第五章　漏洞利用

知识点六：Windows安全防护技术
知识点七：地址定位技术
知识点八：API函数自搜索技术
知识点九：返回导向编程
知识点十：绕过其它安全防护

知识点六：Windows安全防护技术

由于C、C++等高级程序语言在边界检查方面存在的不足，致使缓冲区溢出漏洞等多种软件漏洞已成为信息系统安全的主要威胁之一，尤其对于使用广泛的Windows操作系统及其应用程序造成了极大的危害。为了能在操作系统层面提供对软件漏洞的防范，Windows操作系统自Vista版本开始，到现在普遍采用的Windows7/8/10等版本，陆续提供了多种防范措施和手段，对于提高Windows操作系统抵御漏洞攻击起到了关键作用。

下面介绍**Windows操作系统中提供的主要几种软件漏洞利用的防范技术**。

1 ASLR

地址空间分布随机化ASLR(addressspace layout randomization)是一项通过**将系统关键地址随机化,从而使攻击者无法获得需要跳转的精确地址**的技术。

Shellcode需要调用一些系统函数才能实现系统功能达到攻击目的，因为这些函数的地址往往是系统DLL（如kernel32. Dll）、可执行文件本身、栈数据或PEB（Process Environment Block，进程环境块）中的固定调用地址，所以为shellcode的调用提供了方便。

对于ASLR技术，微软从**操作系统加载时的地址变化**和**可执行程序编译时的编译器选项**两个方面进行了实现和完善。

系统加载地址变化

- ✓ **ASLR随机化的关键系统地址包括：PE文件(exe文件和dll文件)映像加载地址、堆栈基址、堆地址、PEB和TEB（Thread Environment Block，线程环境块）地址**等。
- ✓ 在Windows Vista上，当程序启动将执行文件加载到内存时，操作系统通过内核模块提供的ASLR功能，在原来映像基址的基础上加上一个随机数作为新的映像基址。
- ✓ 随机数的取值范围限定为1至254，并保证每个数值随机出现。

编译器选项-DYNAMICBASE

VS 2005及更高版本提供了选项/DYNAMICBASE，使用了该选项之后，编译后的程序每次运行时，其内部的栈等结构的地址都会被随机化。

实验四：在Windows 7及以后的操作系统里运行下述程序，查看地址变化情

```
#define DLL_NAME "kernel32.dll"
unsigned long gvar = 0;
void PrintAddress() {
        printf("PrintAddress的地址:%p \n", PrintAddress);
        gvar++;
}
int main(){
        HINSTANCE handle;
        handle = LoadLibrary(DLL_NAME);
        if (!handle) {
            printf(" load dll erro !");  exit(0);
        }
        printf("Kernel32.dll文件库的地址： 0x%x\n", handle);
        void *pvAddress = GetProcAddress(handle, "LoadLibraryW");
        printf("LoadLibrary函数地址：%p \n", pvAddress);
        PrintAddress();
        printf("变量gvar的地址：%p \n", &gvar);
        system("pause");
        return 0;
}
```

