

# 信息对抗综合设计实验二

## 机器码探索与病毒初见

电子科技大学 计算机科学与工程学院

李忻洋

**virusli@uestc.edu.cn**





- 专注于二进制安全，主要是恶意代码攻防相关的实践
- 相关课程
  - 计算机病毒原理与防范
  - 网络与系统攻击技术
  - 计算机系统与网络安全
- 反汇编与调试、逆向分析、PE结构与病毒感染...
- 提升动手能力，避免“一听全会，一做全废”
- 飞书群：

# 注意事项



- **没有考试，没有考试，没有考试！！！！**
- **成绩：4次实验**
  - **课堂签到作为报告评分的重要参考**
  - **请在规定的提交时间之前提交，逾期不交则无成绩**
  - **提交后请注意检查是否是最新版本，是否错交为其他课程的报告**
  - **按照实验中心要求，请到机房使用学生端程序提交**
  - **按照学校要求，严禁抄袭（包括将报告发给其他同学参考）**

# 信息对抗综合设计实验二

## 机器码探索与病毒初见

代码初识天地现  
混沌内存沙盘演  
一寸转跳知玄关  
终寻立锥问硬盘





代码初识天地现  
混沌内存沙盘演  
一寸转跳知玄关  
终寻立锥问硬盘



求实求真 大气大为

1

反汇编





## 是否能学会?

### ► 专注于汇编指令及机器码本身

去除硬件相关部分，如各类接口、中断控制、外设IO等  
围绕猜测-实证-构建的模式进行  
以全局变量赋值为案例，展现这一过程

### ► 授课对象

调试技术：大一程序设计学生 大一新生课外实践项目学生（包含非计科专业）

