

# 第三讲



- 熟练使用调试器
- IDA
- 结合调试器使用反汇编器

# 黑客反向工程---C规范的知识

■ strcmp传递两个参数：

```
push    offset s_Mygoodpasswor ; 参考密码  
lea     ecx, [ebp+var_68]; 用户输入密码区  
push    ecx  
call    strcmp
```

var\_68: 可能对应源代码中的buff.

■ C 规范：从右到左的顺序将参数压入堆栈。

■ 恢复的结果为：

```
strcmp(var_68, "myGOODpassword\n")
```

# 黑客反向工程—CALL指令后清栈

- 1 从堆栈中删除参数不由函数自身完成，而是由调用程序完成，这样能创建数目可变的参数。

```
call strcmp                ret 8  
add esp, +08
```

- 2 常用清除堆栈指令

```
add esp, xxx
```

32位:  $n\_args = XXX/4$

16位:  $n\_args = XXX/2$

```
pop reg  
sub esp, -xxx
```

- 3 由call后的add esp, 8 指令知该函数个数为2。

# 黑客反向工程---分析修改的程序

```
text:00401063  add    esp, 8
text:00401066  test   eax, eax
text:00401068  jz     short loc_401079
text:0040106A  push   offset_WrongPassword
text:0040106F  call   printf
```

.....

loc\_401079:

**eax:** 存放strcmp函数返回值。检查eax是否为零,若0,提示密码OK; 否则转错误处理程序。

jz(74)--» jnz(75)

test eax,eax(85C0)→xor eax,eax(31C0)等



# 外科手术---改程序

■ 直接修改代码极其危险,增加字节,会造成指令中偏移量和转向地址不再指向原来的地址。

■ 解决方法: 使用直接编辑二进制工具 hiew32等。

hiew32            list1\_p7.exe

目标:            通过阅读理解, 找出需要修改的代码行, 寻找 JZ 机器代码

■ 定位

1068h-----将 7 4    改为    75

# 修改可执行文件的结果

修改前:

```
00401066:85 C0    test    eax,    eax
00401068:74 0F    jz      00401079
```

把原来程序中的 JZ(JE)74 改为 JNZ(JNE)75

修改后:

```
00401066:85 C0    test    eax,    eax
00401068:75 0F    jnz     00401079
```

则程序把正确的密码当成错误的,反过来把所有错误的密码都当成是正确的(有一种情况不能通过)。

# 直接修改代码极其危险

修改或删除汇编代码造成的限制是：

- 将保护机制的“多余部分”丢弃以后，并不能把指令移开或者把它们“挤压”到一起，这样做会使所有其他指令的偏移量发生变化。而指针值与跳转地址却会因为保持不变而指向错误的位置。
- 用 NOP (0x90) 指令清除“空闲部分”相对简单。
- 由于使用汇编指令进行修改，因此需要进行反汇编。

# 直接修改代码极其危险

## ■ 能否编译经过反汇编的汇编源文件？

如果汇编工具不能识别传递给函数的指针，那么这些指针值就不能得到正确的修正，因此会发生程序不能正确地运行(错误)。

## ■ 反汇编器也不能将指针值(地址)同常量区分开。

```
str(char *s1, int s2)
```

反汇编后有可能变成：

```
str(0x01000, [bp+06])
```

则 汇编器不能将0x1000翻译成某个变量存储单元的地址或者是分配一个绝对地址(均会错误)。



# 另一种修改方式

修改代码的不同方式:

```
00401066:85 C0  test  eax,  eax
00401068:74 0F  je    00401079
```

把原来程序中的 test(85) 改为 xor (31)

```
00401066:31 C0  xor   eax,  eax
00401068:74 0F  je    00401079
```

则程序把正确的和错误的密码都当成正确的。

可以利用 **fc** file1 file2 比较修改前后的不同。

# 熟练使用调试器

- 调试器：一步一步地执行代码(跟踪)。  
不是理解程序的最好方式。

调试器的功能：

- 跟踪写入、读取、或者执行地址（断点）。
- 跟踪对输入或者输出端口的写入或者读取调用。
- 跟踪动态连接库的加载和特定函数的调用。
- 跟踪程序或者硬件的中断。
- 跟踪发送到Windows 消息以及内存中的上下文搜索。

