; 跳下一个 else if 的判断点
{
    printf("c>0");
00411A72   push         offset string "c>0" (4240DCh)
00411A77   call         @ILT+1300(_printf) (411519h)
00411A7C   add          esp,4
}
```

else if 的和 else 的特点是，开始都有一条无条件跳转到判断结束处，阻止前面的分支执行结束后，直接进入这个分支。这个分支能执行到的唯一途径只是,前面的判断条件不满足。

else 则在 jmp 之后直接执行操作。而 else if 则开始重复 if 之后的操作，用 cmp 比较，然后用条件跳转指令进行跳转.

```
else if( c>10 && c<100)
00411A7F   jmp          myfunction+89h (411AA9h)        ;直接跳到判断块外
00411A81   cmp          dword ptr [c],0Ah               ;比较+条件跳转，目标为下一个分支处
00411A85   jle          myfunction+7Ch (411A9Ch)
00411A87   cmp          dword ptr [c],64h
00411A8B   jge          myfunction+7Ch (411A9Ch)
{
    printf("c>10 && c<100");
00411A8D   push         offset string "c>10 && c<100" (424288h)
00411A92   call         @ILT+1300(_printf) (411519h)
00411A97   add          esp,4
}
else
00411A9A   jmp          myfunction+89h (411AA9h)        ;这里是 else,所以只有简单的一条跳转。
{
    printf("c>10 && c<100");
00411A9C   push         offset string "c>10 && c<100" (424288h)
00411AA1   call         @ILT+1300(_printf) (411519h)
00411AA6   add          esp,4
}
return c;
```

条件分支中，有比较特殊的情况是 switch.switch 的特点是有多个判断。因为 swtich 显然不用判断大于小于，所以都是 je,分别跳到每个 case 处。最后一个是无条件跳转，直接跳到 default 处。以下的代码:

```
switch(c)
{
```

```
case 0:
    printf("c>0");
case 1:
    {
        printf("c>10 && c<100");
        break;
    }
default:
    printf("c>10 && c<100");
}
```

反汇编的结果是:

```
switch(c)
00411A66   mov          eax,dword ptr [c]
00411A69   mov          dword ptr [ebp-0E8h],eax
00411A6F   cmp          dword ptr [ebp-0E8h],0
00411A76   je           myfunction+63h (411A83h)
00411A78   cmp          dword ptr [ebp-0E8h],1
00411A7F   je           myfunction+70h (411A90h)
00411A81   jmp          myfunction+7Fh (411A9Fh)
{
…
```

显然是比较 c 是否是 0，1，这两个数字。至于先把 c 移动到 ebp-0E8h 这个地址，然后再比较，这是调试版本编译的特点。可能是为了防止直接操作堆栈而导致堆栈破坏？最后一条直接跳转到 default 处。如果没有 default，就跳到 swtich 之外。

```
case 0:
    printf("c>0");
00411A83   push         offset string "c>0" (4240DCh)
00411A88   call         @ILT+1300(_printf) (411519h)
00411A8D   add          esp,4
case 1:
    {
        printf("c>10 && c<100");
00411A90   push         offset string "c>10 && c<100" (424288h)
00411A95   call         @ILT+1300(_printf) (411519h)
00411A9A   add          esp,4
        break;
00411A9D   jmp          myfunction+8Ch (411AACh)
    }
default:
    printf("c>10 && c<100");
```

```
00411A9F   push          offset string "c>10 && c<100" (424288h)
00411AA4   call          @ILT+1300(_printf) (411519h)
00411AA9   add           esp,4
}
```

至于 case 和 default 都非常简单。如果有 break,则会增加一个无条件跳转。没有 break 的情况下，没有任何循环控制代码。

下面是一个练习，内容是把下面的汇编代码还原成 c 语言。这种练习非常有用处。

```
00411A20   push          ebp
00411A21   mov           ebp,esp
00411A23   sub           esp,0E8h
00411A29   push          ebx
00411A2A   push          esi
00411A2B   push          edi
00411A2C   lea           edi,[ebp-0E8h]
00411A32   mov           ecx,3Ah
00411A37   mov           eax,0CCCCCCCCh
00411A3C   rep stos      dword ptr [edi]
00411A3E   mov           eax,dword ptr [a]
00411A41   add           eax,dword ptr [b]
00411A44   mov           dword ptr [d],eax
00411A47   mov           dword ptr [i],1
00411A4E   mov           dword ptr [c],0
00411A55   cmp           dword ptr [c],64h
00411A59   jge           myfunction+46h (411A66h)
00411A5B   mov           eax,dword ptr [c]
00411A5E   add           eax,dword ptr [i]
00411A61   mov           dword ptr [c],eax
00411A64   jmp           myfunction+35h (411A55h)
00411A66   mov           eax,dword ptr [c]
00411A69   mov           dword ptr [ebp-0E8h],eax
00411A6F   cmp           dword ptr [ebp-0E8h],0
00411A76   je            myfunction+63h (411A83h)
00411A78   cmp           dword ptr [ebp-0E8h],1
00411A7F   je            myfunction+6Ah (411A8Ah)
00411A81   jmp           myfunction+72h (411A92h)
00411A83   mov           dword ptr [d],1
00411A8A   mov           eax,dword ptr [c]
00411A8D   mov           dword ptr [d],eax
00411A90   jmp           myfunction+79h (411A99h)
00411A92   mov           dword ptr [d],0
00411A99   mov           eax,dword ptr [d]
00411A9C   pop           edi
00411A9D   pop           esi
00411A9E   pop           ebx
00411A9F   mov           esp,ebp
```

```
00411AA1   pop          ebp
00411AA2   ret
```

1-4. 数组与结构

写一个简单的函数，用到结构体和数组。

```
typedef struct {
    int a;
    int b;
    int c;
} mystruct;
int myfunction(int a,int b)
{
    unsigned char *buf[100];
    mystruct *strs = (mystruct *)buf;
    int i;
    for(i=0;i<5;i++)
    {
        strs[i].a = 0;
        strs[i].b = 1;
        strs[i].c = 2;
    }
    return 0;
}
```

对结构体和数组的访问是我们感兴趣的地方。相关的反汇编结果是这样的:

```
int i;
for(i=0;i<5;i++)
0041367A   mov          dword ptr [i],0                 ;典型的 for 循环过程
00413684   jmp          myfunction+45h (413695h)
00413686   mov          eax,dword ptr [i]
0041368C   add          eax,1
0041368F   mov          dword ptr [i],eax
00413695   cmp          dword ptr [i],5
0041369C   jge          myfunction+94h (4136E4h)
{
    strs[i].a = 0;
0041369E   mov          eax,dword ptr [i]               ;目的是把 i*0ch 放入到 eax 中。
004136A4   imul         eax,eax,0Ch                     ;0Ch 是结构的大小。
004136A7   mov          ecx,dword ptr [strs]                ;把 strs 的地址放入 ecx
004136AD   mov          dword ptr [ecx+eax],0           ;计算得到 strs[i]的地址，并赋 0
    strs[i].b = 1;
004136B4   mov          eax,dword ptr [i]
004136BA   imul         eax,eax,0Ch
```

```
004136BD   mov          ecx,dword ptr [strs]
004136C3   mov          dword ptr [ecx+eax+4],1              ;这里与.a 不同，增加偏移取得 b 的位置
    strs[i].c = 2;
004136CB   mov          eax,dword ptr [i]
004136D1   imul         eax,eax,0Ch
004136D4   mov          ecx,dword ptr [strs]
004136DA   mov          dword ptr [ecx+eax+8],2
}
004136E2   jmp          myfunction+36h (413686h)        ;典型的循环结束
004136E4   xor          eax,eax                         ;eax 清 0
…
```

1-5. 共用体，枚举类型

不过你可能会感到失望。因为共用体和枚举类型都是在 C 语言中为了让内容更加易读而引入的东西。实际上，只要有结构体和基本的数据类型就足够了。所以在汇编中，这些多余的东西都消失不见了。

为了实验一下我写一段有共用体和枚举类型的代码。然后反汇编一下看看效果。

```
// 定义一个枚举类型
typedef enum {
ENUM_1 = 1,
ENUM_2 = 2,
ENUM_3,
ENUM_4,
} myenum;

// 定义一个结构体
typedef struct {
    int a;
    int b;
    int c;
} mystruct;

typedef union {
    mystruct s;
    myenum e[3];
} myunion;

int myfunction(int a,int b)
{
    unsigned char buf[100] = { 0 };
    myunion *uns = (myunion *)buf;
    int i;
    // 访问共用体，结构体，使用枚举类型的变量
```

```
    for(i=0;i<5;i++)
    {
        uns[i].s.a = 0;
        uns[i].s.b = 1;
        uns[i].e[2] = ENUM_4;
    }
    return 0;
}
```

反汇编的结果和上一小节基本没有区别:

```
for(i=0;i<5;i++)
00411A57   mov          dword ptr [i],0
00411A5E   jmp          myfunction+49h (411A69h)
00411A60   mov          eax,dword ptr [i]
00411A63   add          eax,1
00411A66   mov          dword ptr [i],eax
00411A69   cmp          dword ptr [i],5
00411A6D   jge          myfunction+83h (411AA3h)
{
    uns[i].s.a = 0;
00411A6F   mov          eax,dword ptr [i]              ;基本没有区别。直接去找.a 的地址。
00411A72   imul         eax,eax,0Ch
00411A75   mov          ecx,dword ptr [uns]
00411A78   mov          dword ptr [ecx+eax],0
    uns[i].s.b = 1;
00411A7F   mov          eax,dword ptr [i]
00411A82   imul         eax,eax,0Ch
00411A85   mov          ecx,dword ptr [uns]
00411A88   mov          dword ptr [ecx+eax+4],1
    uns[i].e[2] = ENUM_4;
00411A90   mov          eax,dword ptr [i]
00411A93   imul         eax,eax,0Ch
00411A96   mov          ecx,dword ptr [uns]
00411A99   mov          dword ptr [ecx+eax+8],4
}
00411AA1   jmp          myfunction+40h (411A60h)
```

汇编里一切花哨的东西都消失不见了。和单纯用结构体没有区别。用枚举类型和一般常数也没有任何区别，这是编译器处理的结果。

1-6. 算法的反汇编

C 语言的各种控制流程在反汇编中绝对是最让人舒服的部分。而最痛苦的莫过算法的部分。一个简单的算法就会让复杂到让人眼花缭乱的程度，更不用说复杂的算法了。

显然没有完美的办法来解决这个问题，只能靠你的耐心和智力了。下面举一个 3*3 矩阵相乘的例子。

```
int myfunction(int a[3][3],int b[3][3],int c[3][3])
{
    int i,j;
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        c[i][j] = a[i][0]*b[0][j]+a[i][1]*b[1][j]+a[i][2]*b[2][j];
    }
    return 0;
}
```

这个代码很简单易懂，不过汇编的写法真的很富有挑战性:

```
int i,j;
for(i=0;i<3;i++)
00411A3E   mov          dword ptr [i],0
00411A45   jmp          myfunction+30h (411A50h)
00411A47   mov          eax,dword ptr [i]
00411A4A   add          eax,1
00411A4D   mov          dword ptr [i],eax
00411A50   cmp          dword ptr [i],3
00411A54   jge          myfunction+0AEh (411ACEh)
{
    for(j=0;j<3;j++)
00411A56   mov          dword ptr [j],0
00411A5D   jmp          myfunction+48h (411A68h)
00411A5F   mov          eax,dword ptr [j]
00411A62   add          eax,1
00411A65   mov          dword ptr [j],eax
00411A68   cmp          dword ptr [j],3
00411A6C   jge          myfunction+0A9h (411AC9h)
            c[i][j] = a[i][0]*b[0][j]+a[i][1]*b[1][j]+a[i][2]*b[2][j];
00411A6E   mov          eax,dword ptr [i]                    ;从这里开始的指令只有 mov,add 和
00411A71   imul         eax,eax,0Ch                          ;imul,尝试把它们还原成表达式吧。
00411A74   mov          ecx,dword ptr [a]
00411A77   mov          edx,dword ptr [j]
00411A7A   mov          esi,dword ptr [b]
00411A7D   mov          eax,dword ptr [ecx+eax]
00411A80   imul         eax,dword ptr [esi+edx*4]
00411A84   mov          ecx,dword ptr [i]
00411A87   imul         ecx,ecx,0Ch
00411A8A   mov          edx,dword ptr [a]
00411A8D   mov          esi,dword ptr [j]
00411A90   mov          edi,dword ptr [b]
```