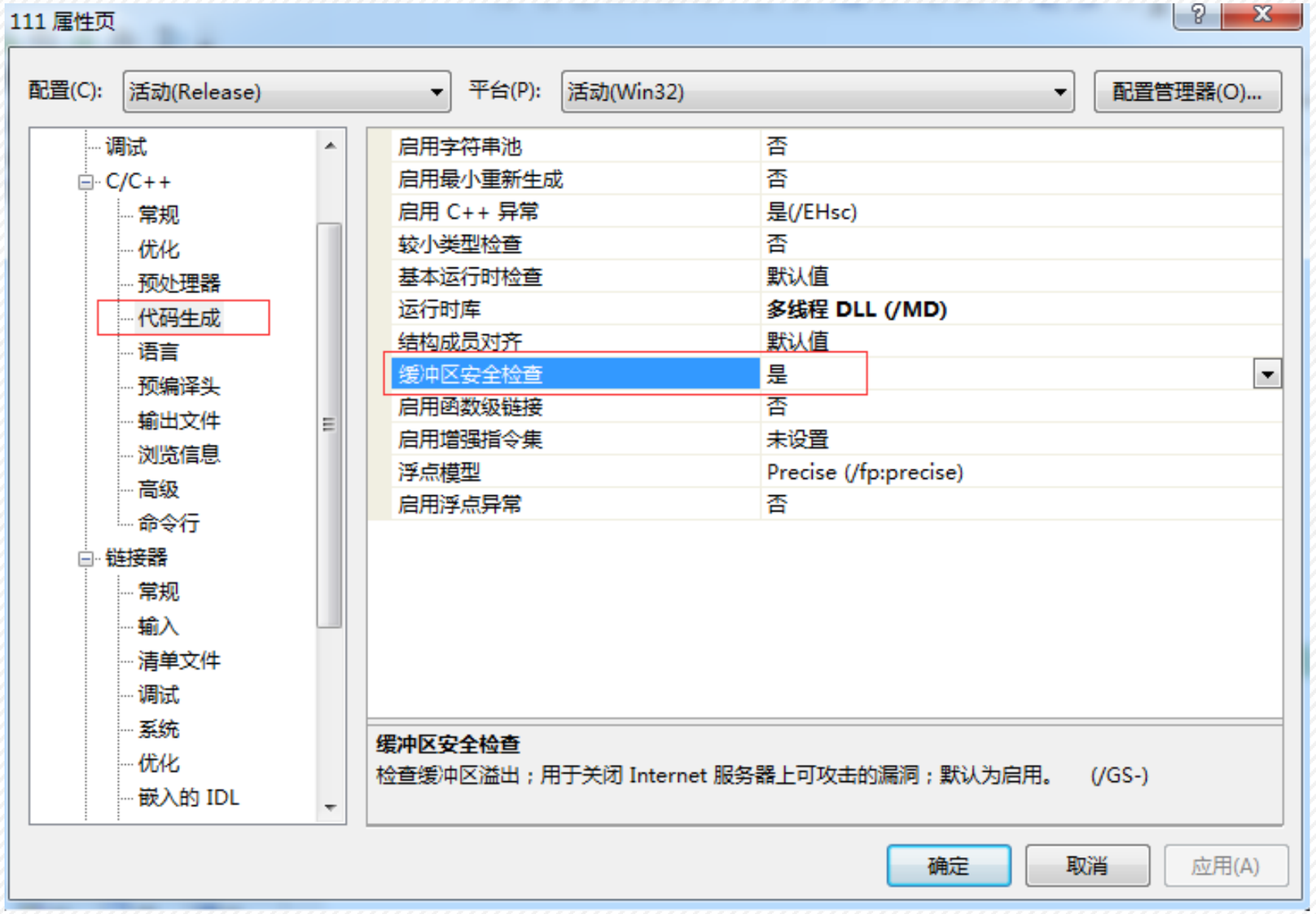
GetProcAddress函数

可以获得DLL中的函数地址

2 GS Stack protection

GS Stack Protection技术是一项缓冲区溢出的检测防护技术。VC++编译器中提供了一个/GS编译选项，在使用VC7.0、Visual Studio 2005及后续版本编译时都支持该选项，如选择该选项，**编译器针对函数调用和返回时添加保护和检查功能的代码，在函数被调用时，在缓冲区和函数返回地址增加一个32位的随机数**security_cookie，在函数返回时，调用检查函数检查security_cookie的值是否有变化。

启用的位置如下图所示：


111 属性页
配置(C): 活动(Release)
平台(P): 活动(Win32)
配置管理器(O)...
调试
C/C++
常规
优化
预处理器
代码生成
语言
预编译头
输出文件
浏览信息
高级
命令行
链接器
常规
输入
清单文件
调试
系统
优化
嵌入的 IDL
启用字符串池 否
启用最小重新生成 否
启用 C++ 异常 是(/EHsc)
较小类型检查 否
基本运行时检查 默认值
运行时库 多线程 DLL (/MD)
结构成员对齐 默认值
缓冲区安全检查 是
启用函数级链接 否
启用增强指令集 未设置
浮点模型 Precise (/fp:precise)
启用浮点异常 否
缓冲区安全检查
检查缓冲区溢出；用于关闭 Internet 服务器上可攻击的漏洞；默认为启用。 (/GS-)
确定
取消
应用(A)

security_cookie在进程启动时

security_cookie在进程启动时会随机产生，并且它的原始存储地址因Windows操作系统的ASLR机制也是**随机存放**的，攻击者无法对security_cookie进行篡改。

当发生栈缓冲区溢出攻击时

当发生栈缓冲区溢出攻击时，对返回地址或其他指针进行覆盖的同时，会覆盖security_cookie的值，因此在函数调用结束返回时，对security_cookie进行检查就会发现它的值变化了，从而发现缓冲区溢出的操作。

因此，GS技术对基于栈的缓冲区溢出攻击能起到很好的防范作用。

3 DEP

数据执行保护DEP(data execute prevention)技术可以**限制内存堆栈区的代码为不可执行状态**，从而**防范溢出后代码的执行。**

Windows操作系统中，默认情况下将包含执行代码和DLL文件的.text段即代码段的内存区域设置为可执行代码的内存区域。其他的内存区域不包含执行代码，应该不能具有代码执行权限，但是Windows XP及其之前的操作系统，没有对这些内存区域的代码执行进行限制。因此，对于缓冲区溢出攻击，攻击者能够对内存的堆栈或堆的缓冲区进行覆盖操作，并执行写入的shellcode代码。

启用DEP机制后，DEP机制将这些敏感区域设置不可执行的**non-executable标志位**，因此在溢出后即使跳转到恶意代码的地址，恶意代码也将无法运行，从而有效地阻止了缓冲区溢出攻击的执行。

3 DEP

DEP分为**软件DEP**和**硬件DEP**。硬件DEP需要CPU的支持,需要CPU在页表增加一个保护位NX(no execute)，来控制页面是否可执行。现在CPU一般都支持硬件NX，所以现在的DEP保护机制一般都采用的硬件DEP，对于DEP设置non-executable标志位的内存区域，CPU会添加NX保护位来控制内存区域的代码执行。

此外，Visual Studio**编译器提供了一个链接标志/NXCOMPAT**，可以在生成目标应用程序的时候使程序启用DEP保护。

4 SafeSEH

SEH

SEH（Structured Exception Handler）是Windows异常处理机制所采用的重要数据结构链表。程序设计者可以根据自身需要，定义程序发生各种异常时相应的处理函数，保存在SEH中。