# 方法0: 使用调试器寻找保密指令

Idag list1\_p7.exe

- .text 00401000 00423000 R CODE
- .rdata 00423000 00425000 R DATA
- .data 00425000 0042A5E4 R W DATA
- .idata 0042B14C 0042B220 R W DATA

# rdata 段内容

- rdata: 0042301C ..... Password OK db 'Password  
OK',0Ah,0
- rdata:00423029 .....
- rdata:0042302C ..... db 'Wrong password',0Ah,0
- .....
- rdata:00423040...db'myGOODpassword',0Ah,0  
(此处设置断点)
- rdata:00423054 .....db 'Enter password: ',0Ah,0



# 判断密码是否正确的程序

按F9后并输入密码得到结果如下(跳到strcmp函数体中):

text:004010F0	mov	eax, [edx]
text:004010F2	cmp	al, [ecx]
text:004010F4	jnz	short loc_401124
text:004010F6	or	al, al
text:004010F8	jz	short loc_401120
text:004010FA	cmp	ah, [ecx+1]
text:004010FD	jnz	short loc_401124
text:004010FF	or	ah, ah

.....

ecx可能就是刚刚设置断点的地址

# 判断密码是否正确的程序分析

推测:

- 此处只设置了一个断点, 在参照密码处, 由于Intel 处理器特定的结构特性, 断点在指令被执行之后激活, text:IP 指向下一条可执行指令, 即此处的

**text:004010F4          jnz    short loc\_401124**

- 假定ecx 包含一个指向参照密码串的指针, 因为它引起了执行过程的中断. 则 edx 必是指向用户输入的字符串指针。查看ecx 和 edx 可以验证结果。

# 判断密码是否正确的程序分析和修改

上页显示的是 strcmp 函数体代码。

## ■ 修改注意事项：

若用 JZ 代替 JNZ 会影响其他进程对 strcmp调用，会使功能完整的其它进程或程序运行失败。

## ■ 解决方法：退出strcmp函数转到调用该函数的代码处，修

改用于确定密码是否正确的 IF 语句。

## ■ 使用 F8(step over)完成。

# 判断密码是否正确的程序分析和修改

text:0040105D	push ecx
text:0040105E	call strcmp
text:00401063	add esp, 8
text:00401066	test(xor) eax, eax
text:00401068	jz(jnz) short loc_401079
text:0040106A	push offset ??_C@_0BA@ MPGF@Wrong?5password?6?\$ AA@ ; "Wrong password\n"
text:0040106F	call printf
text:00401074	add esp, 4
text:00401077	jmp short loc_40107B



# 判断密码是否正确的程序分析和修改

## ■ 修改1:

00401066	test(85)	eax, eax
修改为:	xor(31)	eax, eax

## ■ 修改2:

00401068	jz (74)	00401079
修改为:	jnz(75)	00401079

## 方法1: 直接在内存中搜索用户输入的密码

■ 由于各种复杂的原因, 密码不可能会放在很容易找到的地方, 同时程序中也有非常多的字符串都很像密码串。

■ 想法:

搜索原始密码很难, 搜索用户输入的密码应该简单。

## 方法1: 直接在内存中搜索用户输入的密码

### ■ 工作过程:

idag 运行 程序。

输入 “ KPNC Kaspersky”