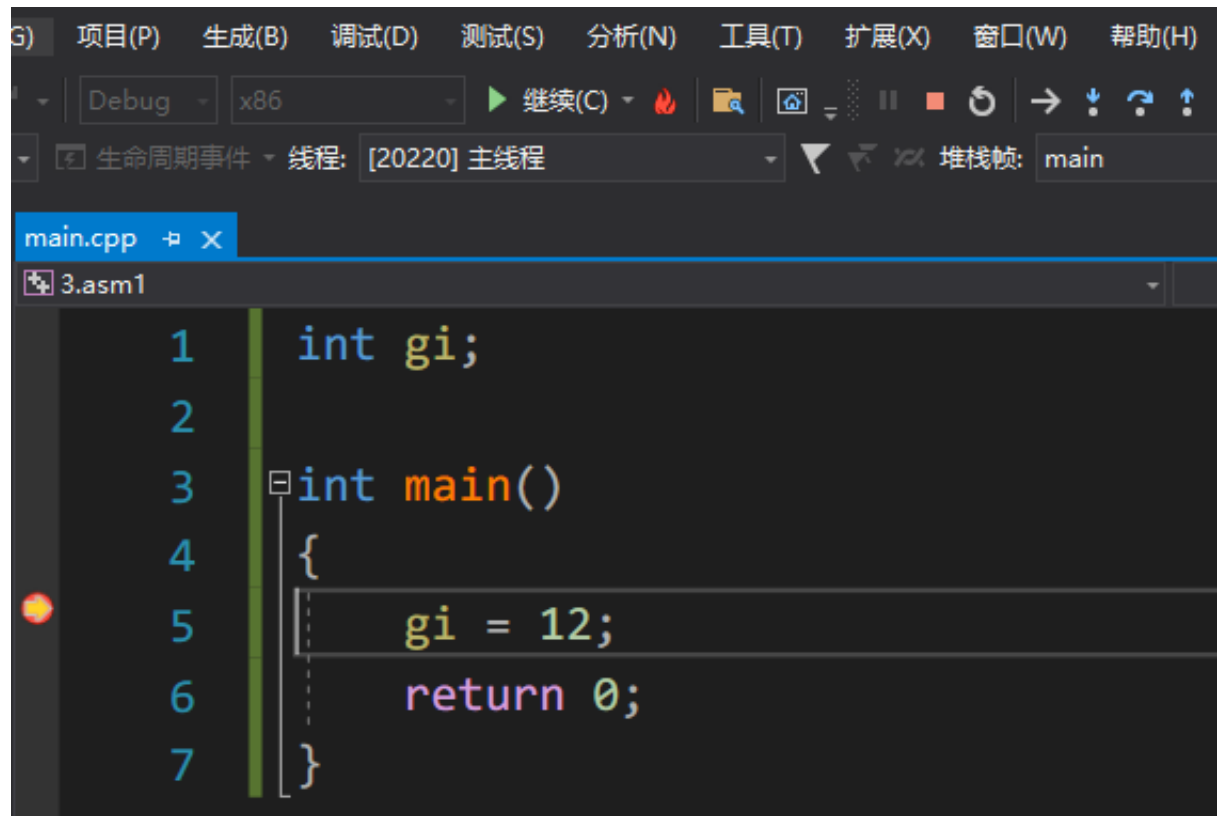
反汇编技术：

- 软件开发系统级技术基础课程（大三，外语学院例子）
- 工程组新生（大一、大二招新，具有C程序设计基础）
- 课程设计（针对计科二学位）

## 从赋值语句开始（最简案例）

```
int gi;  
int main() {  
    gi = 12;  
    return 0;  
}
```

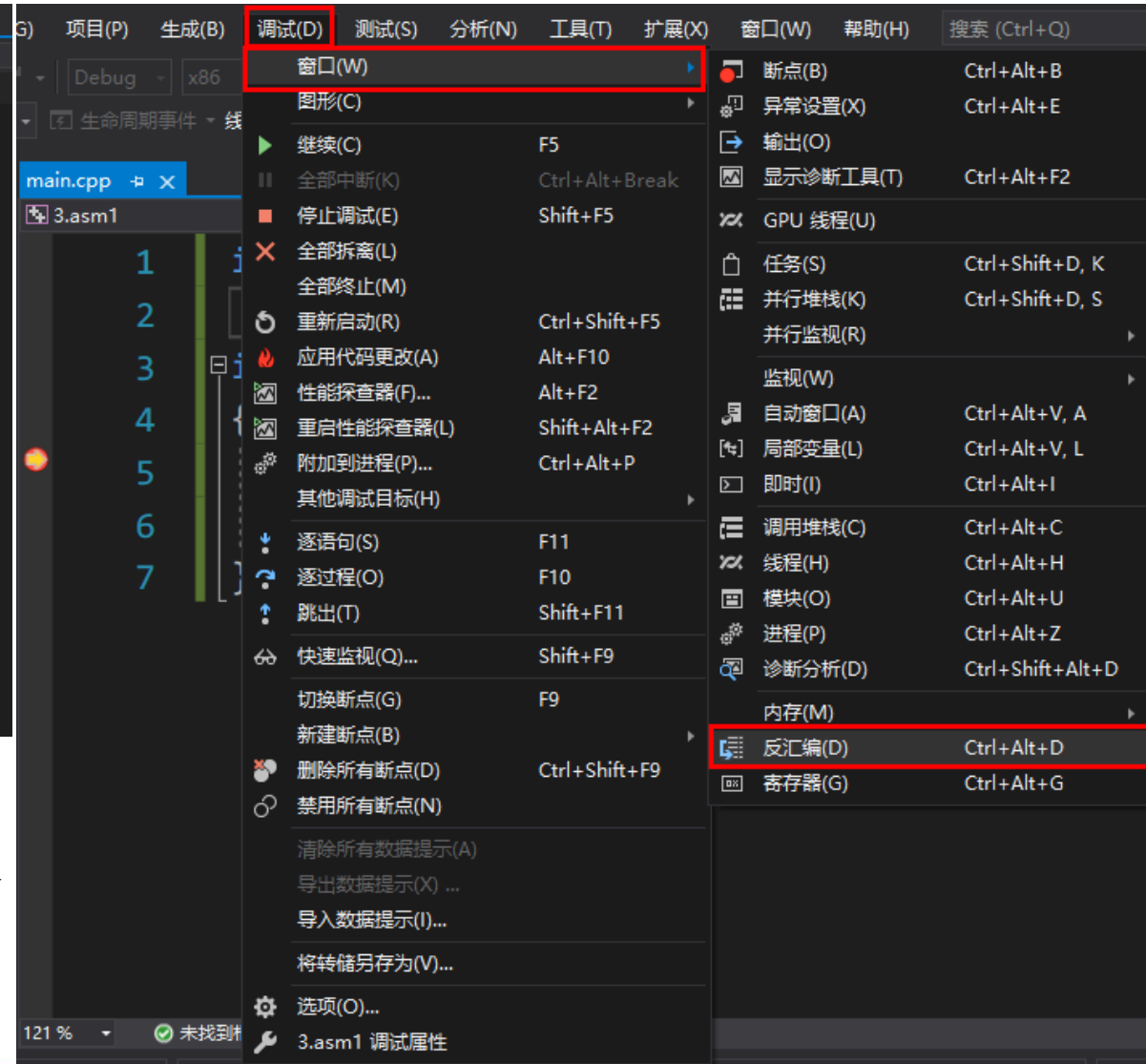
- 将光标置于目标语句处，按F9下断点
- F5调试运行目标程序
- 如左图所示



## 查看反汇编



```
反汇编 反汇编 main.cpp
地址(A): main(void)
查看选项
006E1765 rep stos dword ptr es:[edi]
006E1767 mov ecx,offset _678C007C_main@cpp (06EC000h)
006E176C call @_CheckForDebuggerJustMyCode@4 (06E1307h)
gi = 12;
006E1771 mov dword ptr [gi (06EA138h)],0Ch
return 0;
006E177B xor eax,eax
}
006E177D pop edi
006E177E pop esi
006E177F pop ebx
006E1780 add esp,0C0h
006E1786 cmp ebp,esp
006E1788 call __RTC_CheckEsp (06E1230h)
```



