

王爽汇编第九章, 转移指令的原理

1. 转移指令
2. 操作符offset
3. jmp指令
4. 根据位移进行转移的jmp指令
5. 插播HOOK知识
 - 5.1. Inline Hook
 - 5.2. Inline Hook 原理
 - 5.3. Hook代码开发
6. 转移的目的地址在指令中的jmp指令
7. 转移地址在寄存器或内存中的jmp指令
8. jcxz指令和loop指令
 - 8.1. jcxz指令
 - 8.2. loop指令
9. 几种跳转指令和对应的机器码

1. 转移指令

可以修改IP, 或同时可以修改CS和IP的指令统称为: 转移指令

8086CPU 的 转移行为 有以下几类:

- 只修改IP时, 称为段内转移, 比如: jmp ax。
- 同时修改CS和IP时, 称为段间转移, 比如: jmp 1000:0。

由于转移指令对 IP的修改范围 不同, 段内转移又分为: 短转移和近转移。

- 短转移IP的修改范围为-128~127(0xFF)
- 近转移IP的修改范围为-32768~32767(0xFFFF)

8086CPU 的 转移指令 分为以下几类:

- 无条件转移指令 (jmp)
- 条件转移指令 (jnz jz..)
- 循环指令 (loop)
- 过程
- 中断 (int)

2. 操作符offset

操作符offset是伪指令, 在汇编语言中由编译器处理的符号, 它的功能是 取得标号的偏移地址


```

1  assume cs:code
2
3  code segment
4  main:
5      mov ax,0
6      jmp short s; 注意s不是被翻译成了目的地址，而是根据第一条规则计算成了位移
7      add ax,1
8      s:inc ax;程序执行后,ax的值为1
9  code ends
10 end main

```

编译后标号被翻译成了位移。

AX=FFFF BX=0000 CX=0009 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=0000 NV UP EI PL NZ NA PO NC
076C:0000 B80000 MOV AX,0000
076C:0003 EB03 JMP 0008
076C:0005 B3C001 ADD AX,+01
076C:0008 40 INC AX
076C:0009 00E8 ADD AL,CH
076C:000B 3300 XOR AX,[BX+SI]
076C:000D E83000 CALL 0040
076C:0010 E82D00 CALL 0040
076C:0013 E82A00 CALL 0040
076C:0016 E82700 CALL 0040
076C:0019 E82400 CALL 0040
076C:001C E82100 CALL 0040
076C:001F E81E00 CALL 0040

公式: (位移) = (标号) - jmp后(地址)

CPU不需要这个目的地址就可以实现对IP的修改。这里是依据位移进行转移。

jmp short s指令的读取和执行过程如下：

- (1) (CS)=076C,(IP)=0000, 执行完 `mov ax,0` 后CS:IP指向了 `EB 03` (jmp short s机器码);
- (2) 读取指令码 `EB 03` 进入指令缓冲器;
- (3) (IP) = (IP) + 所读取指令的长度 = (IP) + 2 = 5, CS:IP指向了add ax,1;
- (4) CPU执行指令缓冲器中的指令 `EB 03` ;
- (5) 指令执行后 `IP+位移` =(IP) + 3 = 8, CS:IP(076C:0008) 指向->inc ax