通过精心构造，攻击者通过缓冲区溢出覆盖SEH中异常处理函数句柄，将其替换为指向恶意代码shellcode的地址，并触发相应异常，从而使程序流程转向执行恶意代码。

SafeSEH

SafeSEH就是一项保护SEH函数不被非法利用的技术。微软在编译器中加入了/SafeSEH选项，采用该选项编译的程序将PE文件中所有合法的SEH异常处理函数的地址解析出来制成一张SEH函数表，放在PE文件的数据块中,用于异常处理时候进行匹配检查。

4 SafeSEH

在该PE文件被加载时，系统读出该SEH函数表的地址，使用内存中的一个随机数加密，将加密后的SEH函数表地址、模块的基址、模块的大小、合法SEH函数的个数等信息，放入ntdll.dll的SEHIndex结构中。

在PE文件运行中，如果需要调用异常处理函数，系统会调用加解密函数解密从而获得SEH函数表地址，然后针对程序的每个异常处理函数检查是否在合法的SEH函数表中，如果没有则说明该函数非法，将终止异常处理。接着要检查异常处理句柄是否在栈上，如果在栈上也将停止异常处理。这两个检测可以防止在堆上伪造异常链和把shellcode放置在栈上的情况，最后还要检测异常处理函数句柄的有效性。

从Vista开始，由于系统PE文件在编译时都采用SafeSEH编译选项，因此以前那种通过覆盖异常处理句柄的漏洞利用技术，也就不能正常使用了。

5 SEHOP

结构化异常处理覆盖保护SEHOP（Structured Exception Handler Overwrite Protection）是微软针对SEH攻击提出的一种安全防护方案。

SEH攻击是指通过栈溢出或者其他漏洞，使用精心构造的数据覆盖SEH上面的某个函数或者多个函数，从而控制EIP（控制程序执行流程）。

5 SEHOP

SEHOP的核心是检测程序栈中的所有SEH结构链表的完整性，来判断应用程序是否受到了SEH攻击。

SEHOP针对下列条件进行检测，包括：

SEH结构都必须在栈上，最后一个SEH结构也必须在栈上；

所有的SEH结构都必须是4字节对齐的；

SEH结构中异常处理函数的句柄handle（即处理函数地址）必须不在栈上；

最后一个SEH结构的handle必须是ntdll!FinalExceptionHandler函数F等。

需要说明的是，虽然微软启用了GS、DEP、ASLR、SafeSEH、SEHOP等漏洞利用的防护技术，然而攻击者也在陆续发现着其他的漏洞利用手段，突破微软的防护技术。用魔高一尺道高一丈来描述两者间在漏洞利用技术上的对抗，一点也不为过。

接下来，介绍一些进一步的漏洞利用技术。

知识点七：地址定位技术

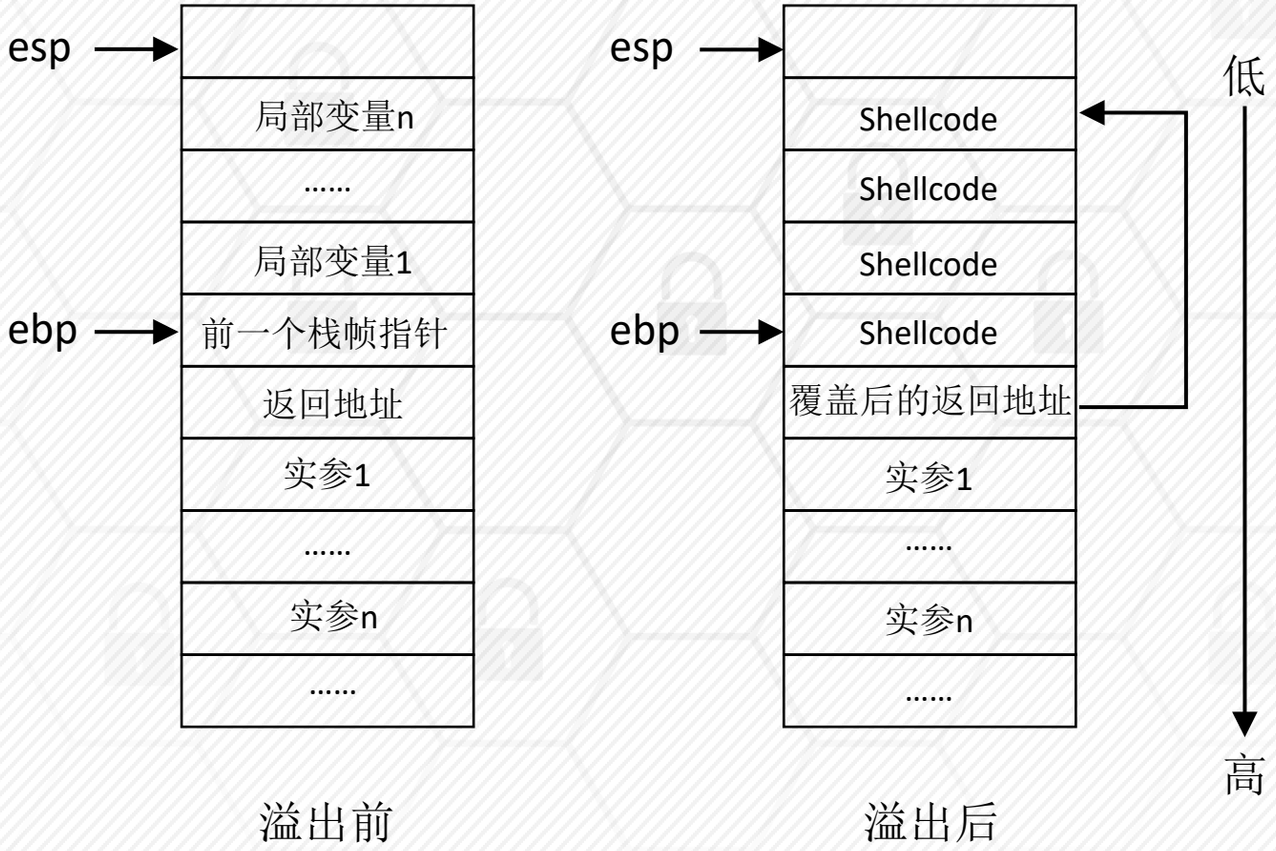
根据软件漏洞触发条件的不同，内存给调用函数分配内存的方式不同，shellcode的植入地址也不相同。下面根据shellcode代码不同的定位方式，介绍三种漏洞利用技术。