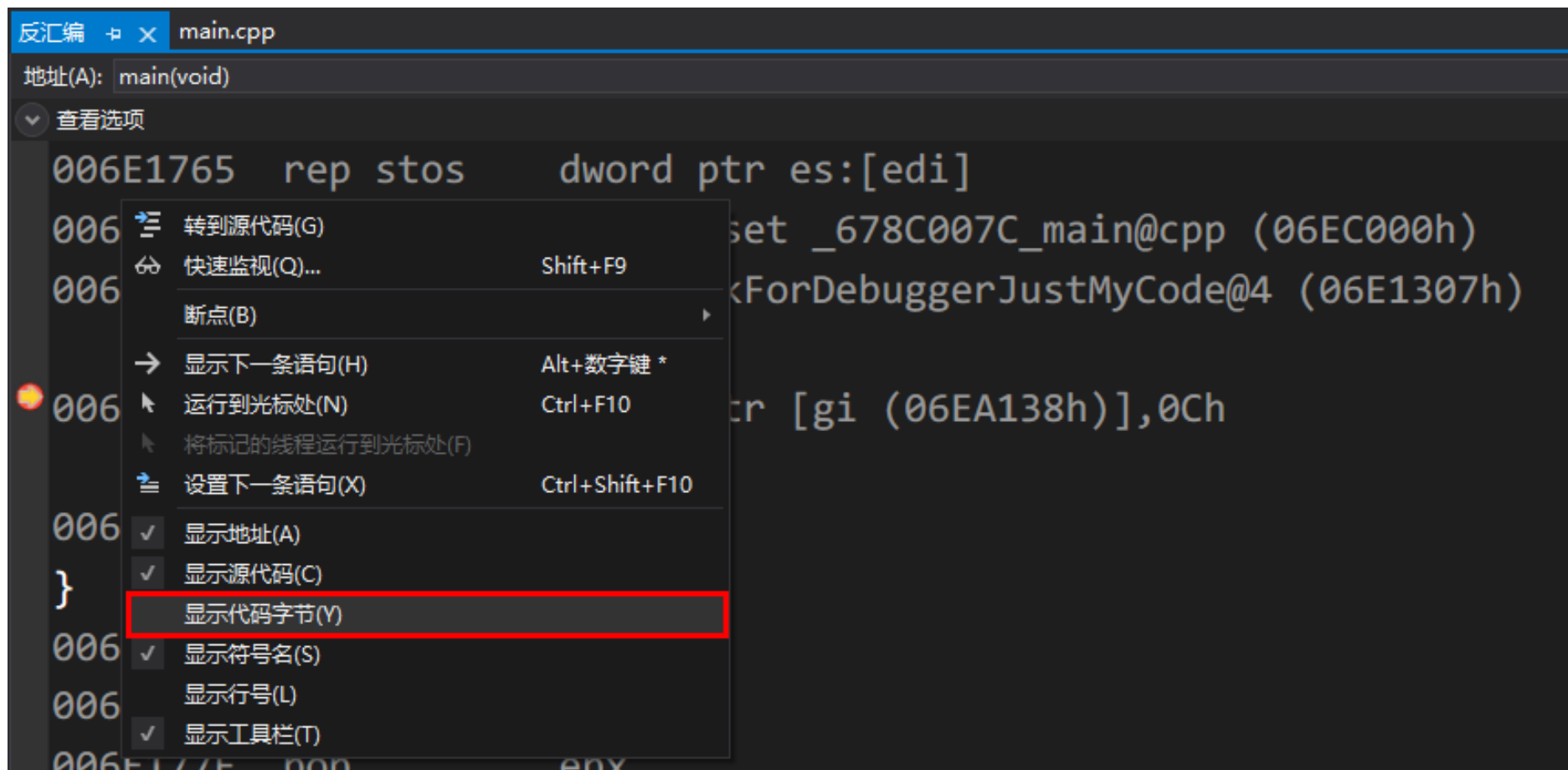
- 如右图，打开反汇编
- 上图即是C语言对应`gi = 12`的反汇编语句了
- 接下来我们查看它的机器码





## 查看机器码

- 在反汇编窗口中鼠标右键单击，如上所示
- 勾选显示代码字节



# 反汇编



```
反汇编  x main.cpp
地址(A): main(void)
查看选项
gi = 12;
006E1771 C7 05 38 A1 6E 00 0C 00 00 00 mov     dword ptr [gi (06EA138h)],0Ch
return 0;
006E177B 33 C0                                xor     eax,eax
}
```

006E1771 C7 05 38 A1 6E 00 0C 00 00 00  
mov dword ptr [gi (06EA138h)],0Ch

赋值语句的这一排可以分成三个部分。

- 红色部分是赋值语句的起始地址006E1771。
- 绿色的部分是mov语句的机器码。
- 蓝色的部分是mov指令的汇编表示。

将12（就是十六进制0Ch）赋值给内存地址为6EA138h的内存。

注意：[gi (06EA138h)]不是标准的汇编写法，它表明要赋值的变量地址是6EA138h。而提示大家该地址就是变量gi的地址



## 设问、猜测、实证、构建

实践学习计算机的很有效的方法，通过它能学到很多书本外的东西，且能自己不断地发现和深入。**我们不相信结论，我们要实证！**面对上页谈到的东西，大家想想有什么可以发问和存疑的东西呢？

```
gi = 12;
```

```
006E1771 C7 05 38 A1 6E 00 0C 00 00 00
```

```
mov dword ptr [gi(06EA138h)],0Ch
```

- ▶ gi的地址真的是06EA138h吗？
- ▶ C7 05 38 A1 6E 00 0C 00 00 00真的是mov指令的机器码吗？
- ▶ 006E1771真的是mov语句的起始地址吗？

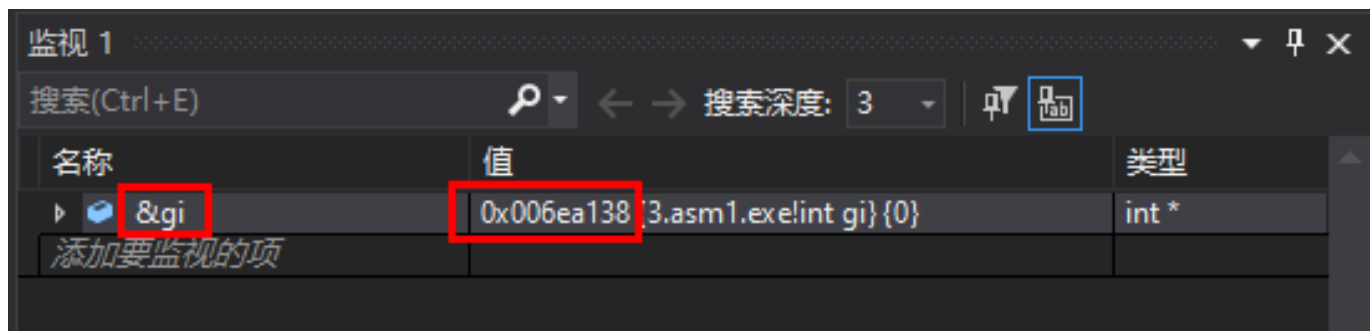
## gi的地址真的是06EA138h吗？

- ① `printf( "gi address=%x \n" , &gi);` 要修改代码
- ② 运用调试器，不修改源码而观察目标程序

设问后的猜测非重要

基本知识的运用就非常关键，调试技术成为基本技能

在监视窗口中输入&gi查看gi的地址值，从下图可见gi的地址确实和语句中的一样 `dword ptr [gi (06EA138h)]`, 都是0x006EA138h





