

信息对抗综合设计实验 II

实验指导书

计算机科学与工程学院
(网络空间安全学院)

实验一 机器码探索与病毒初见

为了在windows阶段进行复杂的病毒实验以及相关的逆向分析，我们需要首先对机器指令建立基本的认知，并具备对其进行操控的能力。本实验将带领同学们熟悉x86指令的分析方法，同时通过“猜测—实证—构建”的方式自主探索求证未知的领域，具备最为基本的底层基本观念和工具能力。

本实验的目的主要是通过调试器，自己设计实验分析机器码的格式，并设计实验修改机器码，完成实证。本实验包含对x86指令集中比较简单的mov指令的分析，并在此基础上分析在函数调用和二进制安全领域有较大作用的call指令。

根据上面的猜测—实证训练，我们通过调试器实现代码补丁的方式修改程序的流程，完成病毒机制中获取执行权限的过程。

一、实验环境

操作系统：Windows XP及之后的版本，如Windows 7、Windows 10等

应用软件：VC++ 6.0及Visual Studio更高版本

二、实验内容和任务

(1) 用调试器分析全局变量赋值的汇编语句。要解决以下几个问题，

a. 赋值语句到底修改了几个字节

b. mov 指令的机器码格式猜测。由那几部分组成。设计实验观察验证猜测的格式。

c. 修改机器码，验证修改的效果

d. 补充题目：自己生成机器码执行

(2) 修改正常函数调用路径，让其调用未调用的函数。并返回到原来的路径，并完成正常的调用。

三、实验原理讲解

我们从下面这段简单的为全局变量赋值语句的反汇编开始

```
int gi;  
  
void main() {  
  
    gi = 12;  
  
}
```

使用vs2008作为调试环境。点击鼠标将光标放在gi=12这行语句上，
然后按f9，在该行上打上断点（如果继续按f9将消除该断点）。

断点：
即当程序运行时，经过该点所在语句，程序将暂停运行。我们可在此时观察程序的各种状态，如变量值、内存值、寄存器；甚至可以修改这些相关状态。英文是 break point。我们可以在程序运行前打断点，也可以在运行中打断点。

如图1.1所示

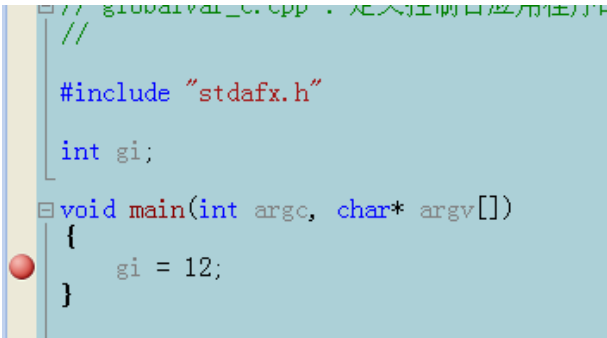
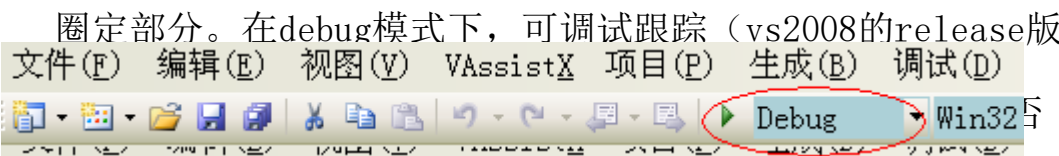