```
00411A93   mov          ecx,dword ptr [edx+ecx+4]
00411A97   imul         ecx,dword ptr [edi+esi*4+0Ch]
00411A9C   add          eax,ecx
00411A9E   mov          edx,dword ptr [i]
00411AA1   imul         edx,edx,0Ch
00411AA4   mov          ecx,dword ptr [a]
00411AA7   mov          esi,dword ptr [j]
00411AAA   mov          edi,dword ptr [b]
00411AAD   mov          edx,dword ptr [ecx+edx+8]
00411AB1   imul         edx,dword ptr [edi+esi*4+18h]
00411AB6   add          eax,edx
00411AB8   mov          ecx,dword ptr [i]
00411ABB   imul         ecx,ecx,0Ch
00411ABE   add          ecx,dword ptr [c]
00411AC1   mov          edx,dword ptr [j]
00411AC4   mov          dword ptr [ecx+edx*4],eax
00411AC7   jmp          myfunction+3Fh (411A5Fh)
}
00411AC9   jmp          myfunction+27h (411A47h)
```

要阅读这样的代码，首先把流程控制的代码与数值计算的代码分开是关键。因此你前面的练习能起很大的帮助。当你得到值计算的代码的部分后，你必须判断输入与输出（一般自然是被读的内部变量为输入，被写的内部变量为输出），然后把它还原成一个 C 语言的表达式。任何一段中间不加任何跳转，连续的 mov 和加减乘除的指令一般都可以还原为一个 C 表达式。当然，这可不是一个轻松的工作。

在这里顺便可以看到，二维数组 a[x][y],处理等同与一个大小为 a[y]的结构的长度大小为 x 的数组。所以，前面讲到的数组访问的代码非常有价值。基本的方法如下:

```
mov         eax,<我要取的数组元素的下标>
imul        eax,eax,<结构的大小>
mov         ecx,<结构数组开始的地址>
mov          eax,dword ptr [ecx+eax]          ;取得数组元素的内容，放到 eax 中.
```

访问结构内部变量的时候最后面一个指令还会加上一个数字：

```
mov           eax,dword ptr[ecx+eax+0Ch]
```

看到这样的代码，我们应该联想到表达式中含有的数组或结构体。

1-7. 发行版的反汇编

前面费了很多工夫解析代码。但是那都是调试版本的代码。使用他们是因为那样汇编代码会更好理解。实际上，到非调试版本的时候，编译器将进行非常多的优化，使汇编代码变得简单的同时，阅读的困难也大大增加了。但是实际上，你碰到的代码显然都是发行版本。

首先是函数调用的过程如下:

```
0040108D   push         eax
0040108E   push         esi
0040108F   call         myfunction1 (401000h)
00401094   add          esp,8
```

优化之后，两个参数根本没有放入内部变量中，而是放在 eax 和 esi 中直接 push 了。然后调用 myfunction1.最后是恢复 esp，与调试版本的情况相同。

然后是函数的执行过程，在堆栈中划分内部区域的代码可能被省略。因为内部变量比较少的情况，都直接使用寄存器了。把内部变量初始化为 INT 3 的代码也被省略。取参数也不会放入内部变量中，而是用 esp 直接取。同时，ebp 根本没被使用。ebx，esi 和 edi 的压入弹出是函数调用和结束最明显的标记。无论调试版本还是发行版本都是如此.

下面是一个循环的简单函数:

```
int myfunction1(int a,int b)

{
00401000   push         ebx                    ;保存 ebx,esi,edi
00401001   push         esi
int i;
for(i=0;i<5;i++)
00401002   mov          esi,dword ptr [esp+0Ch]      ;     取第一个参数.本来第一个参数是 esp+4h,但是因为前面有
                                                    ;     两个 push,所以变成 esp+Ch.
00401006   push         edi
00401007   mov          edi,dword ptr [esp+14h]      ;     取第二个参数，计算方法同上
0040100B   xor          ebx,ebx                ;     清空 ebx,显然没有定义内部变量 i,而是直接用了 ebx.
0040100D   lea          ecx,[ecx]              ;     无意义的指令。
{
```

循环的结尾处是这样的:

```
00401020   inc          ebx
00401021   cmp          ebx,5
00401024   jl           myfunction1+10h (401010h)
```

可见优化后的 for 循环喜欢模仿相对简单的 do 循环的方式。把判断和跳转放在最后。

前面的矩阵相乘的例子如下：请注意这时候 c 代码和后面的汇编已经没有正确的对应关系了:

```
int myfunction(int a[3][3],int b[3][3],int c[3][3])

{
    int i,j;
    for(i=0;i<3;i++)
00401000   mov          eax,dword ptr [esp+4]        ;a 保存到 eax 中
00401004   mov          edx,dword ptr [esp+0Ch]          ;c 保存到 edx 中
00401008   mov          ecx,dword ptr [esp+8]        ;b 保存到 ecx 中。
0040100C   push         ebx
0040100D   push         esi
0040100E   add          eax,4                        ;这里已经开始处理数据。必须把这些指令和
                                                     ;函数入口指令和循环控制指令分开
00401011   push         edi
00401012   add          edx,8                        ;同前面的 add eax,4
00401015   mov          esi,3                        ;循环变量取 3
0040101A   lea          ebx,[ebx]                    ;无意义指令
    {
```

```
        for(j=0;j<3;j++)
            c[i][j] = a[i][0]*b[0][j]+a[i][1]*b[1][j]+a[i][2]*b[2][j];
00401020   mov          ebx,dword ptr [eax]          ;显然从这里开始,是数值计算过程。代码的优化，使内部
00401022   imul         ebx,dword ptr [ecx+0Ch]      ;循环不见了。可以数一下 imul 指令，刚好九次。可以说明
00401026   mov          edi,dword ptr [ecx+18h]      ;这里用一个单循环完成了原来双循环的工作。
00401029   imul         edi,dword ptr [eax+4]
0040102D   add          edi,ebx
0040102F   mov          ebx,dword ptr [eax-4]
00401032   imul         ebx,dword ptr [ecx]
00401035   add          edi,ebx
00401037   mov          dword ptr [edx-8],edi
0040103A   mov          ebx,dword ptr [eax]
0040103C   imul         ebx,dword ptr [ecx+10h]
00401040   mov          edi,dword ptr [ecx+1Ch]
00401043   imul         edi,dword ptr [eax+4]
00401047   add          edi,ebx
00401049   mov          ebx,dword ptr [eax-4]
0040104C   imul         ebx,dword ptr [ecx+4]
00401050   add          edi,ebx
00401052   mov          dword ptr [edx-4],edi
00401055   mov          ebx,dword ptr [eax+4]
00401058   imul         ebx,dword ptr [ecx+20h]
0040105C   mov          edi,dword ptr [ecx+14h]
0040105F   imul         edi,dword ptr [eax]
00401062   add          edi,ebx
00401064   mov          ebx,dword ptr [eax-4]
00401067   imul         ebx,dword ptr [ecx+8]
0040106B   add          edi,ebx
0040106D   mov          dword ptr [edx],edi
0040106F   add          eax,0Ch
00401072   add          edx,0Ch
00401075   dec          esi                          ;这里开始两条，是循环指令
00401076   jne          myfunction+20h (401020h)
00401078   pop          edi
00401079   pop          esi
    }
return 0;
0040107A   xor          eax,eax
0040107C   pop          ebx
}
```

1-8.汇编反 C 练习

下面是另外一个练习题，这个练习完全是发行版本的一个函数。如果能顺利完成这个练习，那么基本 C 语法的反汇编结果阅读也就入门了。

我们不公布 C 语言的源程序，但我们尝试把它还原成 C 语言。这次用的完全发行版本，经过 o2 优化的代码。足够接近实战了。下面的汇编语言程序对应一个完整的 C 函数。

```
00401000  push    ecx                          F
00401001  mov     ecx,dword ptr [esp+10h]      D
00401005  mov     edx,dword ptr [esp+8]        D
00401009  push    ebx                          F
0040100A  mov     ebx,dword ptr [esp+4]        D
0040100E  push    esi                          F
0040100F  mov     esi,dword ptr [esp+14h]      D
00401013  push    ebp                          F
00401014  xor     eax,eax                      C
00401016  push    edi                          F
00401017  jmp     myfunction+20h (401020h)     F
00401019  lea     esp,[esp]                    F
00401020  mov     edi,dword ptr [esi+8]        D
00401023  imul    edi,dword ptr [edx+eax*8+4]  D
00401028  mov     ebp,dword ptr [esi]          D
0040102A  imul    ebp,dword ptr [edx+eax*8]    D
0040102E  add     edi,ebp                      D
00401030  mov     dword ptr [ecx],edi          D
00401032  mov     edi,dword ptr [esi+0Ch]      D
00401035  imul    edi,dword ptr [edx+eax*8+4]  D
0040103A  mov     ebp,dword ptr [esi+4]        D
0040103D  imul    ebp,dword ptr [edx+eax*8]    D
00401041  add     edi,ebp                      D
00401043  mov     ebp,dword ptr [ecx]          D
00401045  add     ebp,edi                      D
00401047  add     ebx,ebp                      D
00401049  mov     dword ptr [ecx+4],edi        D
0040104C  inc     eax                          C
0040104D  add     ecx,8                        D
00401050  cmp     eax,2                        C
00401053  jl      myfunction+20h (401020h)     C
00401055  call    rand (401138h)               C
0040105A  add     ebx,eax                      D
0040105C  cmp     ebx,64h                      C
0040105F  pop     edi                          F
00401060  pop     ebp                          F
00401061  je      myfunction+7Bh (40107Bh)     C
```

```
00401063   cmp          ebx,6Eh                              C
00401066   je           myfunction+88h (401088h)             C
00401068   push         offset string "nothing" (407114h)    F
0040106D   call         printf (401107h)                     F
00401072   add          esp,4                                F
00401075   pop          esi                                  F
00401076   mov          eax,ebx                              F
00401078   pop          ebx                                  F
00401079   pop          ecx                                  F
0040107A   ret                                               F
0040107B   push         offset string "cnt is 100" (407108h) F
00401080   call         printf (401107h)                     F
00401085   add          esp,4                                F
00401088   push         offset string "cnt is 110" (4070FCh) F
0040108D   call         printf (401107h)                     F
00401092   add          esp,4                                F
00401095   pop          esi                                  F
00401096   mov          eax,ebx                              F
00401098   pop          ebx                                  F
00401099   pop          ecx                                  F
0040109A   ret                                               F
```

那么，标准的解法大致如下:

第一步是，我们要把指令分成几类，并实际的把它们区分开来，各个击破:

最先是函数调用相关代码。这些代码用于调用函数或者作为一个函数被调用。几乎凡是堆栈操作（备份寄存器或者是压入参数）我们可全部归入之，此外还有 call 指令，堆栈恢复。这类代码很容易识别。让我们把它们标记为 F.称之为 F 指令。

然后是流程控制代码。涉及判断和跳转指令，以及对循环变量操作的指令。这些对应用于循环，判断语句。比较容易看出。标记为 C.我们称之为 C 指令。

剩余的是数据处理。应该不含有函数调用，多半不含有堆栈操作，也不会含有跳转（跳转已经归入流程控制中）。对于复杂的数据处理代码，只能逐行翻译，然后多行组合。标记为 D.之后我们称之为 D 指令。

标记已经做好了，请回头看上面的代码.当然，你可能出于不同的观点，和我做上不同的标记，或者同样的指令归类不同，这都没有关系。

第二步，我们取出其中标记为 D 的代码进行逐行翻译。首先标记为 F 的语句基本不需要翻译，他们本身就是简单的函数调用。其次标记为 C 的指令我们将很快将他们翻译为 if,for,do 或者是 swtich,工作相对简单。

下面的指令取自上面的前半段的 D 指令。我删除了其中加杂的 F 指令。而 C 指令则是容易翻译的部分，那么我们尝试改写 D 指令为如下最右边的表达式，其中我用了 p1,p2,p3 表示函数的参数 1，参数 2，参数 3。取参数时针对 esp 的偏移，随着 push 指令的执行而有所变动，切不可机械计算。