5. 插播HOOK知识

这里插播点 X86 平台下的hook知识，看到这里我们已经知道了我们可以通过修改ip的方式来让代码跳到想要的地方执行代码。

5.1. Inline Hook

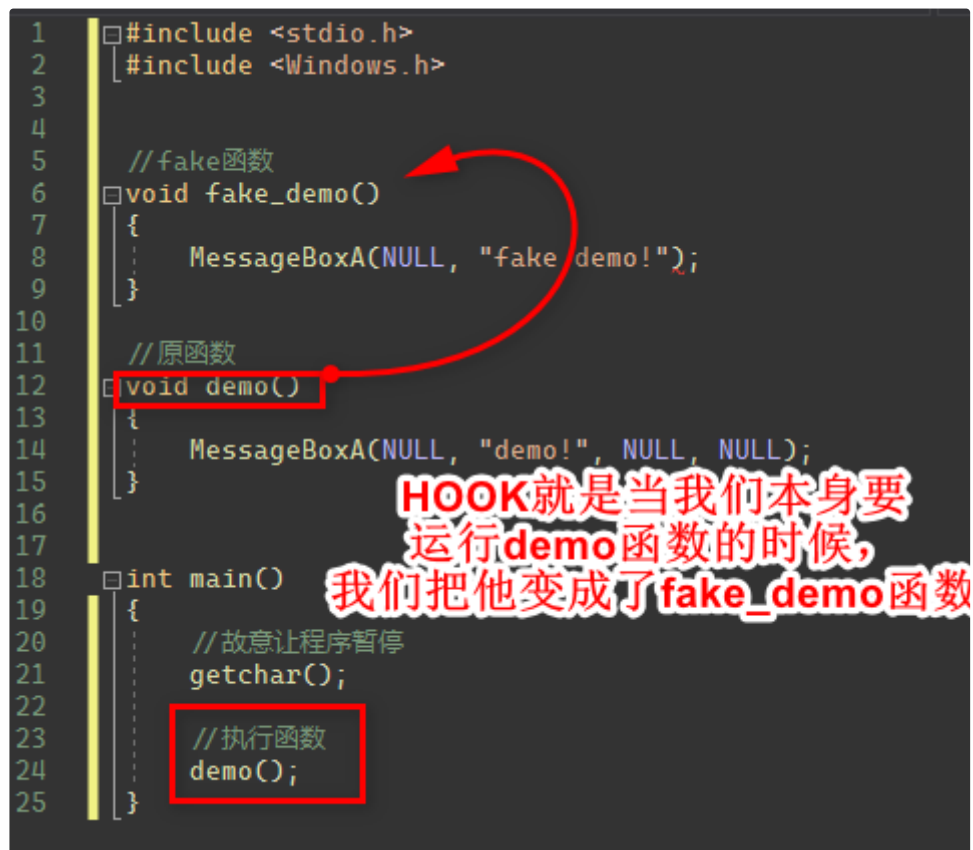
inline hook是一种通过修改机器码的方式来实现hook的技术。

在没有学汇编知识前，我们可能对Hook这种技术感到很深奥，觉得这简直是一种黑魔法，凭什么他就能把正常函数替换成我们自己的假函数。

5.2. Inline Hook 原理

当我们学习了王爽老师的汇编知识后，才明白原来在底层，CPU是根据ip寄存器来控制我们要执行的指令处的。

而学习了《王爽汇编第9章转移指令的原理》后，了解了可以通过jmp指令来修改ip寄存器，目前我们已经学习了短转移的使用，这已经足够用来学HOOK了。



打开OD载入程序，然后找到执行demo函数的汇编代码处，再利用我们今天刚学的短转移知识进行机器码的修改。

首先经过单步调试后，我找到了执行demo函数处的位置，我们先记下他下条指令的地址 0x00481885。

00481850	<hookd	55	push ebp	hookdemo.cpp:19
00481851		8BEC	mov ebp,esp	
00481853		81EC C0000000	sub esp,0xC0	
00481859		53	push ebx	
0048185A		56	push esi	
0048185B		57	push edi	
0048185C	<hookd	8BFD	mov edi,ebp	__\$EncStackInitStart
0048185E		33C9	xor ecx,ecx	
00481860		B8 CCCCCCCC	mov eax,0xC0000000	
00481865		F3:AB	rep stosd	
00481867	<hookd	B9 15C04800	mov ecx,<hookdemo._6EDC718C_h	hookdemo.cpp:15732480
0048186C		E8 A5FAFFFF	call hookdemo.481316	
00481871		8BF4	mov esi,esp	hookdemo.cpp:21
00481873		FF15 7CB14800	call dword ptr ds:[&getchar>	
00481879		3BF4	cmp esi,esp	
0048187A		E8 BAE9FFFF	call hookdemo.48123A	
00481880		E8 E1FAFFFF	call hookdemo.481366	执行demo()函数处
00481885		33C0	xor eax,eax	hookdemo.cpp:25
00481887		5F	pop edi	
00481888		5E	pop esi	
00481889		5B	pop ebx	
0048188A		81C4 C0000000	add esp,0xC0	
00481890		3BEC	cmp ebp,esp	
00481892		E8 A3F9FFFF	call hookdemo.48123A	
00481897		8BE5	mov esp,ebp	