图 1.1 添加断点

然后保证程序为debug模式（缺省即是debug）。见图1.2中红色椭圆



则无法调试跟踪）。 图 1.2 debug 版本

我们按键f5，进入调试运行方式（ctrl+f5为非调试运行方式，该方

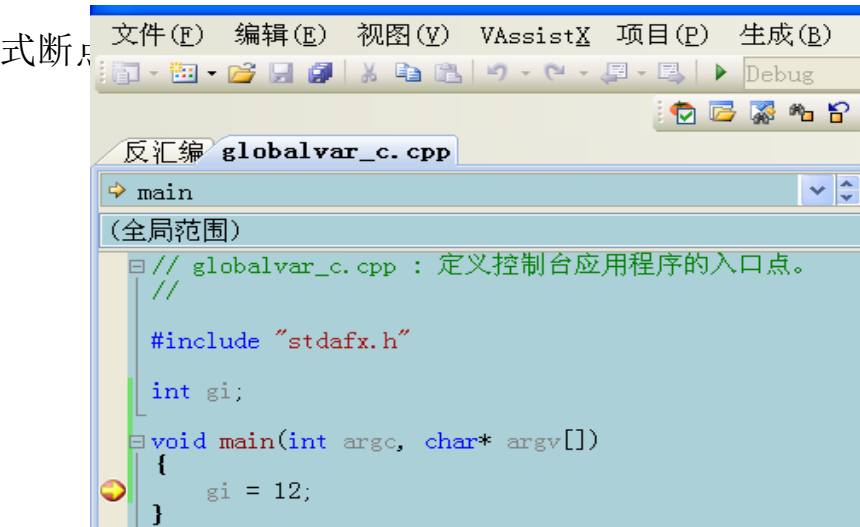


图 1.3 程序在断点处暂停

打开反汇编，见图1.4，单击箭头所示反汇编子菜单（快捷键为组合键ctrl+alt+d，即同时按ctrl,alt和d三个键，反汇编的英语为disassemble）。

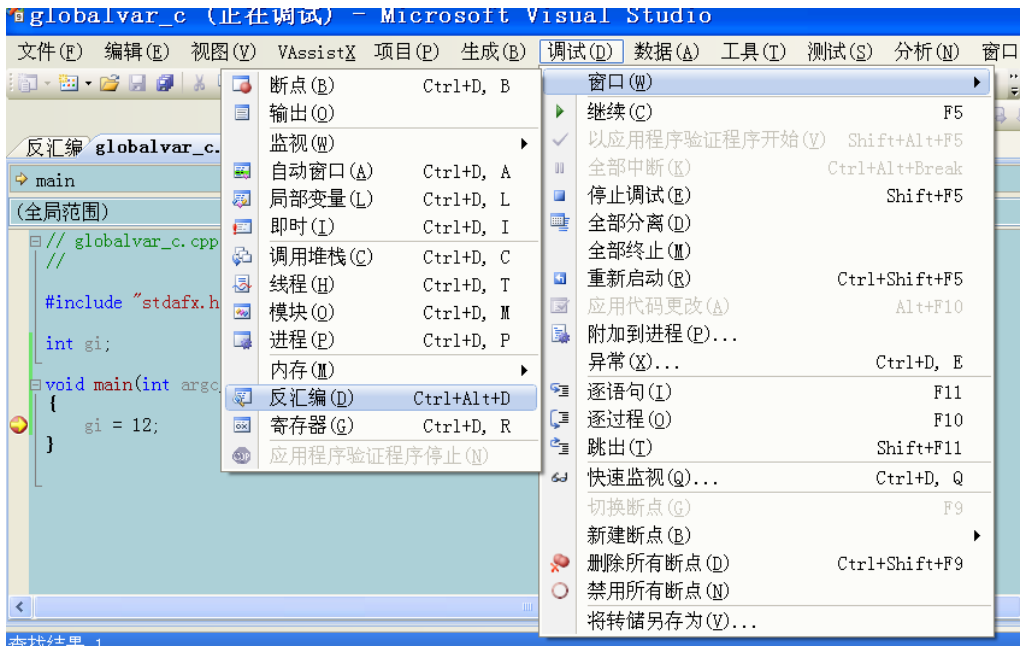


图1.4 反汇编菜单

看到如下反汇编：

```
1  gi = 12;

2  0041138E  mov             dword ptr [gi (417140h)], 0Ch

3  }
```

在上述代码中，第2行是第1行对应的汇编码。第2行左边的数字0041138E是16进制表示的赋值指令mov的存放地址，右边是该条指令

的汇编表示形式。学习过汇编的读者会奇怪，那个gi，也就是全局变量的名字怎么出现在汇编指令中呢？汇编语言是没有变量名啊？因为在vc环境中，为了易理解，开发环境特意将操作对应的符号名也显示在语句中，我们一看就知道这条指令是对变量gi操作。为了看到纯正的汇编语句，我们选择暂时将符号名字从反汇编语句中去除。在代码窗口中点击鼠标右键激活弹出菜单，点击箭头所示的显示符号名，取消其选中的打钩状态（如果需要显示符号，则再次点击）见图1.5。