1 静态shellcode地址的利用技术

如果存在溢出漏洞的程序，是一个操作系统每次启动都要加载的程序，操作系统启动时为其分配的内存地址一般是固定的，则函数调用时分配的栈帧地址也是固定的。

这种情况下，溢出后写入栈帧的shellcode代码其内存地址也是静态不变的，所以可以直接将shellcode代码在栈帧中的静态地址覆盖原有返回地址。在函数返回时，通过新的返回地址指向shellcode代码地址，从而执行shellcode代码。

在shellcode为静态地址时，缓冲区溢出前后内存中栈帧的变化示意图参见下图。



2 基于跳板指令的地址定位技术

有些软件的漏洞存在于某些动态链接库中，它们在进程运行时被**动态加载**，因而在下一次被重新装载到内存中时，其**在内存中的栈帧地址是动态变化**的，则植入的shellcode代码在内存中的起始地址也是变化的。此外，如果在使用ASLR技术的操作系统中，地址会因为引入的随机数每次发生变化。

此时，需要让覆盖返回地址后新写入的返回地址能够自动定位到shellcode的起始地址。

为了解决这个问题，可以**利用esp寄存器**的特性实现：

- **在函数调用结束后，被调用函数的栈帧被释放，esp寄存器中的栈顶指针指向返回地址**在内存高地址方向的相邻位置。
- **可见，通过esp寄存器，可以准确定位返回地址所在的位置。**

利用这种特性，可以实现对shellcode的动态定位，具体步骤如下：

第一步，找到内存中任意一个汇编指令jmp esp，这条指令执行后可跳转到esp寄存器保存的地址，下面准备在溢出后将这条指令的地址覆盖返回地址。

第二步，设计好缓冲区溢出漏洞利用程序中的**输入数据**，使缓冲区溢出后，前面的填充内容为任意数据，紧接着**覆盖返回地址的是jmp esp指令的地址，再接着覆盖与返回地址相邻的高地址位置并写入shellcode代码。**

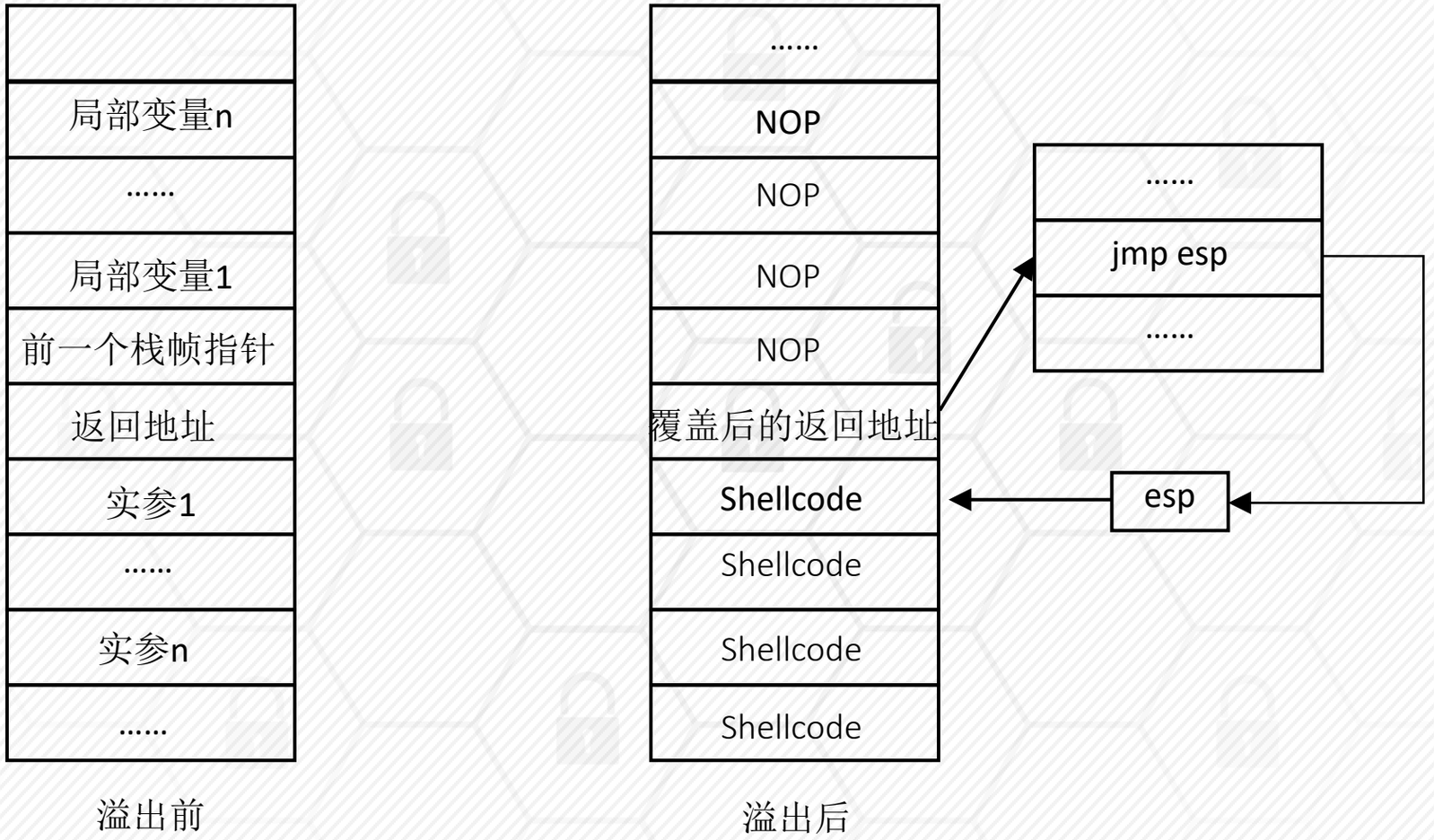
第三步，函数调用完成后函数返回，**根据返回地址中指向的jmp esp指令的地址去执行jmp esp操作**，即跳转到esp寄存器中保存的地址，而函数返回后esp中保存的地址是与返回地址相邻的高地址位置，在这个位置保存的是shellcode代码，则**shellcode代码被执行。**