接着我们需要找到 标号 处，也就是 fake_demo 函数处，也记下地址 0x004817D0

004817D0 <hookd	55	push ebp	fake demo函数处
004817D1	8BEC	mov ebp,esp	
004817D3	81EC C0000000	sub esp,0xC0	
004817D9	53	push ebx	
004817DA	56	push esi	
004817DB	57	push edi	
004817DC <hookd	8BFD	mov edi,ebp	__\$EncStackInitStart
004817DE	33C9	xor ecx,ecx	
004817E0	B8 CCCCCCCC	mov eax,0xC0000000	
004817E5	F3:AB	rep stosd	
004817E7 <hookd	B9 15C04800	mov ecx,<hookdemo._6EDC718C_hookdemo@cpp>	hookdemo.cpp:15732480
004817EC	E8 25FBFFFF	call hookdemo.481316	
004817F1	8BF4	mov esi,esp	hookdemo.cpp:8
004817F3	6A 00	push 0x0	
004817F5	6A 00	push 0x0	
004817F7	68 307B4800	push hookdemo.487B30	487B30:"fake demo!"
004817FC	6A 00	push 0x0	
004817FE	FF15 98B04800	call dword ptr ds:[<&MessageB	
00481804	3BF4	cmp esi,esp	
00481806	E8 2FFAFFFF	call hookdemo.48123A	
0048180B	5F	pop edi	hookdemo.cpp:9
0048180C	5E	pop esi	
0048180D	5B	pop ebx	
0048180E	81C4 C0000000	add esp,0xC0	
00481814	3BEC	cmp ebp,esp	

好了，最后的步骤就是改机器码，根据公式计算出位移：**-181(0xFFFFF4B)**

十六进制

字符串

复制数据

鼠标右键：二进制->编辑

ASCII

UNICODE

UTF-16

十六进制(D):

E9 4B FF FF FF

公式 (位移)=(标号)-(jmp下一句地址)=-181

转换成对应的十六进制: 0xFFFFF4B

>>> hex(-181&0xFFFFFFFF)
'0xfffff4b'

确定(O)

取消(C)