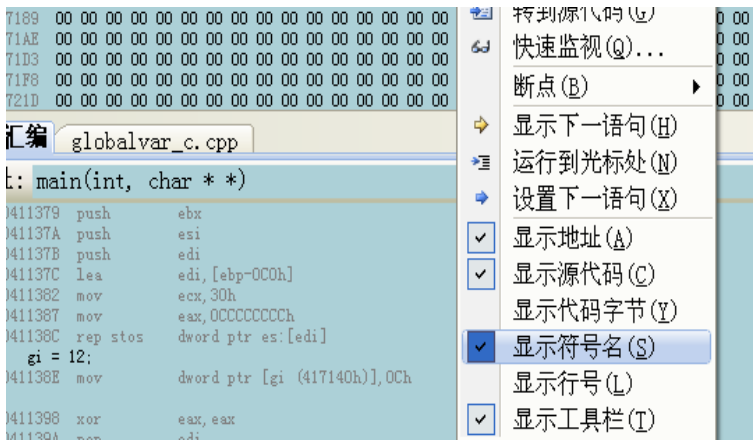


图1.5

此时反汇编如下：

```
gi = 12;
0041138E  mov     dword ptr ds:[00417140h], 0Ch
}
```

注意，此时mov指令中没有C语言的变量名gi了。下面解释该条mov指令的含义，它把0ch（h代表十六进制，即0c即十进制的12。大

家应该将a至f的值对应10进制的值背熟，便于以后快速分析问题）赋值给内存。内存地址为指令方括号中十六进制数所示，即00417140h。

计算机的信息是存储在内存中，通过 mov 指令将数据存放在指定地址的内存中。C 语言的全局变量本质上是一块内存，所以对它的存取也是通过地址进行存取。
mov [地址值], 存储值; 该指令将存储值存入方括号内地址值指向的内存中。注“mov ds:[地址值]...”中的 ds 是数据段寄存器，在 x86 寻址中，是段寄存器和段内偏移合在一起定位（这有点像用街道名和门牌号一起定位的方式）。windows 和 linux 所有段寄存器都指向一个位置，所以相当于只有一个段。因此段寄存器可以不用管。