```
00401001   mov          ecx,dword ptr [esp+10h]          D    ecx = p3
00401005   mov          edx,dword ptr [esp+8]            D    edx = p1
0040100A   mov          ebx,dword ptr [esp+4]            D    ebx = p1
0040100F   mov          esi,dword ptr [esp+14h]          D    esi = p2
00401014   xor          eax,eax                          C
00401020   mov          edi,dword ptr [esi+8]            D    edi = p2[2]
```

```
00401023   imul        edi,dword ptr [edx+eax*8+4]      D   edi*=p1[eax*2+1]
00401028   mov         ebp,dword ptr [esi]              D   ebp = p2[0]
0040102A   imul        ebp,dword ptr [edx+eax*8]        D   ebp *= p1[eax*2]
0040102E   add         edi,ebp                          D   edi += ebp
00401030   mov         dword ptr [ecx],edi              D   ecx [0] = edi
00401032   mov         edi,dword ptr [esi+0Ch]          D   edi = p2[3]
00401035   imul        edi,dword ptr [edx+eax*8+4]      D   edi = p1[eax*2+1]
0040103A   mov         ebp,dword ptr [esi+4]            D   ebp = p2[1]
0040103D   imul        ebp,dword ptr [edx+eax*8]        D   ebp *= p1[eax*2]
00401041   add         edi,ebp                          D   edi += ebp
00401043   mov         ebp,dword ptr [ecx]              D   ebp = ecx [0]
00401045   add         ebp,edi                          D   ebp += edi
00401047   add         ebx,ebp                          D   ebx += ebp
00401049   mov         dword ptr [ecx+4],edi            D   ecx [1] = edi
0040104C   inc         eax                              C
0040104D   add         ecx,8                            D   ecx+=8
00401050   cmp         eax,2                            C
00401053   jl          myfunction+20h (401020h)         C
```

第三步，自然是表达式的合并与控制流程的结合了，前面 D 系列的表达式，把中间过程除去，已经很容易得到表达式如下，其中 eax 开始为 0，ecx 开始为 p3.

```
ecx [0] = p2[2]*p1[eax*2+1]+p2[0]*p1[eax*2];
ecx [1] = p2[3]*p1[eax*2+1]+p2[1]*p1[eax*2];
```

此外 jl 明显导致了一个循环，每次循环的更改是：

```
eax++;
ecx+=8;
```

eax 是循环变量。ecx 显然用于取数组位置，且二者保持固定的同步增加，可以用一个循环变量替代之，所以翻译结果如下:

```
for(i=0;i<2;i++)
{
    p3[i] = p2[2]*p1[2*i+1]+p2[0]*p1[2*i];
    p3[i+1] = p2[3]*p1[2*i+1]+p2[1]*p1[i*2];
}
```

好了，以上就是三步的解法。后面的部分，也是依样画葫芦的完成，值得注意的是以下部分:

```
0040105C   cmp         ebx,64h                          C
0040105F   pop         edi                              F
00401060   pop         ebp                              F
00401061   je          myfunction+7Bh (40107Bh)          C
00401063   cmp         ebx,6Eh                          C
00401066   je          myfunction+88h (401088h)         C
```

除去其中的 F 指令，连续看到 cmp 和 je,说明这是一个 swtich.这和调试版本没有什么区别。翻译的难度也非常小。这里就不再重述详细的过程了。

第二节 演习：内核代码阅读

2-1.认识内核代码，新的函数调用方式

一般的认为，内核程序和驱动程序的概念近乎等价。它们被编译成.sys 文件放置在 Windows\System32\Drivers 目录下。这些代码将运行在 R0 级。拥有高权限。当然我们不能直接阅读机器码，必须得到汇编指令。IDA 是绝好的工具，它帮你将.sys 文件变成汇编代码。网络上有很多文章介绍使用的方法。

虽然能看到 IDA 得到的汇编代码，却不能亲眼看到系统运行，并进行调试，岂不是遗憾?其实调试更简单了，使用 Windbg 或者 Softice 均可。看到汇编语言单步运行，我们的解读能力正可大显身手。

把 sys 文件用 IDA 进行反汇编时，如果没有符号表，得到的函数和全局变量都没有可理解的名字。相信这对高手不是障碍？对我来说，真是很大的困扰。但是幸运的是，微软提供他的 sys 文件的每个版本的符号表文件（扩展名为.pdb）下载。在 windbg 中做简单的设置，便可以从网络上得到所有的 pdb 文件。有了这个，就能看到汇编中的函数名了。阅读一下明朗了很多。

我很想把前面对 c 语言反汇编的练习投入实用，然而，我又不敢直接阅读更复杂的内核代码。那样我可能会一下子晕掉失去信心。所以我直接从 DDK 中取例子 diskperf 编译了一个发行版反汇编了一下，算做初步的练习。这有几个好处，diskperf 仅仅 2000 多行代码，就算最差的情况逐行阅读，精力的损失也在可预测的范围之内。其次 diskperf 机制简单，我早已理解，也不担心过于不明的原理与机制造成困绕。再次 C 代码就在我手中，解读之后，还有机会比较正确答案。

拿到 diskperf 进行反汇编之后，我们当然首先看入口函数 DriverEntry.相关的反汇编结果如下。现在我们就来假定这是一个我们从没见过 C 代码的 Windows 内核组件，来尝试恢复它为 C 语言:

```
INIT:00011480 DriverEntry       proc near
INIT:00011480
INIT:00011480 arg_0             = dword ptr  8
INIT:00011480 arg_4             = dword ptr  0Ch
INIT:00011480

;以下 push 的操作都是 F 指令，目的是保存 esi,edi。然后 mov 是取参数。arg_0,
;arg_4 是 IDA 自己定义的常数。,用来取得参数。[esp+arg_4]就是[esp+0Ch],是第
;二个外部参数。

INIT:00011480                   push    esi
INIT:00011481                   mov     esi, [esp+arg_4]
INIT:00011485                   mov     ax, [esi]
INIT:00011488                   add     ax, 2
INIT:0001148C                   push    edi
INIT:0001148D                   mov     DiskPerfRegistryPath.MaximumLength, ax
INIT:00011493                   movzx   eax, ax

;下面是调用 ExAllocatePoolWithTag 分配内存的过程，IDA 把参数都标示了。

INIT:00011496                   push    66725044h       ; Tag
INIT:0001149B                   push    eax             ; NumberOfBytes
INIT:0001149C                   push    1               ; PoolType
```

```
INIT:0001149E                   call            ds:__imp__ExAllocatePoolWithTag@12  ;  __declspec(dllimport)
    ExAllocatePoolWithTag(x,x,x)
INIT:000114A4                   test    eax, eax
INIT:000114A6                   mov     DiskPerfRegistryPath.Buffer, eax
INIT:000114AB                   jz      short loc_114BB

;以上的 jz 显然涉及到一个 if.只有 eax 不为 0，才执行下面的，否则跳到 114BB 去了。
;下面进行了一个字符串拷贝。

INIT:000114AD                   push    esi             ; SourceString
INIT:000114AE                   push    offset DiskPerfRegistryPath ; DestinationString
INIT:000114B3                   call            ds:__imp__RtlCopyUnicodeString@8  ;  __declspec(dllimport)
    RtlCopyUnicodeString(x,x)
INIT:000114B9                   jmp     short loc_114CB

;以上这个跳转依然是上面的 if 的一部分。这避免了执行下面的给 DiskPerfRegistryPath.Length
;和 DiskPerfRegistryPath.MaximumLength 赋 0 的操作

INIT:000114BB
INIT:000114BB loc_114BB:                        ; CODE XREF: DriverEntry+2B↑ j
INIT:000114BB                   and     DiskPerfRegistryPath.Length, 0
INIT:000114C3                   and     DiskPerfRegistryPath.MaximumLength, 0
INIT:000114CB

; test-jz,然后 jmp，与前面说的 if 为 cmp-jl,然后后面的 else 是只有一个 jmp 一个意思。
;说明上面是一个单 if-else 结构,没有 else if 出现。

INIT:000114CB loc_114CB:                        ; CODE XREF: DriverEntry+39↑ j

;下面取了第 1 个参数。放到了 edx 中。edx+38h 取内容的话，这第一个参数要么是数组
;要么是结构体。当然，因为这个函数是 DriverEntry,我们很清楚第一个参数类型是
;DRIVER_OBJECT,也容易查到+38h 的位置是什么域。

INIT:000114CB                   mov     edx, [esp+4+arg_0]
INIT:000114CF                   lea     esi, [edx+38h]

;以下指令等于 mov ecx,1Ch,这看起来是一个循环的开始(ecx 常常用来做循环变量)。

INIT:000114D2                   push    1Ch
INIT:000114D4                   pop     ecx
INIT:000114D5                   mov     eax, offset DiskPerfSendToNextDriver
INIT:000114DA                   mov     edi, esi
INIT:000114DC                   rep stosd
```

;下面是漫长的赋值操作，最后是返回.

```
INIT:000114DE                   mov     eax, offset DiskPerfReadWrite
INIT:000114E3                   mov     [edx+44h], eax
INIT:000114E6                   mov     [edx+48h], eax
INIT:000114E9                   mov     eax, offset DiskPerfShutdownFlush
INIT:000114EE                   mov     [edx+78h], eax
INIT:000114F1                   mov     [edx+5Ch], eax
INIT:000114F4                   mov     eax, [edx+18h]
INIT:000114F7                   mov     dword ptr [esi], offset DiskPerfCreate
INIT:000114FD                   mov     dword ptr [edx+70h], offset DiskPerfDeviceControl
INIT:00011504                   mov     dword ptr [edx+94h], offset DiskPerfWmi
INIT:0001150E                   mov     dword ptr [edx+0A4h], offset DiskPerfDispatchPnp
INIT:00011518                   mov     dword ptr [edx+90h], offset DiskPerfDispatchPower
INIT:00011522                   mov     dword ptr [eax+4], offset DiskPerfAddDevice
INIT:00011529                   pop     edi
INIT:0001152A                   mov     dword ptr [edx+34h], offset DiskPerfUnload
INIT:00011531                   xor     eax, eax
INIT:00011533                   pop     esi
INIT:00011534                   retn    8       ;retn 8,返回的同时恢复堆栈
INIT:00011534 DriverEntry       endp
```

以上函数调用方式和我们前面所见的标准 C 函数调用方式只有一个区别:由被调用函数自己恢复堆栈。retn 8 等于把 esp+8 后，再调用 ret.这是 std call 的函数调用方式。是默认的内核函数的调用方法。

2-2.尝试反 C 内核代码

无论汇编水平多高，有一个前提，我们要看懂内核代码，首先，我们应该会使用 C 语言开发内核驱动程序。当然，不排除某些绝顶高手，光以汇编思维，通过盲人摸象搞定全局，跳过对 DDK 的学习，直接理解 Windows 内核代码。我是做不到的了，既然微软已经为 DDK 提供了详细的文档和例子，就应该好好利用。

现在尝试用前面学到的方法来反写 C 代码。diskperf 的代码手边就有。但是如果对照着看来写，这练习也就没有意义了。我只好假装我没看过。先看开头的一段:

```
INIT:00011480                   push    esi                             F
INIT:00011481                   mov     esi, [esp+arg_4]
INIT:00011485                   mov     ax, [esi]
INIT:00011488                   add     ax, 2
INIT:0001148C                   push    edi                             F
INIT:0001148D                   mov     DiskPerfRegistryPath..MaximumLength, ax
INIT:00011493                   movzx   eax, ax
```

首先忽略两条 F 指令。剩余的就简单了。mov esi, [esp+arg_4]意味着取第二个参数到 esi 中。DriverEntry 的第二个参数是 PUNICODE_STRING RegisterPath,UNICODE_STRING 的第一个域应该是 Length,那么[esi]就是取它的长度了。下面的指令是把

这个长度加 2，并保存到全局变量 DiskPerfRegistryPath.MaximumLength 中。