00481800

00481801

00481802

00481803

00481804

00481805

00481806

00481807

00481808

00481809

0048180A

0048180B

0048180C

0048180D

0048180E

0048180F

00481810

00481811

00481812

00481813

00481814

00481815

00481816

00481817

00481818

00481819

0048181A

0048181B

0048181C

0048181D

0048181E

0048181F

00481820

00481821

00481822

00481823

00481824

00481825

00481826

00481827

00481828

00481829

0048182A

0048182B

0048182C

0048182D

0048182E

0048182F

00481830

00481831

00481832

00481833

00481834

00481835

00481836

00481837

00481838

00481839

0048183A

0048183B

0048183C

0048183D

0048183E

0048183F

00481840

00481841

00481842

00481843

00481844

00481845

00481846

00481847

00481848

00481849

0048184A

0048184B

0048184C

0048184D

0048184E

0048184F

00481850

00481851

00481852

00481853

00481854

00481855

00481856

00481857

00481858

00481859

0048185A

0048185B

0048185C

0048185D

0048185E

0048185F

00481860

00481861

00481862

00481863

00481864

00481865

00481866

00481867

00481868

00481869

0048186A

0048186B

0048186C

0048186D

0048186E

0048186F

00481870

00481871

00481872

00481873

00481874

00481875

00481876

00481877

00481878

00481879

0048187A

0048187B

0048187C

0048187D

0048187E

0048187F

00481880

00481881

00481882

00481883

00481884

00481885

00481886

00481887

00481888

00481889

0048184E	CC	int3	
0048184F	CC	int3	
00481850 <hookdemo._main>	55	push ebp	hookdemo.cpp:19
00481851	8BEC	mov ebp,esp	
00481853	81EC C0000000	sub esp,0xC0	
00481859	53	push ebx	
0048185A	56	push esi	
0048185B	57	push edi	
0048185C <hookdemo.__\$EncStackI	8BFD	mov edi,ebp	__\$EncStackInitSt
0048185E	33C9	xor ecx,ecx	
00481860	B8 CCCCCCCC	mov eax,0xC0000000	
00481865	F3:AB	rep stosd	
00481867 <hookdemo.__\$EncStackI	B9 15C04800	mov ecx,<hookdemo._6EDC718C_hookdemo@cpp>	hookdemo.cpp:1573
0048186C	E8 A5FAFFFF	call hookdemo.481316	
00481871	8BF4	mov esi,esp	hookdemo.cpp:21
00481873	FF15 7CB14800	call dword ptr ds:[<&getchar>]	
00481879	3BF4	cmp esi,esp	
0048187B	E8 BAF9FFFF	call hookdemo.48123A	
00481880	E9 4BFFFFF	jmp <hookdemo.void __cdecl fake_demo(void)>	执行demo()函数处
00481885	33C0	xor eax,eax	hookdemo.cpp:25
00481887	5F	pop edi	
00481888	5E	pop esi	
00481889	5B	pop ebx	

从这里可以看到，会jmp到我们的fake_demo函数

最后来执行一遍：



5.3. Hook代码开发

在了解了Hook原理后，甚至我们用手动方式实现了HOOK后，就有了思路来代码开发了。

在此之前我们先来认识两个函数 `WriteProcessMemory`、`VirtualProtect`

- `WriteProcessMemory`函数可以对进程的内存进行写入，这样我们就可以修改，写入机器码了。

C++

此函数能写入某一进程的内存区域（直接写入会出Access Violation错误，故需此函数）。

VC++声明

```
1  BOOL WriteProcessMemory(  
2  HANDLE hProcess,  
3  LPVOID lpBaseAddress,  
4  LPVOID lpBuffer,  
5  DWORD nSize,  
6  LPDWORD lpNumberOfBytesWritten  
7  );
```

- `VirtualProtect`函数可以更改虚拟内存中的访问保护，在Win32中代码段具有写保护，所以我们无法直接写入，可以利用此函数修改保护后再调用`WriteProcessMemory`修改和写入机器码。

VirtualProtect

11/18/2015 • 3 minutes to read

A version of this page is also available for

[Windows Embedded CE 6.0 R3](#)

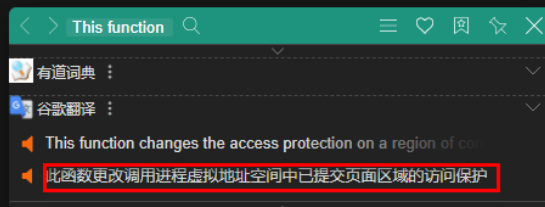
4/8/2010

This function changes the access protection on a region of committed pages in the virtual address space of the calling process.

Syntax

```
BOOL VirtualProtect(  
    LPVOID lpAddress,  
    DWORD dwSize,  
    DWORD flNewProtect,  
    PDWORD lpflOldProtect  
);
```

Copy



```

1  /*****
2  *功能：内联HOOK 函数，可以劫持函数的原本流程
3  *
4  *函数名：InlineHook
5  *参数：(原函数地址)、(fake函数地址)
6  *by：《王爽汇编笔记》
7  *****/
8  void InlineHook(DWORD dwHookAddr, LPVOID pFunAddr)
9  {
10     BYTE jmpCode[5] = {0xE9};
11     //计算偏移
12     *(DWORD*)&jmpCode[1] = (DWORD)pFunAddr - dwHookAddr - 5;
13     // 保存以前的属性用于还原
14     DWORD OldProtext = 0;
15     DWORD dwWritten;
16     // 因为要往代码段写入数据，又因为代码段是不可写的，所以需要修改属性
17     VirtualProtect((LPVOID)dwHookAddr, 5, PAGE_EXECUTE_READWRITE, &OldProtext);
18     WriteProcessMemory(GetCurrentProcess(), (FARPROC)dwHookAddr, jmpCode, 5,
19     &dwWritten);
20     VirtualProtect((LPVOID)dwHookAddr, 5, OldProtext, &OldProtext);
21 }
22
23 int main()
24 {
25     //故意让程序暂停
26     getchar();
27     //执行函数
28     InlineHook((DWORD)&demo, &fake_demo);
29     demo();
30 }