从这里我们就开始探索式学习了。“问题”是探索式学习的关键。

探索mov指令，来看看它的机器码。在代码窗体中点击鼠标右键，激活弹出菜单并选中图1.8箭头所示的菜单“显示代码字节”。

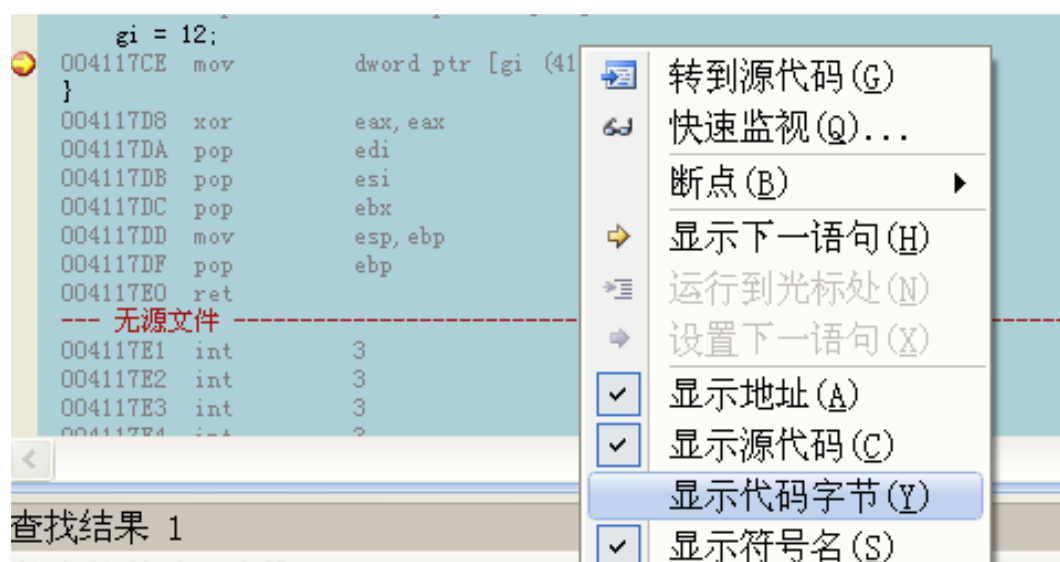


图 1.8 显示反汇编机器码

下面为mov指令的机器码，它处于指令存放地址和汇编指令中间，黑体显示。

```
gi = 12;
```

```
0041138E      C7 05 40 71 41 00 0C 00 00 00      mov dword
                                     ptr ds:[00417140h],0Ch
                                     }
```

现在我们分析这个机器码中包含的和mov指令相关的信息。首先还是猜测，该指令应该包含了哪些信息：

- (1) 要赋的值12
- (2) 要赋值的内存地址00417140h
- (3) 表明该指令是mov指令

按照猜测，我们在给出的机器码中来实证一下：

```
C7 05 40 71 41 00 0C 00 00 00
```

修改源码反汇编查看汇编码的实验：

1 修改赋值的常量，查看机器码的变化。先打断点，可以通过反汇编窗体拿到mov指令的地址。再通过内存窗体找到该地址的内容。并将我们认为的部分修改。然后恢复运行。观察变量的打印值是否变成我们希望的值。

2 直接修改mov指令中的地址部分，验证效果。

通过代码补丁，改变程序流向的病毒机制模拟：

(1) 修改exe执行文件中的机器码

修改执行文件中的机器码（1）

- 将0c修改为0b，则打印结果应该是11

```
004113D7 B8 CC CC CC CC  mov     eax,0CCCCCCCCh
004113DC F3 AB                rep stos dword ptr es:[edi]
gi = 12;
004113DE C7 05 64 71 41 00 0C 00 00 00  mov     dword ptr [gi (417164h)
printf("%d\n", gi);
004113E8 8B F4                mov     esi,esp
004113EA A1 64 71 41 00        mov     eax,dword ptr [gi (417164h)]
```

修改执行文件中的机器码（2）

- 为了找到对应机器码，则需选出被查找的指令 c7 05 64 71 41 00 0c，然后在 **ultraedit** 中寻找

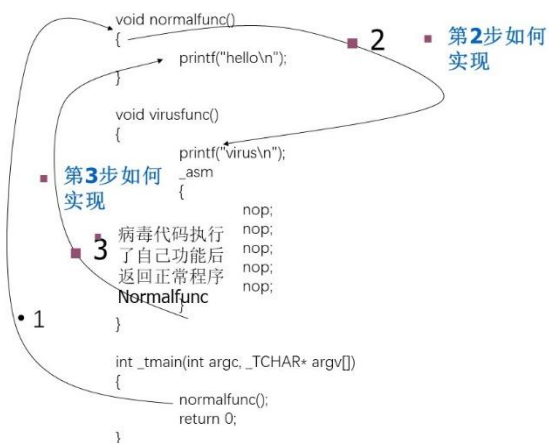
```
004113D7 B8 CC CC CC CC  mov     eax,0CCCCCCCCh
004113DC F3 AB                rep stos dword ptr es:[edi]
gi = 12;
004113DE C7 05 64 71 41 00 0C 00 00 00  mov     dword ptr [gi (417164h)
printf("%d\n", gi);
004113E8 8B F4                mov     esi,esp
004113EA A1 64 71 41 00        mov     eax,dword ptr [gi (417164h)]
```

修改执行文件中的机器码 (5)

- 执行test1.exe,结果如下,打印结果变成了11

```
E:\>cd E:\mydocs\教学\病毒\实验\病毒初识\修改执行文件机器码\test\Debug
E:\mydocs\教学\病毒\实验\病毒初识\修改执行文件机器码\test\Debug>test1.exe
11
E:\mydocs\教学\病毒\实验\病毒初识\修改执行文件机器码\test\Debug>
```