上述方法使用jmp esp指令做为跳板，实现了在栈帧动态分配的情况下，可以自动跳回shellcode的地址并执行。

对于查找jmp esp的指令地址，可以在系统常用的user32.dll等动态链接库，或者其他被所有程序都加载的模块中查找，这些动态链接库或者模块加载的基地址始终是固定的。

虽然采用了ASLR技术，高版本windows系统有很多并没有受到ASLR保护的动态链接库或者系统函数，可以用来查找固定不变的jmp esp等指令。

以jmp esp做为跳板定位shellcode的内存地址示意图见下图。



除了jmp esp之外， mov eax,esp和jmp eax等指令序列也可以实现进入栈区的功能。

```
#include <stdio.h>
#include <windows.h>
#define DLL_NAME "user32.dll" //此处定义需要查找的dll名字

 int main()
 {
  BYTE *ptr;
  int position,address;
  HINSTANCE handle;
  BOOL done_flag = FALSE;
  handle = LoadLibraryA(DLL_NAME);  //LoadLibraryA 是调用dll的
函数名
  if(!handle)  //若没找到则进入该if
  {
   printf(" load dll error!");
   getchar();
   return 0;
  }
  ptr = (BYTE*)handle;
  printf("start at 0x%x\n",handle);
```

```
for(position = 0 ; !done_flag ; position++)
  {
   __try
   {
    if(ptr[position] == 0xFF && ptr[position+1] == 0xE4)
//jmp esp 的机器码为 E4FF
    {
     address = (int)ptr + position;
     printf("jmp esp found at 0x%x\n",address);
    }
   }
   __except(2)
   {
    address = (int)ptr + position;
    printf("END of 0x%x\n",address);
    done_flag = TRUE;
   }
  }
  getchar();
  return 0;
 }
```

通过上述程序运行就可以得到很多jmp esp的指令地址—XP下有效（没有ASLR）

3 内存喷洒技术

有些特殊的软件漏洞，不支持或者不能实现精确定位shellcode。同时，**存在漏洞的软件其加载地址动态变化，采用shellcode的静态地址覆盖方法难以实施。**由于堆分配地址随机性较大，为了**解决shellcode在堆中的定位以便触发，可以采用heap spray的方法。**

内存喷射技术的代表是堆喷洒Heap spray，也称为堆喷洒技术，是在shellcode的前面加上大量的**滑板指令（slide code）**，组成一个非常长的注入代码段。然后向系统申请大量内存，并且**反复用这个注入代码段来填充。**这样就使得内存空间被大量的注入代码所占据。攻击者再结合漏洞利用技术，**只要使程序跳转到堆中被填充了注入代码的任何一个地址，程序指令就会顺着滑板指令最终执行到shellcode代码。**

滑板指令

滑板指令（slide code）是由大量NOP(no-operation)空指令0x90填充组成的指令序列，当遇到这些NOP指令时，CPU指令指针会一个指令接一个指令的执行下去，中间不做任何具体操作，直到“滑”过最后一个滑板指令后，接着执行这些指令后面的其他指令，往往后面接着的是shellcode代码。

随着一些新的攻击技术的出现，**滑板指令除了利用NOP指令填充外，也逐渐开始使用更多的类NOP指令，譬如0x0C，0x0D（回车、换行）等。**

Heap Spray技术通过使用类NOP指令来进行覆盖，对shellcode地址的跳转准确性要求不高了，从而增加了缓冲区溢出攻击的成功率。然而，Heap Spray会导致被攻击进程的内存占用非常大，计算机无法正常运转，因而容易被察觉。