## 除此之外，能直接查看内存变化吗？

实证可以有不同的办法，对于刚才的问题，我们想直接看内存的变化来看看是否真的mov语句将12这个值赋值给了0x006E1771这个地址。

因此，可以使用调试器的内存查看能力。在内存窗口地址栏中输入0x006E1771，回车后可见第一排最左边的一列的值就是0x006E1771，右边就是内存中的值，一个个字节排开显示的，我们要观察这里的变化

```
gi = 12;
006E1771 C7 05 38 A1 6E 00 0C 00 00 00 mov     dword ptr [gi 06EA138h],0Ch
return 0;
```

121 %

内存 1

地址: 0x006EA138

地址	0x006EA138	0x006EA14E	0x006EA164	0x006EA17A
00 00 00 00 01 00 00 00 24 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 01 00 00 00 24 00 00 00 00 00 00 00 00 00 00 00	00 00 d2 10 6e 00 00 00 00 00 02 00 00 00 00 00 00 00 01 00 00	ff ff	ff ff 00
.....\$.....	.....\$.....	..?.n.....	.....	.....

## 直接查看内存变化

单步执行，我们执行一条语句（F10）。注意看截图中的变化，第一个字节变成了红色的0c，即十进制的12。确定将12设定给gi也就是地址为0x006E1771的地址了。红色代表这个字节的内容和单步执行前不同。**问题：针对这个赋值变化，我们有问题，猜测和实证吗？**

```
gi = 12;
006E1771 C7 05 38 A1 6E 00 0C 00 00 00 mov     dword ptr [gi (06EA138h)],0Ch
return 0;
006E177B 33 C0                                xor     eax,eax    已用时间 <= 1ms
```

121 %

内存 1

地址: 0x006EA138

地址	字节	ASCII
0x006EA138	0c 00 00 00 01 00 00 00 24 00 00 00 00 00 00 00 00 00 00 00 00 00	.....\$. ....
0x006EA14E	00 00 d2 10 6e 00 00 00 00 00 02 00 00 00 00 00 00 00 01 00 00	..?.n.....
0x006EA164	ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff	.....
0x006EA17A	ff ff 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

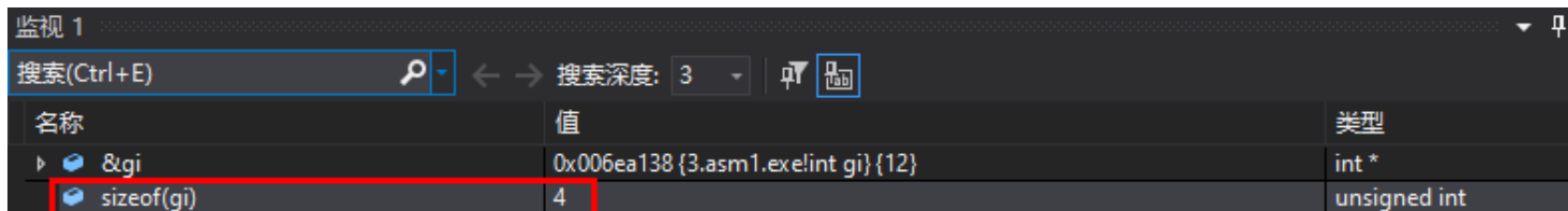
**新的问题：赋值语句到底修改了几个字节？** 是只修改了一个字节码？

思路一：在断点停下后，我们可以先将0c相连的字节修改成一些其他的值，比如11，这样如果赋值语句修改了，则字节颜色比如变化。如下：



## mov修改了几字节

思路二：有部分同学想起了C语言课上学到的sizeof(int)，在监视窗口中输入“sizeof(gi)”我们也可以得到gi的大小。

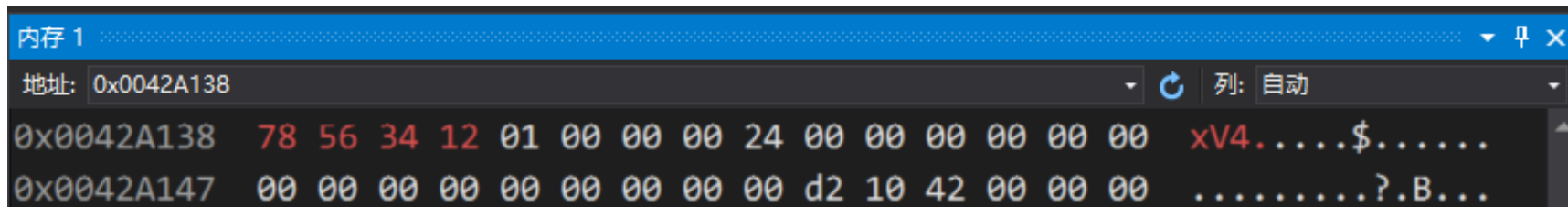


监视 1

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
&gi	0x006ea138 {3.asm1.exe!int gi}{12}	int *
sizeof(gi)	4	unsigned int

既然是4字节的变量，我们可以尝试赋值gi = 0x12345678来探查一下赋值对内存影响。



内存 1

地址: 0x0042A138 列: 自动

0x0042A138	78 56 34 12 01 00 00 00 24 00 00 00 00 00 00	xV4.....\$. ....
0x0042A147	00 00 00 00 00 00 00 00 00 00 d2 10 42 00 00	.....?.B...