(2) 通过内存进行代码补丁



1. 把把头5个字节改成jmp指令使其跳到virusfunc函数中, 我们需要计算jmp的偏移量

- 把把头5个字节改成jmp指令使其跳到virusfunc函数中, 我们需要计算jmp的偏移量

```
void normalfunc()
{
    004113C0 55          push     ebp
    004113C1 8B EC       mov      ebp, esp
    004113C3 81 EC C0 00 00 00 sub      esp, 0C0h
    004113C9 53          push     ebx
    004113CA 56          push     esi
    004113CB 57          push     edi
}
```

1.D 计算jmp的偏移

- 计算jmp的偏移，从0x4113c0开始，占5字节，其后的指令起始为0x4113c5，从0x4113c5跳到0x411440，这样偏移是0x411440 - 0x4113c5=7b。其小端机内存表示为7b 00 00 00

```
void normalfunc()
{
004113C0 55          push     ebp
004113C1 8B EC       mov     ebp, esp
004113C3 81 EC C0 00 00 00 sub     esp, 0C0h
004113C9 53          push     ebx
004113CA 56          push     esi
004113CB 57          push     edi

printf("virus\n");
0041143E 8B F4       mov     esi, esp
00411440 68 44 57 41 00 push    offset string "virus\n" (415744h)
00411445 FF 15 BC 82 41 00 call    dword ptr [__imp_printf (4182BCh)]
0041144B 83 C4 04    add     esp, 4
}
```

1.C 覆盖原来的指令

- 用eb 7b 00 00 00覆盖Normalfunc的入口，即0x4113c0开始的5字节。用内存窗体完成。先在入口打断点，运行中断后，找到反汇编窗体，并在内存窗体中地址栏输入0x4113c0找到该指令内存，修改之

内存 1

地址: 0x004113C0

0x004113C0: e9 7b 00 00 00 c0 00

0x004113C6: 7b 6e 82 41 00 83 c5

```
void normalfunc()
{
004113C0 E9 7B 00 00 00 jmp     virusfunc+20h (411440h)
004113C5 C0 00 00 00 00 rol     byte ptr [eax], 0
}
```

2. 覆盖virusfunc打印指令的Normalfunc

- 用Normalfunc的被jmp覆盖的三条指令覆盖virusfunc打印指令结束后的内容。这样保证Normalfunc能正常执行。
- 用内存窗体修改41144e开始的内存为那三条指令55 8B EC 81 EC C0 00 00 00

问题：为什么要从41144e开始覆盖，可以向前或向后么

3. 完成Normalfunc的正常工作

- 在上节用来覆盖的三条指令后，加一个jmp使其跳回Normalfunc，从而完成该函数的正常工作。

Jmp偏移量就是 4113c9 - 41145c = fffffff6d, 小端机内存表示6d ff ff ff, 最终指令为e9 6d ff ff

3. 在内存中查找病毒代码

内存 1
地址: 0x00411457

0x00411457: e9 6d ff ff 8b 81 c4 c0 00 00 00 3b ec e8 e0 fc ff
0x0041147D: cc cc cc 55 8b ec 81 ec c0 00 00 00 53 56 57 8d bd 40
0x004114A3: 33 c0 5f 5e 5b 81 c4 c0 00 00 00 3b ec e8 95 fc ff ff
0x004114C9: 25 bc 82 41 00 cc cc 75 01 c3 55 8b ec 83 ec 00 50 52
0x004114EF: 5a 58 8b e5 5d c3 cc cc cc cc cc cc cc cc cc cc 8b

内存 1 寄存器

反汇编 test.cpp

地址: virusfunc(void)

0041143C F3 AB rep stos dword ptr es:[edi]
printf("virus\n");
0041143E 8B F4 mov esi, esp
00411440 68 44 57 41 00 push offset string "virus\n"
00411445 FF 15 BC 82 41 00 call dword ptr [__imp__prin
0041144B 83 C4 04 add esp, 4
0041144E 55 push ebp
0041144F 8B EC mov ebp, esp
00411451 81 EC C0 00 00 00 sub esp, 0C0h
nop;
00411457 E9 6D FF FF jmp normalfunc*9 (4113C9h)

- 在内存窗体地址栏输入jmp指令起始地址0x411457，回车。改其开始的5字节如图。
- 可见反汇编窗体最后一行出现了jmp指令，它跳回了Normalfunc