转idag 调试方式， 搜索KPNC Kaspersky 字符串。可找出多个位置，分析选择可能性最大的一个。

在相应位置处004295E0中设置断点

然后重新启动idag 运行。

断点设置在检测调用语句之后的指令上。

退出匹配的进程/修正JMP指令以及.....

```
for( ; ; )
{
    printf("Enter password: ");
    fgets(&buff[0], PASSWORD_SIZE, stdin);
    if (strcmp( &buff[0], PASSWORD))
        // "申斥"密码不匹配
        printf("Wrong password\n");
    else break;
    if (++count>3) return -1;
}
```



```
#define PASSWORD_SIZE 100
#define PASSWORD "password"
int main()
{
    char buff[PASSWORD_SIZE], count=0;
    for (; ; )
    {
        printf("Enter password: ");
        fgets(&buff[0], PASSWORD_SIZE, stdin);
        buff[strlen(buff)-1] = '\0'; //把回车换行符也读进来了
        if (strcmp(&buff[0], PASSWORD))
            // "申斥"密码不匹配
            printf("Wrong password\n");
        else break;
        if (++count > 3) return -1; } }
```

### ■ 分析:

用户提供的密码被放在 buff 缓冲区 并与参照密码比较,如果不匹配,那么需要从用户那里再次请求提供密码.

### ■ 第二次输入前不清除 buff.

依次类推,到达接受 Wrong password 执行分支,使用 ida 调试器,进行搜索,就会找到.

# 实验和验证

利用ida的 view --> open subviews 的 strings 功能输入  
密码: abcdef

- 000000AA2C9AF750 CCCCCCCCCCCCCCCCCC
- 000000AA2C9AF758 CCCCCCCCCCCCCCCCCC
- 000000AA2C9AF760 000A666564636261 //设  
断点
- 000000AA2C9AF768 CCCCCCCCCCCCCCCCCC
- 000000AA2C9AF770 CCCCCCCCCCCCCCCCCC
- 000000AA2C9AF778 CCCCCCCCCCCCCCCCCC

```
.text:00007FF6CF695B08  
.text:00007FF6CF695B08 loc_7FF6CF695B08:  
.text:00007FF6CF695B08 mov     rax, [rbp+1A0h+var_28]  
.text:00007FF6CF695B0F mov     [rbp+rax+1A0h+Buffer], 0  
.text:00007FF6CF695B14 mov     eax, 1  
.text:00007FF6CF695B19 imul    rax, 0  
.text:00007FF6CF695B1D lea     rax, [rbp+rax+1A0h+Buffer]  
.text:00007FF6CF695B22 lea     rdx, Str2      ; "password"  
.text:00007FF6CF695B29 mov     rcx, rax      ; Str1  
.text:00007FF6CF695B2C call    j_strcmp_0 //假设没有用  
strcmp函数, 而是将strcmp 相应代码插入此处增加难度, 见下面  
.text:00007FF6CF695B31 test    eax, eax  
.text:00007FF6CF695B33 jz     short loc_7FF6CF695B43
```



```
int strcmp(char *p1, char *p2)
{
    do {
        if (*p1=='\0') return (*p1-*p2);
        p1++; p2++;
    } while (*p1==*p2);

    return (*p1-*p2);
}
```

指向用户输入密码的指针被放置在寄存器 EAX 中。

```
004013E3: 8A 16          mov     dl, byte ptr [esi]
004013E5: 8A 1E          mov     bl, byte ptr [esi]
004013E7: 8A CA          mov     cl, dl
004013E9: 3A D3          cmp     dl, bl
```

对第一个字符进行比较。

```
004013EB: 75 1E          jne     0040140B <--- (3) ---> (1)
```

如果第一个字符不匹配，就进行跳转。进一步的检查是没有意义的。

```
004013ED: 84 C9          test    cl, cl
```

第一个字符是否等于零？

```
004013EF: 74 16          je      00401407 ----> (2)
```

如果等于零，则我们就到达了字符串的结尾，并且密码相同。

```
004013F1: 8A 50 01       mov     dl, byte ptr [eax+1]
004013F4: 8A 5E 01       mov     bl, byte ptr [esi+1]
004013F7: 8A CA          mov     cl, dl
004013F9: 3A D3          cmp     dl, bl
```

检查下一对字符。

```
004013FB: 75 0E          jne     0040140B ----> (1)
```

如果它们不相等，就停止检查。

```
004013FD: 83 C0 02       add     eax, 2
00401400: 83 C6 02       add     esi, 2
```

```
00401403: 84 C9          test        cl, cl
```