修改字节数我们确定了就是4字节，但又出现了新的问题，观察上图，请问我们的新问题是什么？

**为什么0x12345678在内存中的排列顺序是：78 56 34 12，而不是 12 34 56 78？我们能通过实验找出其规律吗？**





# 内存字节序

再修改程序，观察gi = 0x1234后的内存情况

```
内存 1
地址: 0x00E6A138 列: 自动
0x00E6A138  34 12 00 00 01 00 00 00 24 00 00 00 00 00 00 00  4.....$. ....
0x00E6A147  00 00 00 00 00 00 00 00 00 d2 10 e6 00 00 00 00  .....
```

观察前面修改的语句：12, 0x1234 , 0x12345678。他们在内存中分别是：

内存中的值	逻辑值
0C 00 00 00	0x0C
34 12 00 00	0x1234
78 56 34 12	0x12345678

我们应该总结出一个什么规律？

逻辑上的低字节放在物理上的低字节（intel是小端机），不妨动手实验下0xAA55的结果。



## 小结与后续问题

我们在解决mov dword ptr [gi (06E1771h)], 0ch中gi的地址真的是0xE6A138h这个问题时。通过试探的方法，学习到了如下技能：

1. 调试器的监视功能，可观察高级语言表达式和变量。
2. 调试器的内存查看方法，以及修改方法
3. 整数的长度，以及整数的大小端机表示

gi = 12;

006E1771 C7 05 38 A1 6E 00 0C 00 00 00

mov dword ptr [gi (06EA138h)], 0Ch

**关于mov指令，还有两个没有解决的问题**

1. C7 05 38 A1 6E 00 0C 00 00 00 真的是mov指令的机器码吗？能看出两者的联系吗？
2. 006E1771 C7 05 38 A1 6E 00 0C 00 00 00 其中006E1771真的是mov语句起始地址吗

**如何实证动手验证上述两个问题呢？**



## mov机器码

gi = 12;

006E1771 C7 05 38 A1 6E 00 0C 00 00 00

mov dword ptr [gi (06EA138h)],0Ch

上述机器码中一定要包含哪些信息？

1. mov指令本身的机器码
2. 包含要赋值的12
3. 包含要被赋值的gi的地址

运用我们的前面才学的大小端机，和整数长度的原理，能看出来吗？如果看出来了，请再做一个实验验证一下



## mov机器码

gi = 12;

006E1771 C7 05 38 A1 6E 00 0C 00 00 00

mov dword ptr [gi (06EA138h)],0Ch

上述机器码中一定要包含哪些信息?

1. mov指令本身的机器码
2. 包含要赋值的12
3. 包含要被赋值的gi的地址

| **C7 05 (mov指令)** | **38 A1 6E 00 (地址006EA138h)** | **0c 00 00 00 (代表值12)** |

mov指令本身

要被赋值的gi的地址

要赋值的12

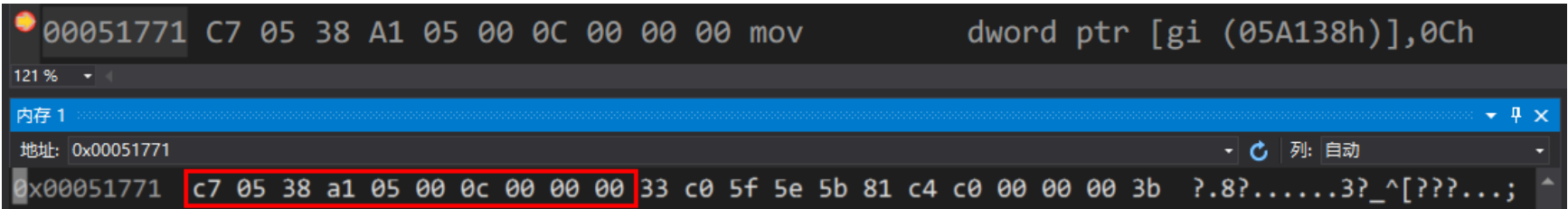
我们需要修改红色的地址部分的值以及后面常量的值,来看看这条指令的效果变化。先证明mov指令确实在006E1771。