它一般配合堆栈溢出攻击，不能用于主动攻击，也不能保证成功。

针对Heap Spray，对于windows系统比较好的系统防范办法是开启DEP功能，即使被绕过，被利用的概率也会大大降低。

知识点八：API函数自搜索技术

API函数自搜索技术

前面的Shellcode都**采用硬编址的方式来调用相应API函数**。首先，获取所要使用函数的地址，然后将该地址写入ShellCode，从而实现调用。如果系统版本变了，很多函数的地址往往会发生变化，那么调用肯定就会失败了。**编写通用shellcode，shellcode自身就必须具备动态的自动搜索所需API函数地址的能力，即API函数自搜索技术。**

以MessageBoxA函数的调用的shellcode为例，来解释通用型shellcode的编写逻辑。

- MessageBoxA位于user32.dll中，用于弹出消息框。
- LoadLibraryA位于kernel32.dll中，用于加载user32.dll。

通用型Shellcode的编写逻辑

调用MessageBoxA函数，应该先使用LoadLibrary(“user32.dll”)装载user32.dll。定位LoadLibrary函数的步骤如下：

- **第一步：定位kernel32.dll。**
- **第二步：解析kernel32.dll的导出表**
- **第三步：搜索定位LoadLibrary等目标函数。**
- 第四步：基于找到的函数地址，完成Shellcode的编写。

难点在于第一步到第三步，即如何实现API函数自搜索。

所有的Win32程序都会自动加载ntdll.dll以及kernel32.dll这两个最基础的动态链接库，接下来，我们看看怎么完成对kernel32.dll里的API的搜索。

（1）定位kernel32.dll

[1] 首先通过段选择字FS在内存中**找到当前的线程环境块TEB**。

[2] 线程环境块偏移地址为**0x30的地址**存放着指向进程环境块PEB的指针。

[3] 进程环境块中偏移地址为0x0c的地方存放着指向PEB_LDR_DATA结构体的指针，其中，存放着已经被进程装载的动态链接库的信息。

[4] PEB_LDR_DATA结构体偏移位置为**0x1C的地址**存放着指向模块初始化链表的头指针 InInitializationOrderModuleList。

[5] 模块初始化链表InInitializationOrderModuleList中按顺序存放着PE装入运行时初始化模块的信息，第一个链表结点是ntdll.dll，第二个链表结点就是kernel32。

[6] 找到属于kernel32.dll的结点后，在其基础上再偏移0x08就是kernel32.dll在内存中的加载基地址。

（1）定位kernel32.dll

上述过程细节读者暂时不需要去深究，如上复杂的操作可以用如下简单的代码来实现：

```
int main()
{
    _asm
    {
        mov eax, fs:[0x30] ;PEB的地址
        mov eax, [eax + 0x0c] ; PEB_LDR_DATA结构体的地址
        mov esi, [eax + 0x1c] ; InInitializationOrderModuleList地址
        lodsd     ;取得是双字节,即mov eax,[esi],esi=esi+4;
        mov eax, [eax + 0x08] ;eax就是kernel32.dll的地址
    }
    return 0;
}
```

（2）定位kernel32.dll的导出表

找到了kernel32.dll，由于它也是属于PE文件，那么我们可以根据PE文件的结构特征，定位其导出表，进而定位导出函数列表信息，然后进行解析、遍历搜索，找到我们所需要的API函数。