是否到达了该行的结尾？

```
00401405: 75 DC          jne         004013E3 ----> (3)
```

不，没有。继续匹配。

```
00401407: 33 C0          xor         eax, eax <---- (2)
```

```
00401409: EB 05          jmp         00401410 ----> (4)
```

寄存器 EAX 被清除（如果成功，则 strcmp 返回零）并退出。

```
0040140B: 1B C0          sbb         eax, eax <---- (3)
```

```
0040140D: 83 D8 FF          sbb         eax, 0FFFFFFFFh
```

当密码不匹配时，执行这个分支。EAX 被置零值（猜一下这是为什么？）。

```
00401410: 85 C0          test        eax, eax <---- (4)
```

如果 EAX 等于零，进行一种检查。

```
00401412: 6A 00          push        0
```

```
00401414: 6A 00          push        0
```

某些数值被放入堆栈。

```
00401416: 74 38          je          00401450 <<<< ----> (5)
```

跳转到某处。

```
00401418: 68 2C 30 40 00  push        40302Ch
```

```
0040302C: 57 72 6F 6E 67 20 70 61 73 73 77 6F 72 64 00 .Wrong password
```

哈！“Wrong password。”（紧跟着的代码不是我们感兴趣的，只是显示错误信息。）

既然已经理解了算法，就能破解它（例如，用一个短的无条件跳转，比如 0xEB，来抵

## 方法2 在密码输入函数上设置断点

```
void CCrackme_01Dlg::OnOk()
{
    char buff[PASSWORD_SIZE];
    CEdit m_password;

    m_password.GetWindowTextA(&buff[0],
    PASSWORD_SIZE);
    if (strcmp(&buff[0], PASSWORD))
    {
        printf("Wrong Password");
        m_password.SetSel(0,-1,0);
        return;
    }
    else
    { printf("Password OK"); }
    CDialog::OnOK();
}
```



## 方法2 在密码输入函数上设置断点解释

- 使用 ida 运行上面程序并输入 kpnc Kaspersky++后跟踪程序(反汇编器能识别系统函数)