需要注意的是,在Win7及以上系统中使用VS时,由于ASLR地址随机化机制的影响,程序加载到内存的基地址可能发生变化。



## mov机器码

需要注意的是，在Win7及以上系统中使用VS时，由于ASLR地址随机化机制的影响，程序加载到内存的基地址可能发生变化。



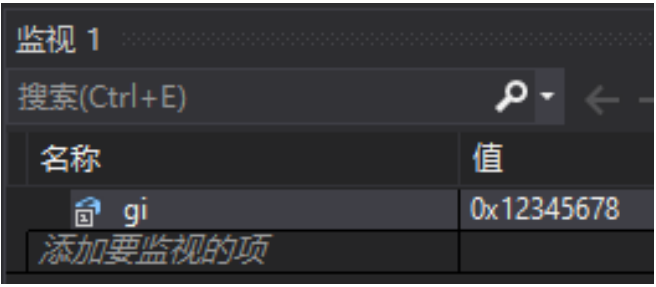
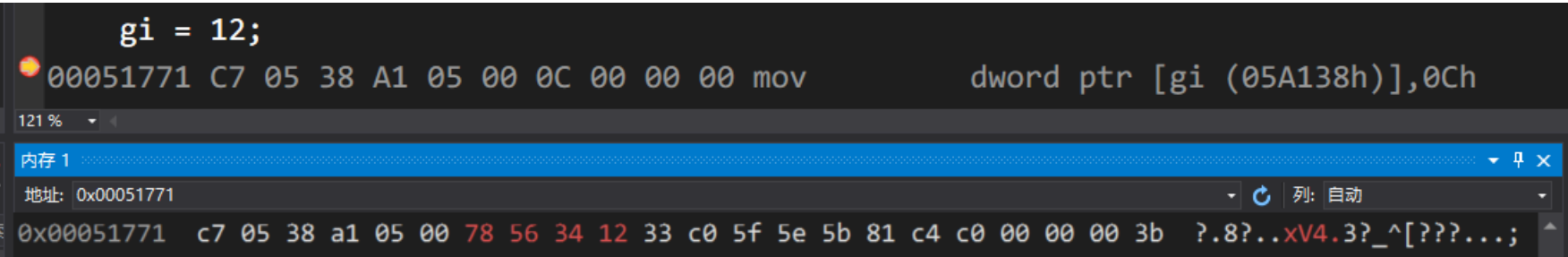
| **C7 05 (mov指令)** | **38 A1 05 00** (地址**0005A138h**) | **0c 00 00 00** (代表值**12**) |

mov指令本身

要被赋值的gi的地址

要赋值的12

通过内存窗体修改指令最后4字节为0a 00 00 00，单步后观察gi的值





## 任务一：验证mov机器码

- ① 定义两个全局变量g1和g2
- ② 写一个c语言的赋值语句 `g1 = 12;`
- ③ 在`g1=12`下断点，运行后中断到该行时，获取对应mov指令地址
- ④ 用内存窗体找到上一步中的mov指令
- ⑤ 将c7 05后的四个字节，如后的下划线 `c7 05 _ _ _ _`，修改成g2的地址。
- ⑥ 单步执行mov，用watch查看g2的内容将变成12