```
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject,PUNICODE_STRING RegisterPath)
{
    DiskPerfRegisteryPath.MaximumLength = RegisterPath.Length + 2;
    … …
}
```

然后是分配内存的代码:

```
INIT:00011496                   push    66725044h       ; Tag
INIT:0001149B                   push    eax             ; NumberOfBytes
INIT:0001149C                   push    1               ; PoolType
INIT:0001149E                   call    ds:__imp__ExAllocatePoolWithTag@12  ;  __declspec(dllimport)
    ExAllocatePoolWithTag(x,x,x)
INIT:000114A4                   test    eax, eax
INIT:000114A6                   mov     DiskPerfRegistryPath.Buffer, eax
```

以上这个函数调用过程，几乎简单得不用翻译。但是编译器把返回值要 mov 来 mov 去的，C 语言就不要那么笨拙了:

```
DiskPerfRegistryPathBuffer.Buffer = ExAllocatePoolWithTag(1,
    DiskPerfRegisteryPathMaximumLength,0x66725044);
```

然后下面的判断是对分配内存结果的判断.当 eax 不为 0 的时候，则拷贝字符串（jz short loc_114BB）。

```
INIT:000114AB                   jz      short loc_114BB
INIT:000114AD                   push    esi             ; SourceString
INIT:000114AE                   push    offset DiskPerfRegistryPath ; DestinationString
INIT:000114B3                   call    ds:__imp__RtlCopyUnicodeString@8  ;  __declspec(dllimport)
    RtlCopyUnicodeString(x,x)
INIT:000114B9                   jmp     short loc_114CB
INIT:000114BB
INIT:000114BB loc_114BB:                                ; CODE XREF: DriverEntry+2B↑ j
INIT:000114BB                   and     DiskPerfRegistryPath.Length, 0
INIT:000114C3                   and     DiskPerfRegistryPath.MaximumLength, 0
INIT:000114CB
INIT:000114CB loc_114CB:                                ; CODE XREF: DriverEntry+39↑ j
```

那么以上的 if-else 模块大致如下：

```
if(DiskPerfRegistryPathBuffer.Buffer == NULL)
{
    DiskPerfRegisterPath.Length = 0;
    DiskPerfRegisterPath.MaximumLength = 0;
}
else
{
    RtlCopyUnicodeString(&DiskPerfRegistryPath,RegisterPath);
}
```

再往下是这么一段:

```
INIT:000114CB                   mov     edx, [esp+4+arg_0]          ;edx = DriverObject
INIT:000114CF                   lea     esi, [edx+38h]              ;esi = DriverObject->DispatchFunctions
INIT:000114D2                   push    1Ch                         ;ecx = 1Ch
INIT:000114D4                   pop     ecx
INIT:000114D5                   mov     eax, offset DiskPerfSendToNextDriver      ;这是拷贝的源
INIT:000114DA                   mov     edi, esi                    ;拷贝的目的
INIT:000114DC                   rep stosd                           ;重复拷贝 1Ch 填满从 esi 开始的区域
```

edx+38h 的理解需要对 DriverObject 结构的了解。但这不是问题，有头文件可以查。一些困难的结构可以通过 Windbg 的指令来了解。还有更多没有公开的结构只能去找找别人的资料碰碰运气。不过完全无法找到资料的话，你可以自己假定一个。我们不论是否理解，都要把它翻译成 C 语言。

```
for(i=0;i<0x1c;i++)
{
    DriverObject->DispatchFunctions[i] = DiskPerfSendToNextDriver;
}
```

… 后面是类似的填写分发函数，就不重复这个过程了。这是我的第一步尝试，看起来以前 C 语言反汇编结果的阅读的基本的方法都是可以用上的。

2-3.寻找需要的信息

如果我们给在 DriverEntry 中出现过的各个函数， 依次来个反 C,同时把这个操作递归下去，我们还真的可以把整个代码变成 C 源代码。不过做做练习还好，如果出于实际的需求这样做的话，就显得笨拙了。作为内核代码的阅读者，应该迅速的扑向自己所需要的信息才对。

Diskperf 这个工程大家都是如此的熟悉，以至于我都想不出什么让人感兴趣的技术点需要通过反汇编去了解，但是出于练习的需要，我就假定：我们都了解 diskperf 通过绑定物理磁盘设备来进行磁盘操作的过滤。但是我们不知道，它是如何发现这些物理磁盘设备（或者枚举？），又是如何绑定的呢？是如何和系统磁盘的增加减少保持同步的呢？现在，我们只有 diskperf 的 sys 文件和符号表，那么我们自己来了解这些信息吧。

很容易想到绑定一个设备会调用 IoAttachDeviceToDeviceStack。打开 IDA 反汇编 diskperf.sys 后，寻找名字 IoAttachDeviceToDeviceStack,只有一处调用。在函数 DiskPerfAddDevice 中。

```
…
PAGE:0001127E loc_1127E:                                ; CODE XREF: DiskPerfAddDevice+87↑j
PAGE:0001127E                   mov     eax, [ebp+TargetDevice]
PAGE:00011281                   push    eax             ; TargetDevice
PAGE:00011282                   mov     [esi+8], eax
PAGE:00011285                   push    [ebp+IoObject]  ; SourceDevice
PAGE:00011288                   call    ds:__imp__IoAttachDeviceToDeviceStack@8 ; __declspec(dllimport)
IoAttachDeviceToDeviceStack(x,x)
```

IoAttachDeviceToDevice 的第二个参数是 TargetDevice,那么被 push 进入堆栈的第一个参数就是 TargetDevice.而这个参数从堆栈中取得，也就是[ebp+TargeDevice],这里的 TargeDevice 是 IDA 定义的常数。在有符号表的情况下，这些参数都有了有意义的名字，而实际的数值可以在 DiskPerfAddDevice 的开头看到:

```
PAGE:000111E2 DiskPerfAddDevice proc near               ; DATA XREF: DriverEntry+A2├o
```

```
PAGE:000111E2
PAGE:000111E2 IoObject          = dword ptr  8
PAGE:000111E2 TargetDevice      = dword ptr  0Ch
```

可见 TargetDevice 的实际数值是 0Ch,这是 DiskPerfAddDevice 传入的第二个参数。而第一个参数名字是 IoObject.

有兴趣的话，现在来反 C 这个函数的其他部分，以便得到更清晰的解答，下面是函数最开始的部分：

```
PAGE:000111E2                   push    ebp
PAGE:000111E3                   mov     ebp, esp
PAGE:000111E5                   push    ebx
PAGE:000111E6                   lea     eax, [ebp+IoObject]
PAGE:000111E9                   push    eax             ; DeviceObject
PAGE:000111EA                   xor     ebx, ebx
PAGE:000111EC                   push    ebx             ; Exclusive
PAGE:000111ED                   mov     eax, 100h
PAGE:000111F2                   push    eax             ; DeviceCharacteristics
PAGE:000111F3                   push    7               ; DeviceType
PAGE:000111F5                   push    ebx             ; DeviceName
PAGE:000111F6                   push    eax             ; DeviceExtensionSize
PAGE:000111F7                   push    [ebp+IoObject]  ; DriverObject
PAGE:000111FA                       call    ds:__imp__IoCreateDevice@28 ; __declspec(dllimport)
IoCreateDevice(x,x,x,x,x,x,x)
```

这一段把大量的参数 push 到堆栈里，然后调用函数 IoCreateDevice,这是本书一开始就讲述的内容，我们应该已经非常的熟悉了,注意参数是倒序压入堆栈的.目前我知道函数有返回值，但是不知道其类型。简单起见，返回 ULONG.同时对于第一个参数 IoObject 我本不了解是什么类型，无奈 IDA 提示我了这是一个 DRIVER_OBJECT。

```
ULONG DiskPerfAddDevice(PDRIVER_OBJECT IoObject, PDEVICE_OBJECT TargetDevice)
{
    NTSTATUS status;
    // 把我们 push 过的参数填入。但是，你要除去其中的函数开始时，对 ebp 和 ebx 进行备份的指令。
    status = IoCreateDevice(IoObject,0x100,0,7,0x100,0,( PDRIVER_OBJECT *)&IoObject);
}
```

上面的代码是不是有点荒谬?IoCreateDevice 最后一个参数明明应该是一个 DEVICE_OBJECT,为何把 IoObject 给传了进去?

正确的做法应该是这样的:

```
ULONG DiskPerfAddDevice(PDRIVER_OBJECT IoObject, PDEVICE_OBJECT TargetDevice)
{
    NTSTATUS status;
    PDEVICE_OBJECT Device;
    status = IoCreateDevice(IoObject,0x100,0,7,0x100,0,&Device);
}
```

是的，但是其实前面的写法也没有错!不要忘记了汇编语言是没有类型概念的.IoObject 在堆栈中传入.函数调用完后被恢复,所以这些空间，是这个函数可以随意使用,不会对外界造成影响.那么，我再在函数内部定义一个

```
PDEVICE_OBJECT Device;
```

真的还不如直接利用传入的参数 PDRIVER_OBJECT IoObject。他们虽然类型不同，大小却是一样的，都是四个字节的指针而已.这是编译器优化的结果.

我们看到的答案就是，DiskPerfAddDevice 负责生成设备并绑定原始的设备。而原始设备的来源是 DiskPerfAddDevice 的参数。而这个函数被设置在 DriverEntry 中:

```
INIT:000114F4                   mov     eax, [edx+18h]
INIT:000114F7                   mov     dword ptr [esi], offset DiskPerfCreate
INIT:000114FD                   mov     dword ptr [edx+70h], offset DiskPerfDeviceControl
INIT:00011504                   mov     dword ptr [edx+94h], offset DiskPerfWmi
INIT:0001150E                   mov     dword ptr [edx+0A4h], offset DiskPerfDispatchPnp
INIT:00011518                   mov     dword ptr [edx+90h], offset DiskPerfDispatchPower
INIT:00011522                   mov     dword ptr [eax+4], offset DiskPerfAddDevice
```

这里的 edx 的位置是 DRIVER_OBJECT,DRIVER_OBJECt 偏移 0x18 之后，取得其地址，那是 DriverExtension，然后 DriverExtension 取第 5 个字节开始的位置，那是 DriverExtension->AddDevice。相关的知识很简单，就不详细讲述了。

2-4. 了解内核调用的位置

下面要用我们的知识来做一些有用的事情。Windows 从 2000 发展到 XP 后，XP DDK 中出现了一些新的调用。内核程序开发者有时会发现，这些调用非常有用（这也是这些新调用产生的原因），但是如果使用他们，会导致驱动在 2000 下无法使用。目前 2000 的用户依然很多，存在这样的可移植问题是非常遗憾的。退一步的方案是 2000 下限制某些功能。在程序中动态加载系统调用，并小心的判断当前的版本。在 2000 的情况下，一些功能被跳过。这样比前者好，但是依然不是最理想的解决方案。

有一些人开始使用非文档的解决方案。非文档解决方案的危害就是，可能不兼容未来的操作系统。但是由于我们现在这样做仅仅是针对一个过去版本的操作系统，问题就不复存在了。在新版本的操作系统上，我们调用新的系统调用，而在某个已经存在而且不会再变的操作系统版本上，我们调用非文档的方法。只要低版本操作系统测试无问题，以后的也不会有问题。

所谓的非文档方案就是，在新版本的操作系统上，把新出现的功能调用进行反汇编，更好的是反写为 C 语言。然后直接编译或者少许修改后放到我们的内核程序里，同时检测当前操作系统为低版本时，调用这些非文档的代码。

下面我们举两个例子，