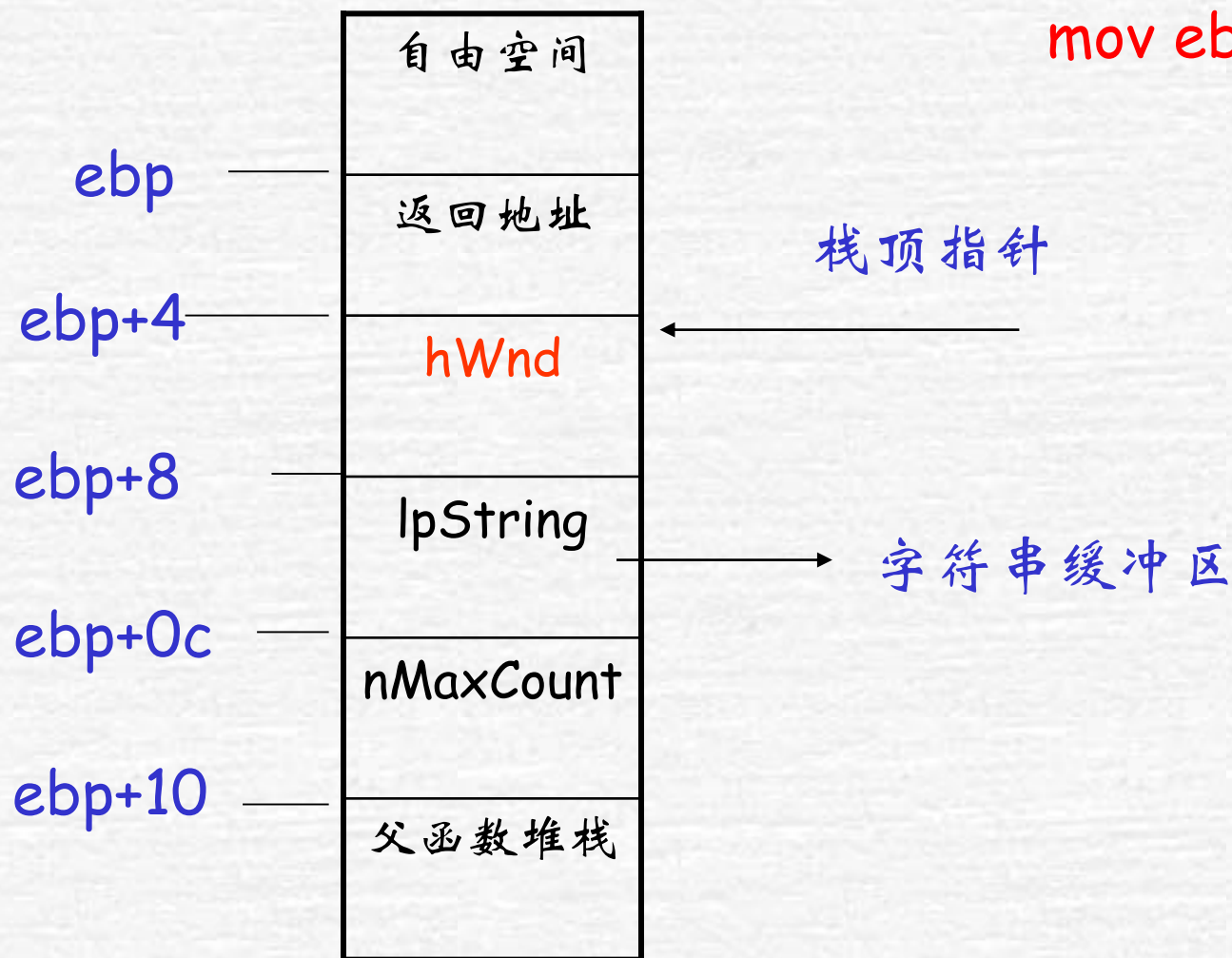
注意调用 GetWindowText

text:77E1A4E2	push ebp
text:77E1A4E3	mov ebp, esp
text:77E1A4E5	push ff
text:77E1A4E7	push 77E1A570
text:77E1A4EC	.....

- 注意GetWindowText 的参数压栈顺序,  
显示 ebp+8 内容:

```
GetWindowText (  
    HWND    hwnd;          /* 文本窗口或者控件句柄 */  
    LPTSTR  lpString;      /* 文本缓冲区地址 */  
    int      nMaxCount;    /* 欲拷贝的最大字符数 */  
);
```

Windows api 函数按stdcall规范从右到左顺序压参数，  
在32 windows中，此函数的所有参数和返回地址都按  
4个字节处理。

`mov ebp,esp`

0023:0012F9FC 1C FA 12 00 3B 5A E1 77-EC 4D E1 77 06 02 05 00.....;Z.w.M.w....  
0023:0012FA0C 01 01 00 00 10 00 00 00-01 00 2A C0 10 A8 48 00.....\*...H.  
0023:0012FA1C 10 9B 13 00 0A 02 04 00-E8 3E 2F 00 00 00 00 00.....>/.....  
0023:0012FA2C 01 02 04 00 83 63 E1 77-08 DE 48 00 0A 02 04 00.....c.w..H.....

0023:0012F9FC 4B 50 4E 43 20 4B 61 73-70 65 72 73 6B 79 2B 2B **KPNC Kaspersky++**  
0023:0012FA0C 00 01 00 00 0D 00 00 00-01 00 1C 80 10 A8 48 00 .....H.  
0023:0012FA1C 10 9B 13 00 0A 02 04 00-E8 3E 2F 00 00 00 00 00 .....>/.....  
0023:0012FA2C 01 02 04 00 83 63 E1 77-08 DE 48 00 0A 02 04 00 .....c.w..H.....



■ 推出函数接受输入的函数:

(ebp+8) = 0012F9FC

D 0023:0012F9FC:

0023:0012F9FC    kpnc Kaspersky++

.....