# 反汇编

## 小结

### 猜测-实证-构建



代码初识天地现  
混沌内存沙盘演  
一寸转跳知玄关  
终寻立锥问硬盘



求实求真 大气大为

2

病毒  
内存模型







## 基本能力



### 执行过程的特点？

- 窃取执行权：将执行流程从正常执行的指令修改为执行病毒指令
- 归还执行权：病毒执行完后，恢复执行被感染对象正常的指令（潜伏）

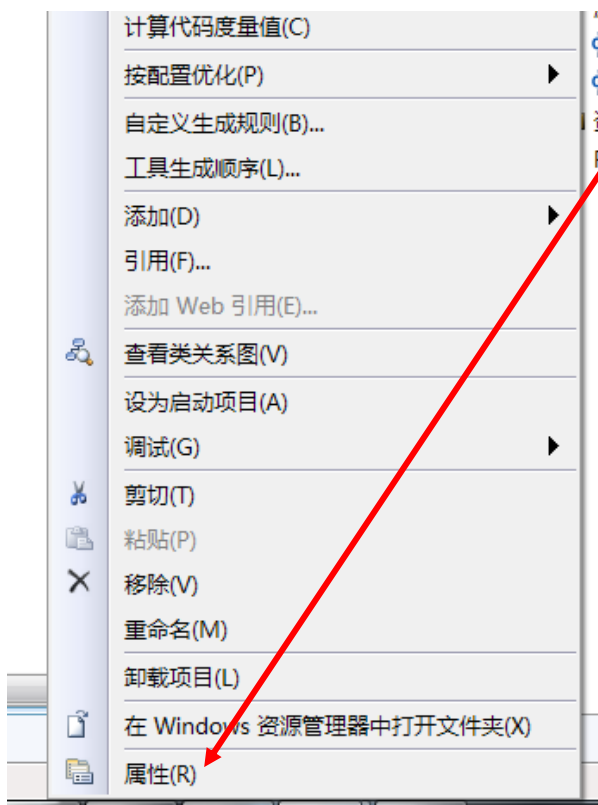
### 如何模拟？通过修改内存机器码！

- 正常执行流程：normalfunc
- 病毒执行流程：virusfunc

## 避免重定位表带来的影响



- 鼠标选项目图标，然后鼠标右键激活菜单，选则“属性”菜单。

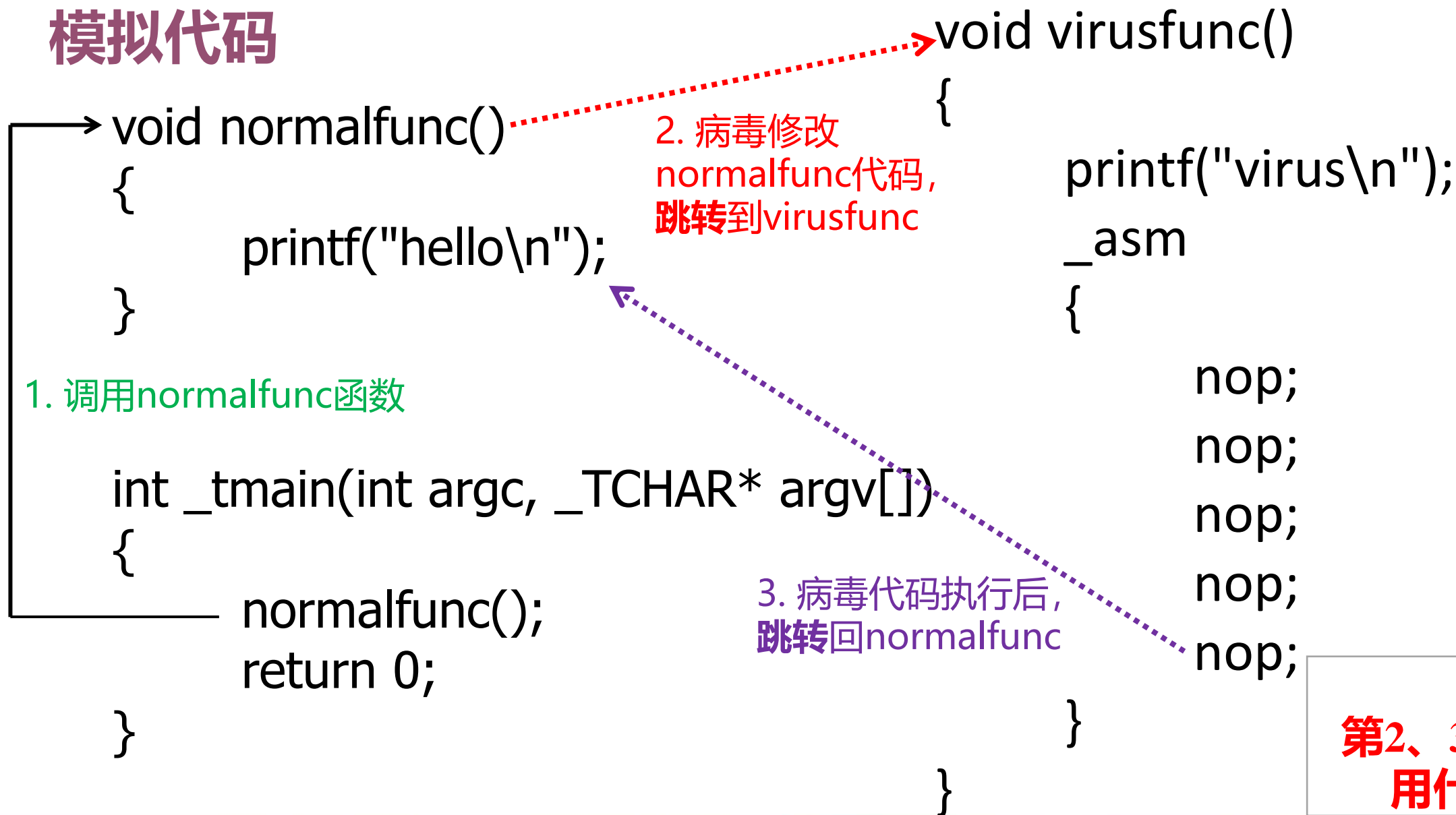


## 避免重定位表带来的影响

- 链接器--高级--随机基址，选择禁止随机化属性



## 模拟代码



第2、3步需要使用什么指令?



## “一寸跳转知玄关”：JMP的力量

地址1	JMP <b>offset</b>
地址2	CMP EAX, EBX
...	...
地址3	MOV EAX, ECX

**Offset = 地址3 - 地址2**

CPU执行指令的过程：

1. 将当前指令放入流水线中
2. EIP指向**下一条指令**
3. 执行当前指令

JMP跳转的偏移量是：

**目的地址-JMP的下一条指令地址！**

E9是JMP指令的机器码，假定Offset为04，则JMP对应的机器码是  
e9 04 00 00 00，占5个字节

↑	↑
jmp	offset

# 病毒模型

1 找到normalfunc入口的位置，  
写入e9 xx xx xx xx.其中，xx  
xx xx xx代表转跳到virusfunc  
中打印指令的偏移量

```
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
```

被修改的3条指令

```
55      1
8B EC   1
81 EC C0 00 00 00
```

2 找到virusfunc中的打印指令，  
在其后修改指令，填充内容为  
normalfunc中被第一步jmp指  
令覆盖的3条完整指令

```
virusfunc
00411440 68 44 57 41 00    push    offset string "virus\n" (
00411445 FF 15 BC 82 41 00 call     dword ptr [__imp_printf
0041144B 83 C4 04          add     esp, 4
0041144E 3B F4             cmp     esi, esp
00411450 E8 F5 FC FF FF    call    @ILT+325(__RTTI_CheckEsp)
asm
{
    nop;
00411455 90                nop
```

共需要添加2  
条jmp指令！

3 在第2步填充的3条指令后填  
写jmp 指令使其跳回第1步覆盖  
的3条指令后

```
E9 6d ff ff ff
```

## “一寸跳转知玄关”：JMP的力量

地址1	JMP <b>offset</b>
地址2	CMP EAX, EBX
...	...
地址3	MOV EAX, ECX

**Offset = 地址3 - 地址2**

CPU执行指令的过程：

1. 将当前指令放入流水线中
2. EIP指向**下一条指令**
3. 执行当前指令

JMP跳转的偏移量是：

**目的地址-JMP的下一条指令地址！**

E9是JMP指令的机器码，假定Offset为04，则JMP对应的机器码是  
e9 04 00 00 00，占5个字节

↑	↑
jmp	offset

## JMP偏移量的计算

- JMP指令从0x4113c0开始, 占5字节
- 后续指令以0x004113c5起始
- 从0x4113c5跳到0x411440, 偏移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  
004113C8 55          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
```





## 内存中修改原有指令

- 用e9 7b 00 00 00覆盖normalfunc的入口，即0x4113c0开始的5字节。
- 定位地址：先在入口下断点，断下后找到反汇编窗体，确定指令位置