```
PDEVICE_OBJECT
IoGetDeviceAttachmentBaseRef(
IN PDEVICE_OBJECT   DeviceObject);
```

这个调用获得设备栈底的设备。这对过滤驱动非常有用。

为了自己实现它，我们首先必须要看到这些调用的反汇编代码。这并不难，你可以下载符号表并用 IDA 反汇编内核模块。不过更好的办法是用 Windbg 调试运行 Windows. Windbg 可以自动下载对应的符号表。操作简单。整个 Windows 的反汇编会迅速展现在你面前并随时可以设置断点调试。

调试 Windows 需要两台机器。或者在一台计算机上使用虚拟机。用实际的串口线或者管道模拟的串口连接。并经过一系列的设置。在网络上有很多文档。最好设置从 MS 直接 load 对应符号表。这样可以保证符号表版本不出问题。

我很希望看到 WindowsXP 下这两个函数的反汇编。但是恰好我的手头只有一台被调试的 Vista.现在打开 Windbg,在命令输入框出输入:

```
u IoGetDeviceAttachmentBaseRef
```

显示如下:

```
kd> u IoGetDeviceAttachmentBaseRef
nt!IoGetDeviceAttachmentBaseRef:
818c151c 8bff            mov     edi,edi
818c151e 55              push    ebp
818c151f 8bec            mov     ebp,esp
818c1521 53              push    ebx
818c1522 56              push    esi
818c1523 6a0a            push    0Ah
```

```
818c1525 59                pop       ecx
818c1526 ff1500118081      call      dword ptr [nt!_imp_KeAcquireQueuedSpinLock (81801100)]
```

现在看到的是 IoGetDeviceAttachmentBaseRef 的开头部分。我们已经看到了地址在 818c151c，那么要看完整的部分也简单了，主菜单 View->Disassembly,出现 Disassembly 窗口，上部的 offset 输入框中输入 818c151c,IoGetDeviceAttachmentBaseRef 的代码如下:

```
nt!IoGetDeviceAttachmentBaseRef:
818c151c 8bff              mov       edi,edi          无意义指令
818c151e 55                push      ebp              ;保存 ebp
818c151f 8bec              mov       ebp,esp          ;保存 esp
818c1521 53                push      ebx              ;保存 ebx
818c1522 56                push      esi              ;保存 esi
818c1523 6a0a              push      0Ah              ;把 0ah 当作第一个参数传给 KeAcquireQueuedSpinLock
818c1525 59                pop       ecx
818c1526 ff1500118081      call      dword ptr [nt!_imp_KeAcquireQueuedSpinLock (81801100)]
818c152c 8ad8              mov       bl,al            ;bl = KeAcquireQueuedSpinLock 的返回值.
818c152e 8b4508            mov       eax,dword ptr [ebp+8]      取得唯一的一个参数 DEVICE_OBJECT
818c1531 e8d3ffffff        call      nt!IopGetDeviceAttachmentBase+0x4 (818c1509)
818c1536 8bf0              mov       esi,eax
818c1538 8bce              mov       ecx,esi
818c153a e8dd5ff8ff        call      nt!ObfReferenceObject (8184751c)
818c153f 6a0a              push      0Ah
818c1541 8ad3              mov       dl,bl
818c1543 59                pop       ecx
818c1544 ff15fc108081      call      dword ptr [nt!_imp_KeReleaseQueuedSpinLock (818010fc)]
818c154a 8bc6              mov       eax,esi
818c154c 5e                pop       esi
818c154d 5b                pop       ebx
818c154e 5d                pop       ebp
818c154f c20400            ret       4
```

同时，IopGetDeviceAttachmentBase 的代码如下:

```
nt!IopGetDeviceAttachmentBase:
818c1500 eb07              jmp       nt!IopGetDeviceAttachmentBase+0x4 (818c1509)
818c1502 8bc1              mov       eax,ecx
818c1504 90                nop
818c1505 90                nop
818c1506 90                nop
818c1507 90                nop
818c1508 90                nop
818c1509 8b88b0000000      mov       ecx,dword ptr [eax+0B0h]
818c150f 8b4918            mov       ecx,dword ptr [ecx+18h]
818c1512 85c9              test      ecx,ecx
818c1514 75ec              jne       nt!IopGetDeviceAttachmentBase+0x2 (818c1502)
818c1516 c3                ret
```

要直接阅读上面的代码，现在可能还有些问题。这在下一小节中描述。

2-5. 自己实现 XP 的新调用，新的函数调用方式

我们阅读 vista 下的 IopGetDeviceAttachmentBase 可能有点小麻烦。因为 Iop 系列的函数是内部函数。而微软的内部函数系统喜欢用 fast call 的方式。这种方式与前面的 std call 方式几乎完全相同。唯一区别是，前两个参数不被放到堆栈中传入，而是放入 ecx 和 edx 中。ecx 中将保存第一个参数，edx 中保存第二个。

这就是为何 IopGetDeviceAttachmentBase 这个函数返回的时候不需要恢复堆栈。因为参数没有通过堆栈传递。此外另人疑惑的是开头的 jmp.这样一来，通过 ecx 传参数就被跳过了。我只能理解为，这个函数应该是从 818c1502 开始的。

IopGetDeviceAttachmentBase 的被调方式是:

```
818c152e 8b4508          mov     eax,dword ptr [ebp+8]      取得唯一的一个参数 DEVICE_OBJECT
818c1531 e8d3ffffff      call    nt!IopGetDeviceAttachmentBase+0x4 (818c1509)
```

这并非直接调用 nt!IopGetDeviceAttachmentBase 这个函数，而是直接把参数 mov 到 eax 中，然后 jmp 到了 818c1509 处。这些代码在 vista 下经常出现。不象是一般的编译结果。像是经过某种特殊的优化后的结果。

818c1509 处开始的代码就很简单了:

```
818c1509 8b88b0000000   mov     ecx,dword ptr [eax+0B0h]
818c150f 8b4918         mov     ecx,dword ptr [ecx+18h]
818c1512 85c9           test    ecx,ecx
818c1514 75ec           jne     nt!IopGetDeviceAttachmentBase+0x2 (818c1502)
818c1516 c3             ret
```

eax 是传入参数，也就是 PDEVICE_OBJECT 的指针。mov ecx,dword ptr [eax+0B0h]中，DEVICE_OBJECT 的 B0 是一个让人颇有些眼熟的位置。有人给我一个文件，里面有 XP 下 DEVICE_OBJECT 的结构，内容如下：

```
struct _XP2600_2180_DEVICE_OBJECT /* sizeof 000000B8 184 */
{
/* off 0x00000000 */  short      Type;
/* off 0x00000002 */  unsigned short     Size;
/* off 0x00000004 */  long ReferenceCount;
/* off 0x00000008 */  struct _XP2600_2180_DRIVER_OBJECT* DriverObject;
/* off 0x0000000C */  struct _XP2600_2180_DEVICE_OBJECT* NextDevice;
/* off 0x00000010 */  struct _XP2600_2180_DEVICE_OBJECT* AttachedDevice;
/* off 0x00000014 */  struct _XP2600_2180_IRP* CurrentIrp;
/* off 0x00000018 */  struct _XP2600_2180_IO_TIMER*    Timer;
/* off 0x0000001C */  unsigned long      Flags;
/* off 0x00000020 */  unsigned long      Characteristics;
/* off 0x00000024 */  struct _XP2600_2180_VPB*      Vpb;
/* off 0x00000028 */  void*     DeviceExtension;
/* off 0x0000002C */  unsigned long      DeviceType;
/* off 0x00000030 */  char StackSize;
/* off 0x00000034 */  union _XP2600_2180__unnamed_000001D5  Queue;
/* off 0x0000005C */  unsigned long      AlignmentRequirement;
/* off 0x00000060 */  struct _XP2600_2180_KDEVICE_QUEUE  DeviceQueue;
/* off 0x00000074 */  struct _XP2600_2180_KDPC     Dpc;
/* off 0x00000094 */  unsigned long      ActiveThreadCount;
/* off 0x00000098 */  void*     SecurityDescriptor;
/* off 0x0000009C */  struct _XP2600_2180_KEVENT  DeviceLock;
/* off 0x000000AC */  unsigned short     SectorSize;
```

```
/* off 0x000000AE */  unsigned short     Spare1;
/* off 0x000000B0 */  struct _XP2600_2180_DEVOBJ_EXTENSION*  DeviceObjectExtension;
/* off 0x000000B4 */  void*     Reserved;
};
```

这个头文件来自吴岩峰。所以请不要问我从如何获得了。如果你有耐心用 Windbg 逐个分析那些结构，或许也可以自己取得。如果你不知道这个结构，请直接用指针移动 0xB0 字节来获取就可以了，没有必要关心这个结构。

然后下一步是 mov ecx,dword ptr [ecx+18h]，可见[eax+0B0h]本身也是结构体指针。我忍不住又查了一下那个结构：

```
typedef struct _XP2600_2180_DEVOBJ_EXTENSION /* sizeof 0000002C 44 */
{
/* off 0x00000000 */  short     Type;
/* off 0x00000002 */  unsigned short     Size;
/* off 0x00000004 */  struct _XP2600_2180_DEVICE_OBJECT*  DeviceObject;
/* off 0x00000008 */  unsigned long     PowerFlags;
/* off 0x0000000C */  struct _XP2600_2180_DEVICE_OBJECT_POWER_EXTENSION*    Dope;
/* off 0x00000010 */  unsigned long     ExtensionFlags;
/* off 0x00000014 */  void*     DeviceNode;
/* off 0x00000018 */  struct _XP2600_2180_DEVICE_OBJECT*  AttachedTo;
/* off 0x0000001C */  long StartIoCount;
/* off 0x00000020 */  long StartIoKey;
/* off 0x00000024 */  unsigned long     StartIoFlags;
/* off 0x00000028 */  struct _XP2600_2180_VPB*     Vpb;
}XP2600_2180_DEVOBJ_EXTENSION,*PXP2600_2180_DEVOBJ_EXTENSION;
```

0x18 处是 AttachedTo,非常的理想，符合我们的猜测。从 XP 到 Vista 下都没有改变过这个结构。那么 2000 下是否是一样呢？这个可以自己验证一下：运行代码，如果没有崩溃，那么一切 OK.

首先自己实现一个,没有头文件的时候，你得自己定义结构中 0x80 和 0x18 这样的位置，当然你定义什么那是无所谓的，只要你的目的是明确的。在有以上头文件的时候，可以这么实现:

```
PDEVICE_OBJECT IopGetDeviceAttachmentBase(PDEVICE_OBJECT device)
{
    PXP2600_2180_DEVICE_OBJECT mydevice = (PXP2600_2180_DEVICE_OBJECT)device;
    PXP2600_2180_DEVOBJ_EXTENSION my_extension =
    (PXP2600_2180_DEVOBJ_EXTENSION)mydevice-> DeviceExtension;
    if(my_extension->AttachedTo == NULL)
        return device;
    else
        return IopGetDeviceAttachmentBase((PDEVICE_OBJECT)my_extension->AttachedTo);
}
```

然后是 IoGetDeviceAttachmentBaseRef，从前文的汇编看，流程是这样的:

第一步，call dword ptr [nt!_imp_KeAcquireQueuedSpinLock (81801100)]，参数为 0xA.

第二步，调用 IopGetDeviceAttachmentBase。

第三步，call nt!ObfReferenceObject (8184751c)。目的是对这个设备记一次引用。

第四步，call dword ptr [nt!_imp_KeReleaseQueuedSpinLock (818010fc)]，这是对前面的 KeAcquireQueuedSpinLock 调用一个释放。

以上的调用 ObfReferenceObject 在 2000 下存在。IopGetDeviceAttachmentBase 我们自己实现了。

KeAcquireQueuedSpinLock 和 KeReleaseQueuedSpinLock 是为了同步设备链。考虑多线程的情况，我遍历设备链表的时候，显然不希望其他线程操作这个设备链表。KeAcquireQueuedSpinLock 这个调用在 2000 下也没有。但是仅仅是阻止线程切换的话，我们可以通过提高中断级来实现。这按理和 KeAcquireQueuedSpinLock 效果是一样的。