```

6. 转移的目的地址在指令中的jmp指令

jmp far ptr 标号 (远转移、段间转移)

- (CS) = 标号所在段的段地址;
- (IP) = 标号所在段中的偏移地址。
- far ptr 指明了指令用标号的 **段地址** 和 **偏移地址** 修改 **CS和IP**。

```

1  assume cs:code
2
3  code segment
4  main:
5      mov ax,0
6      mov bx,0
7      jmp far ptr s ;s被翻译成转移目的地址076C:0101
8      db 256 dup(0) ;填充256字节
9  s:  add ax,1
10     inc ax
11 code ends
12 end main

```

DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Program: DEBUG

```

D:\>if exist CHANGEAS.OBJ link CHANGEAS.OBJ; >X:\LINK.LOG

Microsoft (R) Segmented Executable Linker Version 5.31.009
Copyright (C) Microsoft Corp 1984-1992. All rights reserved.

LINK : warning L4021: no stack segment

D:\>if exist CHANGEAS.exe c:\masm\debug CHANGEAS.exe
-U
076C:0000 B80000 MOV AX,0000
076C:0003 BB0000 MOV BX,0000
076C:0006 EA0B016C07 JMP 076C:010B
076C:000B 0000 ADD [BX+SI],AL
076C:000D 0000 ADD [BX+SI],AL

```

在 Win32 中的 远转移 可以跳到其他DLL的空间，其中关键的机器码是 0xFF25 + 目标地址

004010E0	55	push ebp	
004010E1	8BEC	mov ebp,esp	
004010E3 <JMP.	FF25 040FE174	jmp dword ptr ds:[&CreateFileA]	JMP.&CreateFileA
004010E9	0000	add byte ptr ds:[eax],al	
004010EB	00C0 0x74e10fd4	add al,al	
004010ED	83E1 03	and ecx,0x3	
004010F0	6A 00	push 0x0	
004010F2	68 80000000	push 0x80	
004010F7	6A 03	push 0x3	
004010F9	6A 00	push 0x0	
004010FB	51	push ecx	
004010FC	50	push eax	
004010FD	FF75 08	push dword ptr ss:[ebp+0x8]	
00401100	FF15 04204000	call dword ptr ds:[&CreateFileA]	
00401106	C9	leave	

地址	值	ASCII	注释
74E10FD4	76D0A520	×Ðv	kernelbase.CreateFileA
74E10FD8	76CF3DE0	à=İv	kernelbase.CreateDirectoryW
74E10FDC	76CD0690	.Öİv	kernelbase.CreateDirectoryA
74E10FE0	76D06F40	@oÐv	kernelbase.CompareFileTime
74E10FE4	76D93160	`1İv	kernelbase.GetFinalPathNameByHandleA
74E10FE8	76CDA660	`İİv	kernelbase.GetFinalPathNameByHandleW
74E10FEC	76CD4200	.Bİv	kernelbase.GetFullPathNameA
74E10FF0	76CF3400	.4İv	kernelbase.GetFullPathNameW
74E10FF4	76D8CD40	@İİv	kernelbase.GetLogicalDriveStringsW
74E10FF8	76CDBFB0	° İİv	kernelbase.GetTempFileNameW
74E10FFC	76D08490	..Ðv	kernelbase.GetVolumeInformationByHandleW
74E11000	76CD5970	pYİv	kernelbase.GetVolumeInformationW
74E11004	76CF55C0	ÀUİv	kernelbase.GetVolumePathNameW
74E11008	76CDCB70	pEİv	kernelbase.LocalFileTimeToFileTime
74E1100C	76CD9800	..İv	kernelbase.LockFile
74E11010	76D0F9E0	àùÐv	kernelbase.LockFileEx
74E11014	76D0B600	..İv	kernelbase.QueryDosDeviceW
74E11018	76CF0D80	..İv	kernelbase.ReadFile
74E1101C	76D932A0	2İİv	kernelbase.ReadFileEx