```
void normalfunc()  
{  
  004113C0 55  
  004113C1 8B EC
```

- 在内存窗体中地址栏输入0x4113c0，定位到目标内存，进行修改

内存 1

地址: 0x004113C0

0x004113C0	e9 7b 00 00 00	c0 00
0x004113E6	1b bc 82 41 00 83	ca

- 观察反汇编窗口指令的变化

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



## 指令恢复

1 找到normalfunc入口的位置，  
写入e9 xx xx xx xx.其中，xx  
xx xx xx代表转跳到virusfunc  
中打印指令的**偏移量**

### normalfunc

004113C0	55		bp
004113C1	8B EC		bp, esp
004113C3	81 EC C0 00 00 00		esp, 0C0h
004113C9	53	push	ebx
004113CA	56	push	esi

### 被修改的3条指令

55	]
8B EC	]
81 EC C0 00 00 00	

2 找到virusfunc中的打印指令，  
在其后修改指令，填充内容为  
normalfunc中被第一步jmp指  
令覆盖的3条完整指令

### virusfunc

00411440	68 44 57 41 00	push	offset string "virus\n" (
00411445	FF 15 BC 82 41 00	call	dword ptr [__imp_printf
0041144B	83 C4 04	add	esp, 4
0041144E	3B F4	cmp	esi, esp
00411450	E8 F5 FC FF FF	call	@ILT+325(__RTX_CheckEsp)
		asm	
		{	
		nop;	
00411455	90	nop	

3 在第2步填充的3条指令后填  
写jmp 指令使其跳回第1步覆盖  
的3条指令后

E9 6d ff ff ff

代码初识天地现  
混沌内存沙盘演  
一寸转跳知玄关  
终寻立锥问硬盘

UNIVERSITY OF  
ELECTRONIC  
SCIENCE  
AND TECHNOLOGY OF CHINA



求实求真 大气大为

3

病毒  
文件模型





## 模拟病毒

上节实验的本质是修改机器码，只要内存中和硬盘中最终执行的机器码没有不同，则**两者等价**。

意义：忽略了执行程序文件格式的复杂因素，直指本问题的本质。







## 文件修改

使用文本编辑工具进行修改：C32ASM

找到对应机器码，进行修改

```
gi = 12;
```

```
006E1771 C7 05 38 A1 6E 00 0C 00 00 00
```

```
mov dword ptr [gi (06EA138h)],0Ch
```

**在硬盘文件中：**如果我们将机器码中的0c修改为0b，则打印结果也应该是11



## 文件修改

### 搜索机器码

- 使用C32ASM工具以16进制编辑方式打开目标exe
- Ctrl+F, 以16进制方式搜索C7 05 38 A1 6E 00 0C 00 00 00
- 搜索到后, 进行目标指令的修改, 然后保存
- 为保险, 可再搜索一次, 确认全局只有一处



## 定位normalfunc入口

### 搜索机器码

```
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
```

**“特征码”：尽可能独特的标志性机器码**

**被打印字串的地址的指令68 3c 57 41 00**

**你可以尝试选取其他机器码作为特征码，如55 8B EC，搜索看看结果**

## 有去有回



- 在恢复被覆盖的三条指令后，加一个jmp使其跳回normalfunc，从而完成该函数的正常工作。

void normalfunc()  
{  
004113C0 55  
004113C1 8B EC  
004113C3 81 EC C0 00 00 00  
004113C9 55  
004113CA 56  
004113CB 57

push  
mov  
sub  
push  
push  
push

反汇编 test.cpp  
地址: virusfunc(void)  
0041143C F3 AB rep stos dword ptr  
printf("virus\n");  
0041143E 8B F4 mov esi, esp  
00411440 68 44 57 41 00 push offset str  
00411445 FF 15 BC 82 41 00 call dword ptr  
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  
00411457 e9 XX XX XX XX  
0041145c ...

←

- Jmp偏移量就是  $4113c9 - 41145c = ffffffff6d$ ,  
小端机内存表示6d ff ff ff, 最终指令为e9 6d ff ff ff





## 实验任务

- 任务1：尝试一下修改机器码的执行
  - 1) 定义两个全局变量g1,g2
  - 2) 写一个c语言的赋值语句g1=学号后6位
  - 3) 利用watch窗口输入&g1, &g2查看其对应的内存地址
  - 4) 在“g1=学号后6位”下断点，运行后，查看对应mov指令机器码的指令地址
  - 5) 用内存窗体找到（4）中的mov指令机器码存放的指令地址
  - 6) 将c705后的四个字节，如后的下划线c7 05\_\_\_\_，修改成g2的内存地址
  - 7) 单步执行mov，用watch窗口查看g2的内容将变成12
- 任务2：在内存和硬盘上重现以上病毒模型的执行过程，virusFunc中打印学号后六位
- 任务3：在内存中模拟病毒时，取消nop，当篡改normalFunc跳转到virusFunc时，跳转到virusFunc的第一条指令然后在virusFunc最后一条ret语句处，进行对被覆盖字节的恢复，并跳回normalFunc，看是否能顺利完成病毒。

UNIVERSITY OF  
ELECTRONIC  
SCIENCE  
AND TECHNOLOGY OF CHINA



求实求真 大气大为

4

扩展  
内容





- 调试器与断点背后的原理：异常处理机制
- 代码钩子的思想与中间人攻击
- 在编写钩子库时的坑
  - 线程重入问题
  - 指令截断问题



- 演示断点背后的秘密——int 3

# 谢谢

THANK YOU

感谢聆听 欢迎指正