```
PDEVICE_OBJECT
IoGetDeviceAttachmentBaseRef(
IN PDEVICE_OBJECT   DeviceObject)
{
    KIRQL irql;
    PDEVICE_OBJECT baseDevice;
    irql =   KeRaiseIrqlToDpcLevel();
    baseDevice = IopGetDeviceAttachmentBase(DeviceObject);
    ObReferenceObject(baseDevice);
    KeLowerIrql(irql);
    return baseDevice;
}
```

2-6. 没有符号表的反汇编

这一小节将在本书的正式版中补全。

第三节 实战:反汇编引擎, HOOK 系统调用

3-1 反汇编引擎 XDE32 之熟悉指令

当我们需要在程序中动态分析指令的时候，反汇编引擎是必要的东西。一个反汇编引擎的作用是：把机器码解析成可以理解的指令。这个理解不仅仅针对人，也可以针对程序。这样我们就不必开发严格依赖于具体实现的代码，而可以根据读取的机器码的分析结果，进行随机应变了。而且幸运的是，反汇编引擎通常很小。网上能找到很多开源的反汇编引擎。加入到你自己的程序中并不困难。

理解反汇编引擎需要对 i386 的机器指令编码有深入的了解。下载 intel 公司发布的说明很简单，但是理解起来确实非常的困难。我略过其中很多细节，打算努力的理解一些我需要使用的信息。

首先是一条指令的组成，大致如下图所示：

前缀(可选)	操作码	ModR/M(可选)	SIB(可选)	地址偏移(可选)	立即操作数(可选)

前缀：指令前缀，包括加锁，重复等信息。每个前缀 1 个字节。但是最多可能有 4 个前缀。

操作码：指令的机器编码。可能占有 1-3 个字节。一般只有一个操作码。

ModR/M：修改的寄存器与存储器。如果有，则必为 1 个字节。

SIB：和 ModR/M 相关。一些特殊的寻址指令需要的信息(Scale,Index,Base)。如果有，必为 1 字节。

地址偏移：取决于寻址方式的需要。可能为 1，2，4 字节。

立即操作数：取决于寻址方式的需要。可能为 1，2，4 字节。

一条指令既然基本如此，那么反汇编引擎的目标就是把这些解析出来。我使用了 wowocock 所使用的 XDE32 反汇编引擎。这些代码都是开放源代码。主要有三个文件组成，xde32_Table.h,xde32.h,xde32.c.有兴趣的读者请自己下载研究。

xde32 用一个结构来代表一条指令的所有信息，这个结构如下：

```
#pragma pack(push)
#pragma pack(1)
struct xde_instr
{
    unsigned char   defaddr;    // 2 或者 4 字节数。
    unsigned char   defdata;    // 2 或者 4 字节数。
    unsigned long   len;        // 这个结构对应的指令的总长度。
    unsigned long   flag;       // 指令特征标记
    unsigned long   addrsize;   // 地址偏移量的字节数(1,2,4)
    unsigned long   datasize;   // 立即数的字节数(1,2,4)
    unsigned char   p_lock;     // 是否有加锁前缀，0 或 F0
    unsigned char   p_66;       // 是否有数据覆盖前缀，0 或 66
    unsigned char   p_67;       // 是否有地址覆盖前缀，0 或 67
    unsigned char   p_rep;      // 是否有重复前缀 0，F2 或 F3
    unsigned char   p_seg;      // 是否有段地址寄存器覆盖前缀 0 或 26/2E/36/3E/64/65
    unsigned char   opcode;     // 操作码。当操作码为 0F 的时候，有指令第二字节
    unsigned char   opcode2;    // 指令第二字节
    unsigned char   modrm;      // 指令中的 ModR/M 字节
    unsigned char   sib;        // 指令中的 SIB 字节
    unsigned long   src_set;    // 指令操作源对象（包括寄存器，存储器等）
    unsigned long   dst_set;    // 指令操作目标对象（包括寄存器，存储器等）
    union                       // 地址
    {
        unsigned char   addr_b[8];
        unsigned short addr_w[4];
        unsigned long   addr_d[2];
        signed char     addr_c[8];
        signed short    addr_s[4];
        signed long     addr_l[2];
    };
    union                       // 立即数
    {
        unsigned char   data_b[8];
        unsigned short data_w[4];
        unsigned long   data_d[2];
        signed char     data_c[8];
        signed short    data_s[4];
        signed long     data_l[2];
    };
}; /* struct xde_instr */
```

```
#pragma pack(pop)
```

以上信息就覆盖了一条指令的所有信息。作为一个反汇编引擎，必须解析一个机器码并把所有的信息填入其中。一个指令对应的指令特征（在上面用灰底突出打印）起非常大的作用。在 xde32 中，特征是自己定义的，用来描述一个指令的结构、各段的字节长度等等信息。每个操作码都拥有一些指令特征，这决定了下面如何进行继续解析。指令对应指令特征形成了一个表。所有的指令特征定义如下:

```
#define C_ADDR1      0x00000001     // }
#define C_ADDR2      0x00000002     // } 指令的地址字节的有效位
#define C_ADDR4      0x00000004     // }
#define C_MODRM      0x00000008     // 指令有一个 ModR/M 字节
#define C_SIB        0x00000010     // 指令有一个 SIB 字节
#define C_ADDR67     0x00000020     // 地址长度覆盖前缀存在，此时地址字节数为 defaddr
#define C_DATA66     0x00000040     // 立即数长度覆盖前缀存在，此时立即数字节数为 defaddr
#define C_UNDEF      0x00000080     // 寄存器没有定义(这条指令操作未知的寄存器)
#define C_DATA1      0x00000100     // }
#define C_DATA2      0x00000200     // }指令的立即数字节的有效位
#define C_DATA4      0x00000400     // }
#define C_BAD        0x00000800     // 坏指令，很少用到的一个标记
#define C_REL        0x00001000     // 这是跳转指令 jxx 或者 call
#define C_STOP       0x00002000     // 这是回跳指令，ret 或者 jmp
#define C_OPSZ8      0x00004000     // 操作数的大小是 8 位。如果没有这个标记则是 16 位或者 32 位。
#define C_SRC_FL     0x00008000     // 这条指令需要读取标志寄存器
#define C_DST_FL     0x00010000     // 这条指令影响标志积存器
#define C_MOD_FL     (C_SRC_FL+C_DST_FL) // 上面两条的组合
#define C_SRC_REG    0x00020000     // 这条指令需要读取一般寄存器???
#define C_DST_REG    0x00080000     //这条指令需要写一般寄存器???
#define C_MOD_REG    (C_SRC_REG+C_DST_REG) // 上面两条的组合
#define C_SRC_RM     0x00040000     /* src_set |= f(R/M)   //???
#define C_DST_RM     0x00100000     /* dst_set |= f(R/M)   can be used w/o modrm*/ ???
#define C_MOD_RM     (C_SRC_RM+C_DST_RM) ???
#define C_SRC_ACC    0x00200000     // 需要读取 AL,AX,EAX
#define C_DST_ACC    0x00400000     // 需要操作 AL,AX,EAX
#define C_MOD_ACC    (C_SRC_ACC+C_DST_ACC)
#define C_SRC_R0     0x00800000     /* src_set |= f(opcode & 0x07) */ ???
#define C_DST_R0     0x01000000     /* dst_set |= f(opcode & 0x07) */ ???
#define C_MOD_R0     (C_SRC_R0+C_DST_R0) ???
#define C_PUSH       0x02000000     /* dst_set |= XSET_ESP | XSET_MEM */ ???
#define C_POP        0x04000000     /* dst_set |= XSET_ESP, src_set |= XSET_MEM*/ ???
```

以上是指令特征。一个操作码可能拥有好几个指令特征。所以形成一张表。这里，按操作码从 0x00 开始排列，每一行代表一个操作码对应的指令特征的集合，这是一个表，在头文件 xde32_table.h 中:

```
unsigned long xde_table[ TBL_max ] =
{
        // add modrm
/* 00 */   C_MODRM+C_DST_FL+C_SRC_REG+C_MOD_RM+C_OPSZ8,
/* 01 */   C_MODRM+C_DST_FL+C_SRC_REG+C_MOD_RM,
```

```
/* 02 */   C_MODRM+C_DST_FL+C_MOD_REG+C_SRC_RM+C_OPSZ8,
/* 03 */   C_MODRM+C_DST_FL+C_MOD_REG+C_SRC_RM,
           // add al, c8
/* 04 */   C_DATA1+C_DST_FL+C_MOD_ACC+C_OPSZ8,
           // add ax/eax, c16/32
/* 05 */   C_DATA66+C_DST_FL+C_MOD_ACC,
           // push es
/* 06 */   C_BAD+C_PUSH+C_SPECIAL,
           // pop es
/* 07 */   C_BAD+C_POP+C_SPECIAL,
           // or modrm
/* 08 */   C_MODRM+C_DST_FL+C_SRC_REG+C_MOD_RM+C_OPSZ8,
/* 09 */   C_MODRM+C_DST_FL+C_SRC_REG+C_MOD_RM,
/* 0A */   C_MODRM+C_DST_FL+C_MOD_REG+C_SRC_RM+C_OPSZ8,
/* 0B */   C_MODRM+C_DST_FL+C_MOD_REG+C_SRC_RM,
           // or al, c8
/* 0C */   C_DATA1+C_DST_FL+C_MOD_ACC+C_OPSZ8,
           // or ax/eax, c16/32
/* 0D */   C_DATA66+C_DST_FL+C_MOD_ACC
… …
}
```

这个表很长。有多少条指令就有多少行。为了节约篇幅，我只引用了前面的小部分。从这个表里可以迅速找到一个指令的特征用于之后的分析。这个表是反汇编引擎的核心所在。你可以在网上下载完整的表。

3-2 反汇编引擎 XDE32 之具体实现

这一小节来看看 xde32.c 中的实现部分。首先是汇编函数:

```
int __cdecl xde_asm(/* OUT */ unsigned char* opcode,
                    /* IN */ struct xde_instr* diza);
```

这个函数输入为前面的 xde_instr 结构。也就是被解析过的指令。输出为指令机器码。你会发现写机器码真的是很简单的一件事情:

```
int __cdecl xde_asm(/* OUT */ unsigned char* opcode,
                    /* IN */ struct xde_instr* diza)
{
  unsigned char* p;
  unsigned int i;
  p = opcode;
  // 首先写各个前缀
  if (diza->p_seg )          *p++ = diza->p_seg;
  if (diza->p_lock)          *p++ = diza->p_lock;
  if (diza->p_rep )          *p++ = diza->p_rep;
  if (diza->p_67   )          *p++ = diza->p_67;
  if (diza->p_66   )          *p++ = diza->p_66;
  // 然后写操作码
```

```
  *p++ = diza->opcode;
  if (diza->opcode == 0x0F)        *p++ = diza->opcode2;
  // 写 ModR/M 字节
  if (diza->flag & C_MODRM)        *p++ = diza->modrm;
  if (diza->flag & C_SIB)          *p++ = diza->sib;
  // 写地址字节
  for(i=0; i<diza->addrsize; i++) *p++ = diza->addr_b[i];
  // 写立即数字节
  for(i=0; i<diza->datasize; i++) *p++ = diza->data_b[i];
  // 返回写了多少个字节
  return p - opcode;
}
```

当然这离一个汇编编译程序还差得远。一个汇编编译程序应该能解析字符串，根据汇编助记符来生成机器码。绝对不是通过解析过的机器码信息。可见反汇编引擎和汇编编译程序的差别还是很大的。下面看反汇编部分的实现，这个函数原型是这样的：

```
int __cdecl xde_disasm(/* IN */ unsigned char *opcode,
                       /* OUT */ struct xde_instr *diza)
```

我们拿到的是机器码 opcode(只知道所在指针，不知道到底有多长)。i386 的指令是变长指令。你不得不去把一堆连续的字节拆解成一条一条的指令，而且还没有非常直观的方法。这个函数返回的是，当前解析出的这条指令有多长。这样，你就可以把指针移动一个长度来解析下一条指令。解析出的信息放在 diza 中。下面是具体的实现过程：