■ 在此处设置断点, 再从头执行, 就可以到你要去的程序代码段.

```

0023:0012F9FC 4B 50 4E 43 20 4B 61 73-70 65 72 73 6B 79 2B 2B KPNC Kaspersky++
0023:0012FA0C 00 01 00 00 0D 00 00 00-01 00 1C 80 10 A8 48 00 .....H.
0023:0012FA1C 10 9B 13 00 0A 02 04 00-E8 3E 2F 00 00 00 00 .....>/.....
0023:0012FA2C 01 02 04 00 83 63 E1 77-08 DE 48 00 0A 02 04 00 .....c.w..H.....

```

这就是我们需要的缓冲区。设置一个断点，并等待调试器窗口的出现。看！（你认出了这个比较过程吗？）经过第一次尝试，我们就达到了目的：

```

001B:004013E3 8A10          mov     dl, [eax]
001B:004013E5 8A1E          mov     bl, [esi]
001B:004013E7 8ACA          mov     cl, dl
001B:004013E9 3AD3          cmp     dl, bl
001B:004013EB 751E          jnz     0040140B
001B:004013ED 84C9          test    cl, cl
001B:004013EF 7416          jz      00401407
001B:004013F1 8A5001        mov     dl, [eax+01]

```

# 方法3 针对消息设置代码段

■ 利用消息获取编辑窗口文本.代替 GetWindowText()

开发人员可以利用

SendMessageA(hWnd, WM\_GETTEXT,(LPARAM)&buff[0])

代替 GetWindowTextA(GetDlgItemTextA)

在 SendMessageA 函数上设置断点，常用的是在

■ WM\_GETTEXT 消息处设置断点

# 方法3 针对消息设置代码段

## ■ 利用消息断点

在处理字符串方面可以利用消息断点WM\_GETTEXT和WM\_COMMAND。前者用来读取某个控件中的文本，比如拷贝编辑窗口中的序列号到程序提供的一个缓冲区里；后者则是用来通知某个控件的父窗口的，比如当输入序列号之后点击OK按钮，则该按钮的父窗口将收到一个WM\_COMMAND消息，以表明该按钮被点击。

■ BP xxxx WM\_GETTEXT (拦截序列号)

■ BP xxxx WM\_COMMAND (拦截OK按钮)



# 消息原型

Handle	Class	WinPorc	TID
Module			
050140	Dialog	6c291b81	2Dc
05013E	Button	6c291b81	2Dc
05013C	Edit	6c291b81	2Dc
05013A	Static	6c291b81	2Dc

可以很快地根据窗口过程定位编辑窗口，0x6c291b81，  
可以在此设置断点，也可以在后面设

# 消息原型

LRESULT CALLBACK WindowProc (

- hwnd: 指向窗口的句柄。
  - uMsg: 指定消息类型。
  - wParam: 指定其余的、消息特定的信息。该参数的内容与uMsg参数值有关。
  - lParam: 指定其余的、消息特定的信息。
- );