(3) 文件中进行代码补丁

1.a 寻找normalfunc的入口点

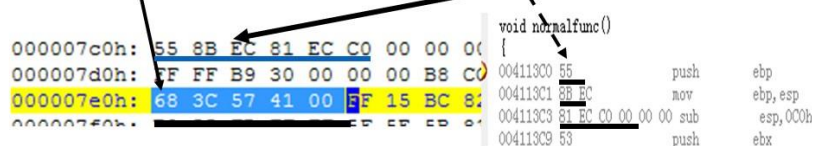
- 为了找到normalfunc的位置，我们应该寻找其中的标志性机器码，那就是包含被打印字串的地址的指令68 3c 57 41 00

```
printf("hello\n");  
004113DE 8B F4 mov esi, esp  
004113E0 68 3C 57 41 00 push offset string "hello\n" (41573Ch)  
004113E5 FF 15 BC 82 41 00 call dword ptr [__imp__printf (4182BCh)  
004113EB 83 C4 04 add esp, 4
```

- 在ultraedit中ctrl+f查找68 3c 57 41 00

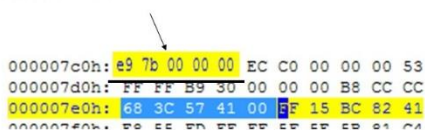


normalfunc入口机器码(从vc反汇编获取)在
68 3c 57 41 00之前,即 55 8b ec 81...



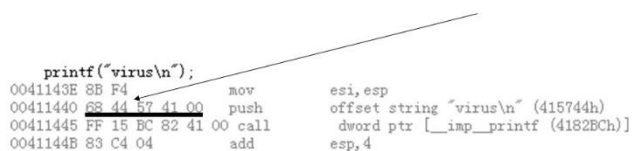
1.b 修改normalfunc入口指令

- 根据前面内存修改时的结果, 该jmp指令为E9 7b 00 00 00

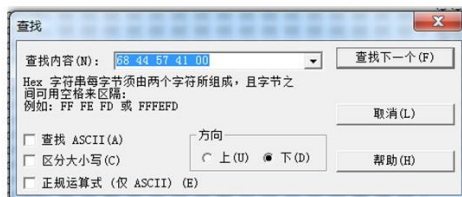


2.a找到virusfunc

- 为了找到virusfunc的位置, 我们应该寻找其中的标志性机器码, 那就是包含被打印字符串的地址的指令68 44 57 41 00



- 在ultraedit中ctrl+f查找68 44 57 41 00



Virusfunc被覆盖机器码(从vc反汇编获取)在
68 44 57 41 00之后,即 3b f4 ...开始

```

68 44 57 41 00 FF 15 BC 82 41 00 83 C4 04 3B F4 ;
E8 F5 FC FF FF 90 90 90 90 90 5F 5E 5B 81 C4 C0 ;
printf("virus\n");
0041143E 8B F4      mov     esi,esp
00411440 68 44 57 41 00 push    offset string "virus\n" (415'
00411445 FF 15 BC 82 41 00 call    dword ptr [__imp_printf (4:
0041144B 83 C4 04      add     esp,4
0041144E 3B F4      cmp     esi,esp

```

2.b 修改virusfunc指令

- 根据前面内存修改时的结果, 3b f4开始的内容
修改为normalfunc被覆盖的3条指令值55 8B EC
81 EC C0 00 00 00

```

68 44 57 41 00 FF 15 BC 82 41 00 83 C4 04 55 8B ;
EC 81 EC C0 00 00 00 90 90 90 5F 5E 5B 81 C4 C0 ;
void normalfunc()
{
004113C0 55      push    ebp
004113C1 8B EC   mov     ebp,esp
004113C3 81 EC C0 00 00 00 sub     esp,0C0h
004113C9 53      push    ebx

```

3 增加转跳指令跳回Normalfunc

- 紧接着刚才填充的指令，我们从90开始的5个字节填充为jmp，即e9 6d ff ff ff

68 44 57 41 00 FF 15 BC 82 41 00 83 C4 04 55 8b
ec 81 ec c0 00 00 00 e9 6d ff ff ff 5B 81 C4 C0

- Jmp的计算参考内存模拟的计算