定位导出表及函数列表的步骤如下：

[7] 从kernel32.dll加载基址算起，偏移0x3c的地方就是其PE头的指针。

[8] PE头偏移0x78的地方存放着指向函数导出表的指针。

[9] 获得导出函数偏移地址（RVA）列表、导出函数名列表：

- 导出表偏移0x1c处的指针指向存储导出函数偏移地址（RVA）的列表。
- 导出表偏移0x20处的指针指向存储导出函数函数名的列表。

（2）定位kernel32.dll的导出表

定位导出表及函数列表，可以用如下简单的代码来实现：

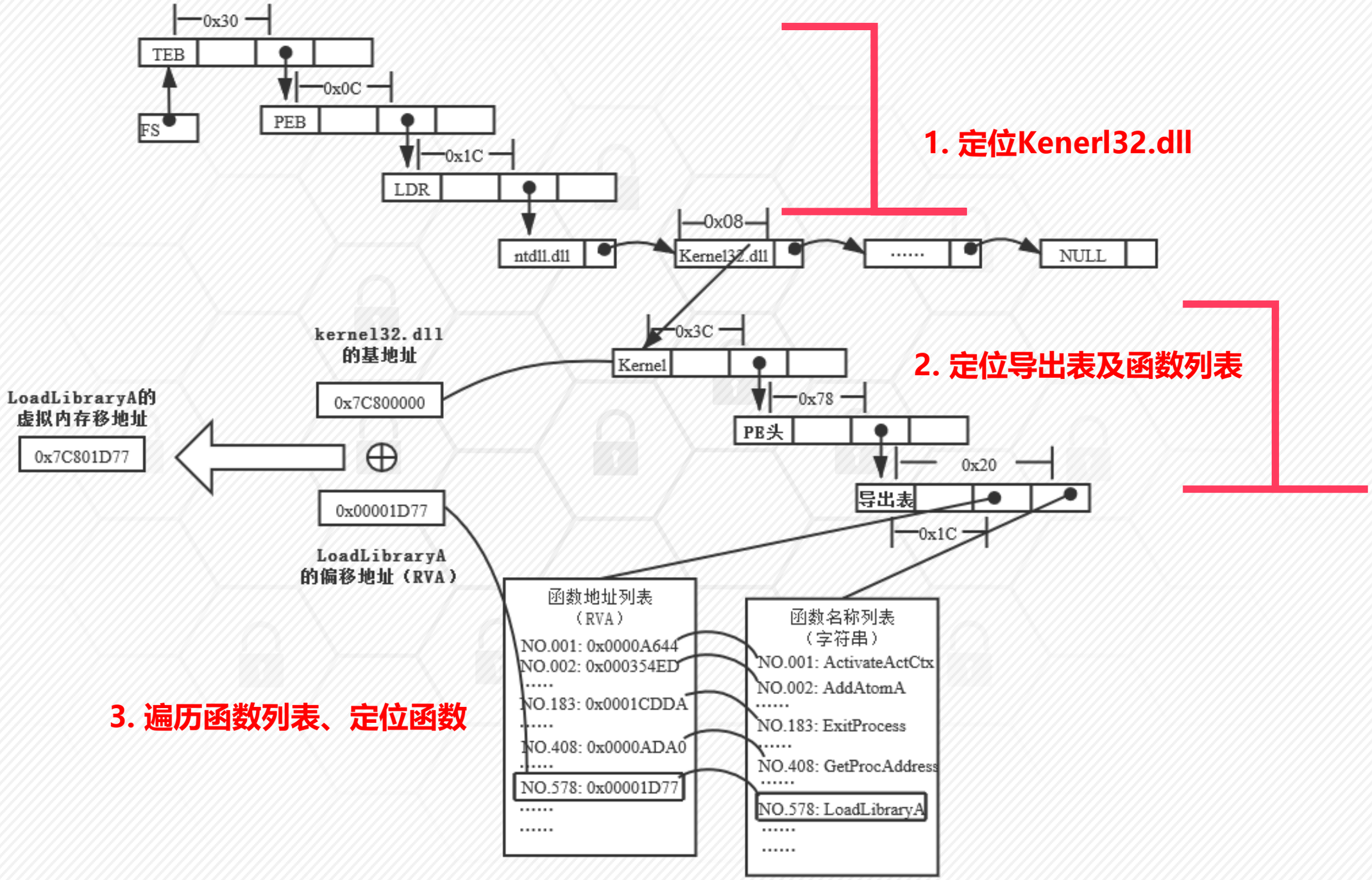
```
mov    ebp, eax                                    //将kernel32.dll基地址赋值给ebp
mov          eax,[ebp+0x3C]          //dll的PE头的指针（相对地址）
mov          ecx,[ebp+eax+0x78]          //导出表的指针（相对地址）
add          ecx,ebp                    // 得到导出表的内存地址
mov          ebx,[ecx+0x20]                  //导出函数名列表指针
add          ebx,ebp                     //导出函数名列表指针的基地址
```

（3）搜索定位目标函数

可以通过遍历两个函数相关列表，算出所需函数的入口地址：

- 函数的RVA地址和名字按照顺序存放在上述两个列表中，可以在名称列表中定位到所需的函数是第几个，然后在地址列表中找到对应的RVA。
- 获得RVA后，再加上前边已经得到的动态链接库的加载地址，就获得了所需API此刻在内存中的虚拟地址，这个地址就是最终在ShellCode中调用时需要的地址。

按照这个方法，就可以获得kernel32.dll中的任意函数。


0x30
TEB
FS
0x0C
PEB
0x1C
LDR
ntdll.dll
0x08
Kernel32.dll
.....
NULL
1. 定位Kenerl32.dll
kernel32.dll
的基地址
0x7C800000
0x3C
Kernel
0x78
PE头
0x20
导出表
0x1C
2. 定位导出表及函数列表
LoadLibraryA的
虚拟内存移地址
0x7C801D77
0x00001D77
LoadLibraryA
的偏移地址（RVA）
函数地址列表
（RVA）
NO.001: 0x0000A644
NO.002: 0x000354ED
.....
NO.183: 0x0001CDDA
.....
NO.408: 0x0000ADA0
.....
NO.578: 0x00001D77
.....
.....
函数名称列表
（字符串）
NO.001: ActivateActCtx
NO.002: AddAtomA
.....
NO.183: ExitProcess
.....
NO.408: GetProcAddress
.....
NO.578: LoadLibraryA
.....
.....
3. 遍历函数列表、定位函数

完整API函数自搜索代码。基于上述流程找到函数的入口地址，之后就可以编写自己的shellcode，如课本示例5-11，各位同学还需要自行阅读了解代码的工作原理。

完整的通用型Shellcode

基于上面提到的自定义API搜索技术，可以在找到函数入口地址之后，进行利用。

```
int main(){
   __asm   {
      CLD                    //清空标志位DF
      push   0x1E380A6A
//压入MessageBoxA的hash-->user32.dll
      push   0x4FD18963
//压入ExitProcess的hash-->kernel32.dll
      push   0x0C917432
//压入LoadLibraryA的hash-->kernel32.dll
      mov esi,esp
 //esi=esp,指向堆栈中存放LoadLibraryA的地址
      lea  edi,[esi-0xc]
//空出8字节应该是为了兼容性
      //======开辟一些栈空间
      xor                ebx,ebx
      mov                bh,0x04
      sub                esp,ebx          //esp-=0x400
```