76D0A520 <kernelbase.CreateFileA>	8BFF	mov edi,edi	CreateFileA
76D0A522	55	push ebp	
76D0A523	8BEC	mov ebp,esp	
76D0A525	51	push ecx	
76D0A526	51	push ecx	
76D0A527	8B55 08	mov edx,dword ptr ss:[ebp+0x8]	
76D0A52A	8D4D F8	lea ecx,dword ptr ss:[ebp-0x8]	
76D0A52D	E8 646DFEFF	call kernelbase.76CF1296	
76D0A532	85C0	test eax,eax	
76D0A534	74 2E	je kernelbase.76D0A564	
76D0A536	56	push esi	
76D0A537	FF75 20	push dword ptr ss:[ebp+0x20]	
76D0A53A	FF75 1C	push dword ptr ss:[ebp+0x1C]	
76D0A53D	FF75 18	push dword ptr ss:[ebp+0x18]	
76D0A540	FF75 14	push dword ptr ss:[ebp+0x14]	
76D0A543	FF75 10	push dword ptr ss:[ebp+0x10]	
76D0A546	FF75 0C	push dword ptr ss:[ebp+0xC]	
76D0A549	FF75 FC	push dword ptr ss:[ebp-0x4]	

7. 转移地址在寄存器或内存中的jmp指令

jmp 16位寄存器 功能: $IP = (16\text{位寄存器})$ [段内转移]

转移地址在内存中的jmp指令有两周格式:

- `jmp word ptr [...]` (段内转移)

```

1  mov ax,0123h
2  mov ds:[0],ax
3  jmp word ptr ds:[0]
4  ;ip = 0123h

```

- `jmp dword ptr [...]` (段间转移)

功能: 从内存单元地址处开始存放着两个字, **高地址** 出的字是转移目的 **段地址**, **低地址** 处是转移目的 **偏移地址**。

1: (CS) = (内存单元地址+2)

2: (IP) = (内存单元地址)

```

1  mov ax,010Bh
2  mov ds:[0],ax
3  mov ax,076ch
4  mov ds:[2],ax
5  jmp dword ptr ds:[0]
6  ;(CS) = 076C
7  ;(IP) = 010B
8  ;CS:IP = 076C:010B

```

8. jcxz指令和loop指令

8.1. jcxz指令

jcxz指令为有条件转移指令，所有的有条件转移指令都是短转移，在对应的机器码中包含转移的 **位移**，**而不是目的地址**。对IP的修改范围都为-128~127(0xff)。

指令格式：jcxz **标号** (如果 **cx=0**，则转移到标号处执行。)

当(cx) = 0时，(IP) = (IP) + 8位移

- 8位移 = "标号" 处的地址 - jcxz指令后的第一个字节的地址；
- 8位移的范围为-128~127，用补码表示；
- 8位移由编译程序在编译时算出。

当(cx) != 0时，程序向下执行什么都不做！

8.2. loop指令

loop指令为循环指令，所有的循环指令都是短转移，在对应的机器码中包含转移的 **位移**，而不是目的地址。

对IP的修改范围都为-128~127(0xFF)。

指令格式：loop **标号** ((cx) = (cx) - 1, 如果 **(cx) ≠ 0**，转移到标号处执行)。

(cx) = (cx) - 1; 如果 (cx) != 0, (IP) = (IP) + 8位移。

- 8位移 = ""标号""处的地址 - loop指令后的第一个字节的地址；
- 8位移的范围为-128~127，用补码表示；
- 8位移由编译程序在编译时算出。

如果 (cx) = 0，什么也不做（程序向下执行）。

9. 几种跳转指令和对应的机器码

注：(Win32 x86架构下)

机器码	指令	解释
0xE8	CALL	后面的四个字节是地址（近转移）
0xE9	JMP	后面的四个字节是偏移（近转移）
0xEB	JMP	后面的二个字节是偏移（近转移 8086）
0xFF15	CALL	后面的四个字节是存放地址的地址（远转移）
0xFF25	JMP	后面的四个字节是存放地址的地址（远转移）
0x68	PUSH	后面的四个字节入栈
0x6A	PUSH	后面的一个字节入栈

参考文献：

<https://www.cnblogs.com/Archimedes/p/14823218.html> inline hook原理和实现

<https://blog.csdn.net/wzsy/article/details/17163589> 几种跳转指令和对应的机器码

https://blog.csdn.net/qq_39654127/article/details/88698911 王爽《汇编语言》笔记（详细）

《王爽汇编第四版 第9章》