返回值: 返回值就是消息处理结果,它与发送的消息有关。

容易算出: 当该函数被调用时, 参数uMsg 相对于ESP值的偏移量为8个字节, 如果该位置的值等于  
**WM\_GETTEXT** 就可以暂停运行

# 消息原型

返回类型是LRESULT，是由Windows所定义的数据类型，通常相当于 long 型。

4个参数的传递提供了引起函数被调用的具体消息的情况。  
每个参数的意义如下：

HWND hWnd：事件引起消息发生的那个窗口。

UINT message：消息ID，它是32位值，指明了消息类型。

LPARAM wParam：32位值，包含附加信息，决定于消息的种类。例如键盘的哪个键代码。

LPARAM lParam：32位值，同上。

例前16位 = 重复数

接着8位：扫描码（决定于厂家）

# 消息原型

■ 第24位：为1时表示扩展键。

第25到28位：保留区

第29位为1时 = alt按下，否则为0。

第30位为1时 = 消息前按下，否则为0。

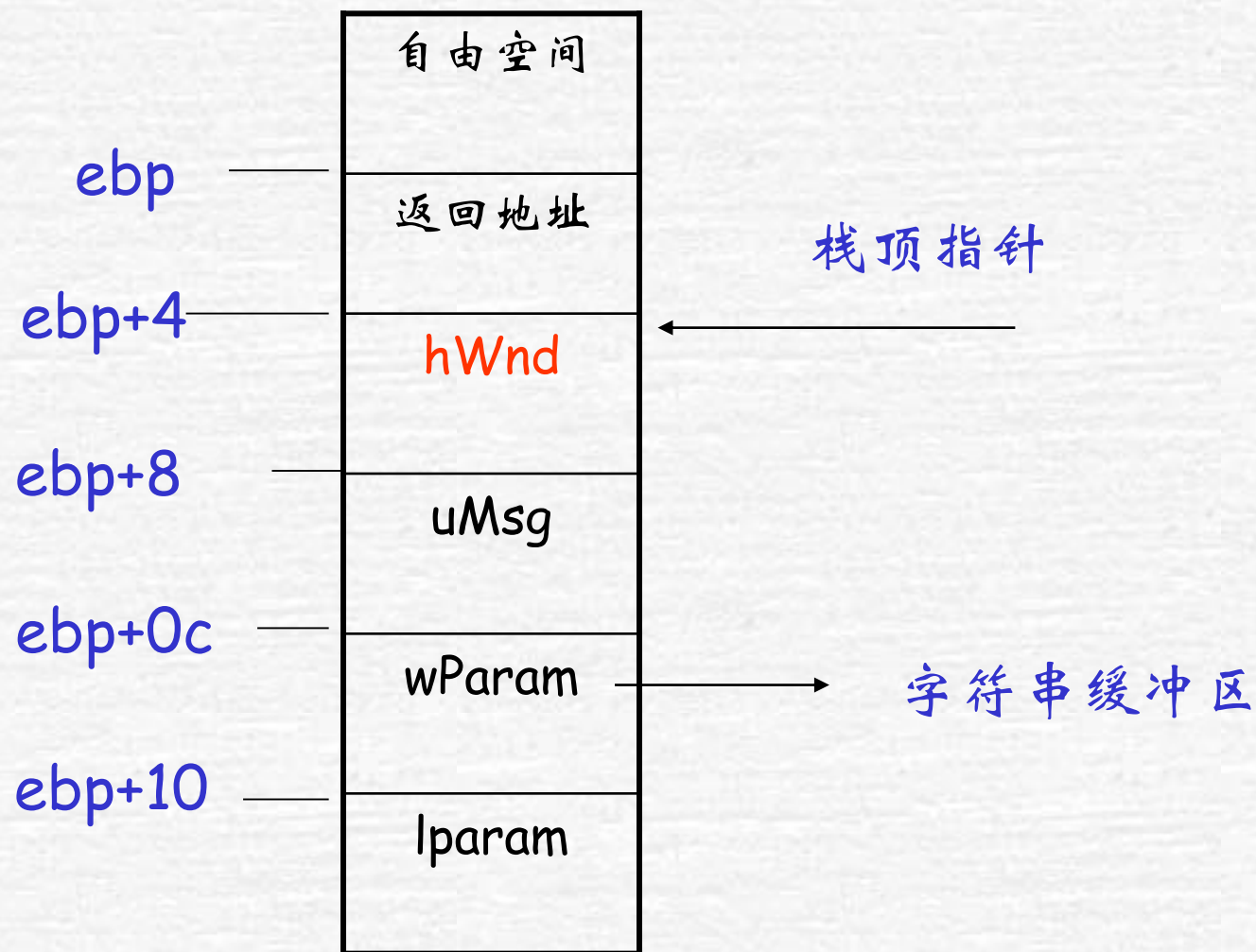
第31位为1时 = 正在被释放，否则为0。

当用户按下一个键，什么键，由这最后两个变量来说明。

消息的符号常数以WM\_开头，例如WM\_PAINT,相应于要求窗口的用户区部分重绘。又如

WM\_LBUTTONDOWN表示鼠标左键被按下。可参考MSDN Library.