```
int __cdecl xde_disasm(/* IN */ unsigned char *opcode,
                       /* OUT */ struct xde_instr *diza)
{
  unsigned char c, *p;
  unsigned long flag, a, d, i, xset;
  unsigned long mod, reg, rm, index, base;
  // 操作码
  p = opcode;
  // 把 diza 清 0
  memset(diza, 0x00, sizeof(struct xde_instr));
  // 得到数据字节数和地址字节数
  diza->defdata = XDE32_DEFAULT_ADDR/8;
  diza->defaddr = XDE32_DEFAULT_DATA/8;

  // 指令的特征标记，先清 0
  flag = 0;

  // 如果前两个字节为全 0 或者全 F,认为是坏指令??
  if (*(unsigned short*)p == 0x0000) flag |= C_BAD;
  if (*(unsigned short*)p == 0xFFFF) flag |= C_BAD;
  … …
}
```

前面只是一些准备活动。然后下面的解析按前面的指令图解，分为这么几步：

第一步，解析前缀。

第二步，解析操作码。并取得对应的指令特征。

第四步，根据指令特征，如果存在 ModR/M 字节，则解析之。

第三步，根据前面的解析，如果存在 SIB 字节，则解析之。

第五步，如果根据前面的解析，有地址字节存在，则解析之。

第六步，如果有立即数存在，则解析之。

中间每一步的时候，还会随时统计涉及到的的（被读或者被写）寄存器或者存储器等。这统计烦琐，几乎占用了这中间绝大多数篇幅。而我对此又不是很感兴趣的，因此就没怎么去看了。

```
int __cdecl xde_disasm(/* IN */ unsigned char *opcode,
                       /* OUT */ struct xde_instr *diza)
{
  ……
  // 解析前缀
  while(1)
  {
    c = *p++;
    // 数据覆盖前缀
    if (c == 0x66)
    {
      diza->p_66 = 0x66;
      diza->defdata = (XDE32_DEFAULT_DATA^32^16)/8;
      continue;
    }
    // 地址覆盖前缀…
    if(c == 0x67)
    {
      … …
    }
    // 各种重复前缀
    if ((c == 0x26) || (c == 0x2E) || (c == 0x36) || (c == 0x3E) ||
      (c == 0x64) || (c == 0x65))
    {
      … …
    }
  } // 前缀解析到此为止
  // 前缀解析完后，c 就变成了操作码，取得指令特征
  flag |= xde_table[ TBL_NORMAL + c ];
  if (flag == C_ERROR) return 0;
  // 如果是 2 字节指令
  if (c == 0x0F)
  {
    c = *p++;
    // 根据第二字节取得指令特征
    flag |= xde_table[ TBL_0F + c ];
    if (flag == C_ERROR) return 0;
    diza->opcode2 = c;
```

```
        … … // 这里根据各种指令来获取影响到的寄存器和存储器
      }
      // 如果根据特征，有 ModR/M 字节，则解解析之
      if (flag & C_MODRM)
      {
         c = *p++;
         diza->modrm = c;
      }
      … …
      … … // 其他的步骤基本类似，这里一并略去
      … …
      return diza->len;     // 最后返回指令的总字节长度.
}
```

好了，我看到这里，基本理解了一个反汇编引擎的工作原理。限于作者的水平，没有详细的介绍。但是相对下面要用的技术，是非常足够了。也许有兴趣的话我们可以自己开发一个反汇编引擎。32 位的反汇编引擎有很多。但是 64 位的 i386 反汇编引擎似乎没有。（也许有但是没有找到？）。wowocock 为此很恼火，他说要自己写一个，不过他似乎一直没有时间。

3-3 XP 下 HOOK 系统调用 IoCallDriver

从这里开始，我希望用新的内核阅读能力，做一些更有用的事情。很多安全软件用了系统内核调用 HOOK 技术。当然威胁系统安全的软件也一样在使用着它们。这虽然是 MS 所不希望看到的，但却不是我们所可以不关心的。XP 下做任何已经导出的系统调用的 HOOK 都非常容易。这正是 HOOK 流行的原因。

IoCallDriver 是一个非常重要的调用。在这里可以过滤到所有的系统请求。IoCallDriver 的另一个常用的内部版本是 IofCallDriver. 几乎所有的内核驱动都调用了 IofCallDriver.即使你在开发中写下 IoCallDriver,编译后的代码中也只能发现 IofCallDriver 这个符号。IofCallDriver 的反汇编非常简单：

```
; Exported entry 42. IofCallDriver
; LONG __cdecl IofCallDriver(PDEVICE_OBJECT DeviceObject, PIRP Irp)
public @IofCallDriver@8
@IofCallDriver@8 proc near
jmp ds:_pIofCallDriver
@IofCallDriver@8 endp
```

几乎所有的系统调用都如此：进入之后，立刻出现一个 jmp …，跳到真实的调用处。这形成了一个系统调用跳转表。这真是一个糟糕的地方，因为只要修改这个地址，一切系统调用的 HOOK 都是简单轻松的了。

不过初学者可能会产生一些错误的想法：既然是 jmp ds:_pIofCallDriver,那么我只要把符号_pIofCallDriver 的值修改掉，一切就 OK 了。

有以下几个原则：

1．凡是你看见的代码，你都可以随意修改。

2．符号的值，不可以修改。

你看见的符号，在机器码中并非符号，而是已经编译得出的数字。假设_pIofCallDriver = 8099c99h,那么这条指令本质上是：

jmp ds: 8099c99h

你可以修改这个值，但是你无法修改_pIofCallDriver 这个符号。假设其他地方也调用了这个符号的话，你在这里对值的修改，并不影响其他地方对这个符号的调用。那些地方依然是 8099c99h。

这样一来，如果你想修改符号的值，似乎唯一的办法是对 8099c99h 进行全面查找替换。但是不幸的是，除了_pIofCallDriver 这个符号之外，其他的数据也可能为这个值。所以结论为：符号的值，不可以修改。

下面考试考虑修改 jmp ds: _pIofCallDriver,32 位下，jmp 后面直接就是 32 位地址。考虑前面看过的反汇编引擎的技术，很容易想到第一个字节是前缀，第二个字节是 jmp 这个操作码，后一字节就是跳转地址，为 32 位。当前要修改它还有一个前提，就是我能找到这个指令本身所在。然而这也难不倒我们。用 MmGetSystemRoutineAddress 很容易得到 IofCallDriver 的地址，而这个 jmp 正是第一条指令。那么写法如下：

```
// 首先定义一个函数指针类型
typdef NTSTATUS FASTCALL (*PMY_IOFCALLDIVER_FP)( IN PDEVICE_OBJECT,IN OUT PIRP);
// 以下函数用函数 newIofCallDriver 去代替现有的 IofCallDriver,同时返回旧的 IofCallDriver 所跳转
// 的实际地址。如果失败，则返回空。
PMY_IOFCALLDIVER_F MyHookIofCallDriverXP(
    IN PMY_IOFCALLDIVER_F newIofCallDriver,
    IN BOOLEAN hookOrUnhook)
{
    UNICODE_STRING functionName;
    PBYTE address;
    static PMY_IOFCALLDIVER_F oldIofCallDriverBody = NULL;
    RtlInitUnicodeString( &functionName, L"IofCallDriver" );

    // 得到 IofCallDriver 的入口地址
    address = MmGetSystemRoutineAddress( &functionName );
    if(address == NULL)
        return NULL;
    if(hookOrUnhook)
    {
        // 获得入口地址后，加 2 字节的地址就是旧的 IofCallDriver 的执行体的地址
        oldIofCallDriverBody = (PMY_IOFCALLDIVER_F)(*(PLONG)(address + 2));
        InterlockedExchange((PLONG)(address + 2),newIofCallDriver);
        return oldIofCallDriverBody;
    }
    else
    {
        // 复原
        if(oldIofCallDriverBody == NULL)
            return NULL;
        InterlockedExchange((PLONG)(address + 2), oldIofCallDriverBody);
        return oldIofCallDriverBody;
    }
}
```

以上函数不是线程安全的。应该避免多线程同时调用。调用后新的 IofCallDriver 将调用你设定的函数。同时你得到了旧的 IofCallDriver 的实现体指针。新的函数调用完后，你可以选择继续调用旧的或者不再调用而直接结束掉。从而形成 HOOK。

3-4 Vista 下 IofCallDriver 的跟踪

下面是我在 vista 对 IofCallDriver 的反汇编结果。我把阅读解释作为注释写在右边便于理解。可以看到，这不是一个简单的跳转过程。

```
nt!IofCallDriver:
81827e6b 8bff            mov     edi,edi                 ; 无意义指令
81827e6d 55              push    ebp                     ; F 指令，备份 ebp
81827e6e 8bec            mov     ebp,esp                 ; F 指令，备份 esp
81827e70 51              push    ecx                     ; F 指令，备份 ecx
81827e71 a17c1b9381      mov     eax,dword ptr [nt!pIofCallDriver (81931b7c)]
                                                         ; 取一个全局变量 pIofCallDriver
                                                         ; 的值。
81827e76 56              push    esi                     ; F 指令，备份 esi
81827e77 8bf1            mov     esi,ecx                 ; 取第一个参数。第一个参数是一个
                                                         ; PDEVICE_OBJECT device, 相当于
                                                         ; esi = device
81827e79 33c9            xor     ecx,ecx                 ; ecx = 0
81827e7b 3bc1            cmp     eax,ecx                 ; 判断如果 pIofCallDriver 为 NULL,
                                                         ; 则跳到 81827e88。如果 pIofCallDriver 不为 NULL,
                                                         ; 则直接直接结束函数。
81827e7d 7409            je      nt!IofCallDriver+0x1d (81827e88)
81827e7f ff7504          push    dword ptr [ebp+4]       ; ebp+4 是函数的返回地址。
81827e82 8bce            mov     ecx,esi                 ; 把参数传回 ecx
81827e84 ffd0            call    eax                     ; 呼叫 pIofCallDriver。
81827e86 eb47            jmp     nt!IofCallDriver+0x63 (81827ecf)        ; 跳到函数结束处
81827e88 fe4a23          dec     byte ptr [edx+23h]      ; irp->CurrentLocation--;
81827e8b 384a23          cmp     byte ptr [edx+23h],cl   ; if(irp->CurrentLocation < 0) 就出 bugcheck
81827e8e 7f0c            jg      nt!IofCallDriver+0x30 (81827e9c)
81827e90 51              push    ecx                     ; 这一段就出 BugCheck.
81827e91 51              push    ecx
81827e92 51              push    ecx
81827e93 52              push    edx
81827e94 6a35            push    35h
81827e96 e810070b00      call    nt!KeBugCheckEx (818d85ab)
81827e9b cc              int     3
81827e9c 8b4260          mov     eax,dword ptr [edx+60h] ; (PBYTE)irp->Tail.Overlay.
                                                         ; CurrentStackLocation -= 24
                                                         ; 这个等于 IoSkipCurrentIrpStackLocation 的
                                                         ; 逆操作
81827e9f 83e824          sub     eax,24h
81827ea2 894260          mov     dword ptr [edx+60h],eax

81827ea5 8a08            mov     cl,byte ptr [eax]       ; 这里取得 CurrentStackLocation 的主功能号
81827ea7 80f916          cmp     cl,16h                  ; 判断主功能号是否是 IRP_MJ_POWER
```

```
81827eaa 897014          mov     dword ptr [eax+14h],esi
81827ead 7514            jne     nt!IofCallDriver+0x57 (81827ec3)
81827eaf 8a4001          mov     al,byte ptr [eax+1]
81827eb2 3c02            cmp     al,2                        ；如果是 IRP_MN_SET_POWER，就跳 ba
81827eb4 7404            je      nt!IofCallDriver+0x4e (81827eba)
81827eb6 3c03            cmp     al,3                        ；如果不是 IRP_MN_QUERY_POWER，
                                                             ；就跳 c3
81827eb8 7509            jne     nt!IofCallDriver+0x57 (81827ec3)；
81827eba 8bf2            mov     esi,edx                     ；似乎是无意义指令，难道用 esi 传参数?
81827ebc e81bd1fdff      call    nt!IopPoHandleIrp (81804fdc)     ；这里是电源设置和电源查询之外的处理
81827ec1 eb0c            jmp     nt!IofCallDriver+0x63 (81827ecf)
81827ec3 8b4608          mov     eax,dword ptr [esi+8]       ；这里开始是除了电源设置和电源查询之外
                                                             ；的处理．取 device->DriverObject.
81827ec6 52              push    edx                         ；
81827ec7 0fb6c9          movzx   ecx,cl
81827eca 56              push    esi                         ；似乎是 std call 方式的调用方法.
81827ecb ff548838        call    dword ptr [eax+ecx*4+38h]   ；调用对应的分发函数.
81827ecf 5e              pop     esi
81827ed0 59              pop     ecx
81827ed1 5d              pop     ebp
81827ed2 c3              ret
81827ed3 90              nop
81827ed4 90              nop
81827ed5 90              nop
81827ed6 90              nop
81827ed7 90              nop
```

根据上面的解读，反 C 的结果如下:

```
NTSTATUS FASTCALL IofCallDriver(PDEVICE_OBJECT device, PIRP irp)
{
    PIO_STACK_LOCATION irpsp;

    // 首先检查一个全局变量 pIofCallDriver，如果这个不为空，则调这个，否则继续
    if(pIofCallDriver != NULL)
        return pIofCallDriver(device,irp);

    // 移动 irp 的 Current Stack Location.
    irp->CurrentLocation--;
    if(irp->CurrentLocation < 0)
    {
        KeBugCheckEx(0x35,irp,0,0,0,0);
    }
    irpsp = --irp->Tail.Overlay.CurrentStackLocation;
```

```
    // 如果是电源设置与查询，则调用特殊的 IopPoHandleIrp.
    if(irpsp->MajorFunction == IRP_MJ_POWER)
    {
        if(irp->MiniorFunction == IRP_MN_SET_POWER || irp->MiniorFunction == IRP_MN_QUERY_POWER)
        {
            return IopPoHandleIrp(device,irp);
        }
    }

    // 否则调用对应的 Device 的 Driver 的分发函数.
    return device->DriverObject->DispatchFunctions[irpsp->MajorFunction](device,irp);
}
```

上面的反 C 供有兴趣的读者研究．一般的说进行 HOOK 的话，只要看汇编指令就足够了。这里有个有趣的地方，似乎是微软为了自己使用的方便，留了一个全局变量名字为 pIofCallDriver，只要设置这个指针，IofCallDriver 就会调用这个函数，而忽略自己其他的处理．一般的跟踪表明，这个全局变量都为空．我们可以在这里加上我们的函数的地址来进行处理．这就形成了一个 HOOK.调用结束后，我们可以设法再 jmp 回来继续执行.

但是这样的方法过于依赖 IofCallDriver 这个函数的内部实现.pIofCallDriver 这个变量并没有导出,要搜索这个变量就不太容易.而且我将面向未来的 Vista 操作系统，而不是已经确定的 XP。微软随时可能修改这些代码。所以，强烈依赖于具体实现的方法是不可取的.

3-5 Vista 下 inline hook

wowocock 用了 inline hook 的方式。基本原理是，动态解析 IofCallDriver 开头的几条指令。并把他们拷贝到另一个地方。同时用一个调用我们的函数的代码加以替换。如果要继续执行旧的 IofCallDriver,在我们的函数调用完毕后，再执行被我们移动过的几条指令。执行完后后跳回原来的地方继续执行。具体步骤是:

第一，把 IofCallDriver 开头指令拷贝下来，移动到我自己的中继函数里。

第二，在 IofCallDriver 开头写入跳转指令跳转到我的中继函数。

第三，中继函数中执行对我自己的 IofCallDriver 钩子函数的调用。

第四，中继函数中执行原来 IofCallDriver 函数中拷贝过来的几条指令。

第五，跳转回原来的 IofCallDriver 后开始的跳转点继续执行。

下面假设我们的中转函数如下:

```
static __declspec(naked)    MyVistaIofCallDriverRelay();
```

这个函数将容纳 IofCallDriver 的开头一些指令。并跳转到我们设置的自己的 IofCallDriver 函数。完毕后继续执行旧的 IofCallDriver 的代码。naked 表明这个函数将不生成函数框架指令。这有利于我们控制生成的代码。

首先的问题是如何实现写入跳转指令。

跳转用 jmp 就可以实现。但是在计算地址的时候比较麻烦。jmp 后可以指定绝对地址和相对地址，但这段代码我们得写入 IofCallDriver 前部，这样相对地址就变了。此外除了 jmp 之外，很多指令都能实现跳转。有些老手不喜欢用 jmp，因为这容易被其他工具检测而暴露。

我们必须设法使用 jmp 的绝对地址跳转，或者自己计算相对地址。这都不是很简单。还好我们有反汇编引擎来帮助我们生成机器码。首先必须查一下 jmp 的指令手册，你发现当 jmp 的指令码为 ea 时，我们可以使用绝对地址。这样就免除了计算相对地址的麻烦。

使用前面的反汇编引擎的汇编功能如下:

```
size_t length;
byte_t code[12];            // 使用一个比较大的空间来容纳未知指令
struct xde_instr myjmp = { 0 };
myjmp.opcode = 0xea;            // 填写指令码 EAh
myjmp.addrsize = 4;             // 地址长度为 4 个字节（32 位）
myjmp.datasize = 2;             // 选择子作为立即数。长度为两个字节
myjmp.addr_l[0] = my_address;   // 填写我要跳转到的地址
myjmp.data_s[0] = 8;            // 选择子。内核代码段选择子为 8
length = xde_asm(code,&myjmp);
```

这样一来，一条绝对跳转指令就被写到了 code 中。以后可以把 code 中的代码拷贝到 IofCallDriver 的前部。

然后我们的任务仅仅是把这段代码拷贝到 IofCallDriver 的前部。这并不困难。IofCallDriver 的开始地址可以用前面 MmGetSystemRoutineAddress 的方法来获得。之前这些将被覆盖的代码必须先移动到另一个地方。这个地方将是我们自己模块的代码段。如果是一般的情况，代码段是不允许写的，这会引发系统异常。但是用下面的代码可以清除这一设置：

```
dword_t old_cr0;
_asm
{
    mov eax,cr0
    mov old_cr0,eax
    and eax,0xfffeffff
    mov cr0,eax
}
```

当然最后你还要恢复它。

```
_asm
{
    mov eax,old_cr0
    mov cr0,eax
}
```

下面的任务是拷贝代码。我们前面得到的 jmp 指令为 7 个字节。就是说我们至少要拷贝出 7 个字节的代码。我们不能只拷贝 7 字节。指令长度不定，这可能把一条指令分成两段。我们只好逐条执行进行反汇编。当得到的总的字节数达到或者超过 7 个字节的时候，大功告成。下面假设 IofCallDriver 的开始地址为 start_address.

```
size_t length,total_length = 0;
struct xde_instr code_instr={0};
byte_t *start_address = (byte_t *)MmGetSystemRoutineAddress(…);
while(total_length < 7)
{
    length = xde_disasm(start_address ,&code_instr); // 反汇编一条指令
    if(length == 0)             // 如果有指令解析失败，就直接返回失败
        return false;
    total_length += length;  // 计算已经反汇编的指令的总长度
}
```

得到长度后就简单了，拷贝即可。这些代码将拷贝到我们的中继函数中去。这中间最好不要调用 C 库函数，同时提高中断级禁止线程切换，以免节外生枝发生其他的问题：

```
KIRQL irql;
```

```
irql =   KeRaiseIrqlToDpcLevel();      ; 提高中断级
_asm {
      mov esi,start_address            ; IofCallDriver 的原来的开始地址
      mov edi,g_new_address            ; 这里写入我要拷贝到哪里
      mov ecx,total_length             ; 总长度
      cld
      rep movsb                        ; 拷贝

      …                                ; 然后是写入前面的 jmp 指令的过程，这是和前面是一样的。

}
KeLowerIrql(irql);
…
```

最后是中继函数的实现。这个函数的流程前面已经介绍过了，除了耐心细致的做好堆栈平衡工作之外，有人可能疑惑如何容纳旧的部分代码然后执行。你可以这样实现：

```
static __declspec(naked)    MyVistaIofCallDriverRelay()
{
      … …
      _asm {
old_codes:
            nop                ; 这里写入足够多的 nop,形成一片空间来容纳拷贝过来的旧代码。
            nop
            nop
            …
      };
      … …
}
```

不过又出来一个新的问题，那就是 old_codes 这个标号对应的具体地址如何传出，以便外面完成拷贝。既然在这个函数内部，容易得到这个地址，那么我可以在这个函数执行的早期，把这个地址传入一个叫做 g_new_address 的全局变量里。

```
static byte_t *g_new_address = NULL;
…
static __declspec(naked)    MyVistaIofCallDriverRelay()
{
      _asm push old_codes
      _asm pop g_new_address
      …
      …
      _asm {
old_codes:
            nop                ; 这里写入足够多的 nop,形成一片空间来容纳拷贝过来的旧代码。
            nop
            nop
            …
```

```
    };
    … …
}
```

这样只要 MyVistaIofCallDriverRelay 执行过一次，就会把 old_codes 这个地址写入全局变量 g_new_address 中。当然，你得在 hook 之前，执行一次这个函数，并在写 g_new_address 前加一些条件判断防止执行后面的代码出错。

这样的 hook 还是有一些局限的。在被拷贝的代码中，不能有各种跳转，否则相对地址会发生错误。这限制了我们拷贝地址的长度，也限制了我们加入其他的处理。比如说，我要设置我的函数执行完毕后，是否继续执行后面的代码，或者直接返回。

网上已经有一些库通过反汇编引擎修改各种跳转指令的地址，以便可以拷贝大部分的指令，你可以参考它们。

3-6 总结与展望

我们费了很多的工夫，从基础的汇编，到阅读 windows 内核，到了解反汇编引擎，但是最后只给一个系统调用做了一个 HOOK,真是大费周折呢。如果项目实际需要，也许从网络上搜索一段 IofCallDriver HOOK 的代码，不就轻松搞定了吗。还好这不是实际项目，只是一个不太轻松的娱乐过程。

我还在读书的时候，技术类的书籍特别喜欢看侯捷写的书，尤其崇拜他在《深入浅出 MFC》中每章标题后的寄语：“勿在浮沙筑高台”，“唯有深入，可以浅出”。VC 类的书，真是堆积如山，但是看了大多是不明就里，做起程序来依然是碍手碍脚。惟独看了《深入浅出》那几句话，恍然大悟。有时候做技术，略去一些不需要关心的细节，使项目大大加快，但是却也培养了自己的惰性。真的碰到问题时，束手无策。要回头重头学起，时间又不济，心慌意乱，最后不得不放弃了事。与其贪图速度，不如一开始就打好基础，无论研究什么，都追根揭底，搞个明白。看似麻烦，却是似远实近的选择。

Vista 下许多内核代码都改变了。微软大幅度升级，如果内核代码还是原样，那用户的大笔钞票可真是白出了。改变内核代码相信使系统运行得更合理，更有效率，或者更安全。但是新写的代码，肯定会带来更多想不到的 bug，或者是给第三方的内核开发者带来麻烦。

我就碰到过这样的事情。在 WindowsXP 下认为绝对不会重入的代码，在 Vista 下居然重入导致内核调用栈耗光蓝屏了。微软当然可以说它没有错。因为如果我不是相信了他文挡上所说的事实，而利用了这样的特性的话，Vista 依然是稳定的。只是目前事实改变，所以大家一起崩溃罢了。但这绝对是微软的新人写的新代码犯的愚蠢错误。

现在相信所有的第三方的内核开发者，都不得不重头检查他们的代码，并小心的搬到 Vista 上去调试。不要指望微软会他的新手程序员们给系统带入的新“特性”写入文档中。我们怎么办呢？网络上的资料大多还停留在 XP。那么我们只好自己看吧。这就是我们付出努力的目的。

遗憾的是我写这些内容的时候，还没有得到能运行的 Vista64 位版本。所以也只能写到 32 位系统为止。期待后来者的继续努力。本书的试读版本到此为止，你可以把它视为一本完整的小册子。（完）