# 设置断点


在ida 调试器中设置:

```
bpx 6c291b81 if (ebp->8) == WM_GETTEXT
```

退出调试器，运行输入密码比如“Hello”回车，则出现:

```
Break due to BPX #0008:6C291B81  
if( (esp→8)==0xD)
```

因 wparam 为最大字符数，所以密码存放在 lparam 中。

 **Breakpoint settings** ✕

Location

Condition  ...

Settings

- ☒ Enabled
- ☐ Hardware
- ☐ Module relative
- ☐ Symbolic
- ☒ Source code
- ☐ Low level condition

Actions

- ☒ Break
- ☐ Trace
- ☐ Refresh debugger memory
- ☐ Enable tracing
- ☐ Disable tracing

Tracing type

Hardware breakpoint mode

- ☐ Read
- ☐ Write
- ☐ Execute

Size

Group  [Edit breakpoint groups](#)

我们需要确定所读字符串的地址。指向缓冲区的指针转移为指向参数 lParam 所表示的缓冲区（参见 SDK 中关于 WM\_GETTEXT 的描述），而参数 lParam 被放置在堆栈中，其偏移地址相对于 ESP 的值为 0x10：

```
Return address  ← esp
hwnd           ← esp + 0x4
uMsg           ← esp + 0x8
wParam         ← esp + 0xC
lParam         ← esp + 0x10
```

现在，将缓冲区输出到数据窗口，用命令 P RET 来退出窗口过程，就能看见我们刚才输入的文本“Hello”：

```
:d *(esp+10)
:p ret
0023:0012EB28 48 65 6C 6C 6F 00 05 00-0D 00 00 00 FF 03 00 00 Hello.....
0023:0012EB38 1C ED 12 00 01 00 00 00-0D 00 00 00 FD 86 E1 77 .....w
0023:0012EB48 70 3C 13 00 00 00 00 00-00 00 00 00 00 00 00 p<.....
0023:0012EB58 00 00 00 00 00 00 00 00-98 EB 12 00 1E 87 E1 77 .....w
```



## 在bpm 0023: 12EB28的位置 设置断点后执行，能看到希望看到的代码

地看见如下的代码：

```
0008:A00B017C  8A0A      mov     cl, [edx]
0008:A00B017E  8808      mov     [eax], cl
0008:A00B0180  40        inc     eax
0008:A00B0181  42        inc     edx
0008:A00B0182  84C9      test    cl, cl
0008:A00B0184  7406      jz      A00B018C
0008:A00B0186  FF4C2410  dec     dword ptr [esp+10]
0008:A00B018A  75F0      jnz     A00B017C
```

哈！这个缓冲区是通过值而不是通过引用来传递的。系统不允许你直接访问缓冲区，它只是提供了一份拷贝。该缓冲区中的一个字符被拷贝到 CL，寄存器 EDX 的值指向该字符。显然，EDX 包含一个指向该缓冲区的指针，就是它导致调试器的出现。然后，它从 CL 中被拷贝到位置[EAX]，EAX 是一个指针（现在我们还不清楚它是什么）。两个指针都被加一，而且检查 CL（最后读取的一个字符）是否等于零。如果还没有到达字符串的结尾，就重复上述过程。如果你看见了两个缓冲区，就再设置另一个断点：

在 Windows 9x 中，消息的处理方式与 Windows NT 有些不一样。特别地，编辑窗口过程是用 16 位代码实现的，使用了令人讨厌的段内存模式：segment:offset。地址的传递方式也不一样。哪个参数包含段地址呢？为了回答这个问题，先看看 SoftIce 的断点报告：

```
Break due to BMSG 0428 WM_GETTEXT (ET=513.11 milliseconds)
hWnd=0428 wParam=0666 lParam=28D70000 msg=000D WM_GETTEXT
```

窗口句柄

要读的字符的最大数量

段地址

偏移地址

完整的地址都包含在 32 位参数 lParam 中，其中的 16 位用于表示段地址，另外的 16 位用于表示偏移地址。因此，断点的形式如下：bpm 28D7:0000.000000。