1. ESI保存三个哈希值地址

```
//======压入"user32.dll"
      mov                bx,0x3233
      push ebx           //"\0 32"
      push 0x72657375          //"user"
      push esp
      xor                edx,edx          //edx=0
 //======找kernel32.dll的基地址
      mov                ebx,fs:[edx+0x30]
//[TEB+0x30]-->PEB
      mov                ecx,[ebx+0xC]
//[PEB+0xC]--->PEB_LDR_DATA
      mov                ecx,[ecx+0x1C]
      mov                ecx,[ecx]
//进入链表第一个就是ntdll.dll
      mov                ebp,[ecx+0x8]
//ebp= kernel32.dll的基地址
```

2. 保存user32.dll字符串地址

3. EBP保存kernel32.dll基地址

```
//======是否找到了
find_lib_functions:
    lodsd   //即mov eax,[esi],esi+=4, 第一次取LoadLibraryA的hash
    cmp   eax,0x1E380A6A      //hash比较
    jne   find_functions  //如果没有找到函数，继续找
    xchg
    call   [edi-0x8]    //LoadLibraryA("user32") |
    xchg  eax,ebp                6. 是,将执行LoadLibrary(user32.dll)
//ebp=user32.dll基地址,eax=MessageBoxA的hash  <-- |
 3.不是
    //======导出函数名列表指针
find_functions:
    pushad                  //保护寄存器
    mov  eax,[ebp+0x3C]        //dll的PE头
    mov  ecx,[ebp+eax+0x78]  //导出表的指针
    add   ecx,ebp    //ecx=导出表的基地址
    mov  ebx,[ecx+0x20]//导出函数名列表指针
    add   ebx,ebp    //ebx=导出函数名列表指针的基地址
    xor    edi,edi
//======找下一个函数名
next_function_loop:
    inc    edi
    mov    esi,[ebx+edi*4]     //从列表数组中读取函数名
    add   esi,ebp       //esi = 函数名称所在地址
    cdq                 //edx = 0
```

依次取入栈的函数哈希
最后一个是MessageBoxA

MessageBoxA？
（最后一个）

```
//======函数名的hash运算
hash_loop:
    movsx eax,byte ptr[esi]
    cmp    al,ah          //字符串结尾就跳出当前函数
    jz       compare_hash
    ror    edx,7
    add     edx,eax
    inc     esi
    jmp     hash_loop

//======比较找到的当前函数的hash是否是自己想找的
compare_hash:
    cmp     edx,[esp+0x1C]  //
    jnz     next_function_loop
    mov     ebx,[ecx+0x24]   //ebx = 顺序表的相对偏移量
    add     ebx,ebp         //顺序表的基地址
    mov     di,[ebx+2*edi]   //匹配函数的序号
    mov     ebx,[ecx+0x1C]   //地址表的相对偏移量
    add     ebx,ebp         //地址表的基地址
    add     ebp,[ebx+4*edi]  //函数的基地址
    xchg    eax,ebp         //eax<==>ebp 交换

    pop     edi
    stosd                   //把找到的
    push    edi
    popad                   //一次性完
    cmp eax,0x1e380a6a  //找到函数MessageBox后，跳出循环
    jne     find_lib_functions
```

4. 如果不是要找的函数，跳转到前面继续判断

5. 找到了存到EDI寄存器位置，如果不是最后一个，跳转到前面find_lib_functions位置继续

```
 //======让他做些自己想做的事
function_call:
        xor                 ebx,ebx
        push ebx
        push  0x74736577
        push 0x74736577     //push "westwest"
        mov                 eax,esp
        push ebx
        push eax
        push eax
        push ebx
        call    [edi-0x04]
//MessageBoxA(NULL,"westwest","westwest",NULL)
        push ebx
        call    [edi-0x08]       //ExitProcess(0);
        nop
        nop
        nop
        nop
    }
    return 0;
}
```

这个完整的代码，实现了MessageBox弹窗的功能，而且可以在任意系统里执行。

前面将找到的函数地址保存到了EDI所保存的地址区域，是按照前面依次入栈的顺序，即MessageBox、ExitProcess和Loadlibrary的入口地址。

整个Shellcode用nop指令作为结束。
之后，可以用上述Shellcode提取方式来得到对应的机器码。

知识点九：返回导向编程

DEP技术可以限制内存堆栈区的代码为不可执行状态，从而防范溢出后代码的执行，已经成为Windows的重要保护措施，但是它依然可以被绕过。

支持硬件DEP的CPU会拒绝执行被标记为不可执行的(NX)内存页的代码。

当我们尝试在启用DEP的内存执行代码，程序将会返回访问冲突STATUS_ACCESS_VIOLATION (0xc0000005) 并终止程序，对于攻击者来说这显然不是好事。

然而，考虑应用可用性，程序有时候需要在不可执行区域执行代码，这意味着调用某个Windows API可以把某段不可执行区域设置为可执行。

在DEP保护下，怎么去编写shellcode来完成函数调用呢?

1 基本思想

ROP

ROP的全称为Return-oriented programming（返回导向编程）；

是一种新型的**基于代码复用技术**的攻击，**它从已有的库或可执行文件中提取指令片段，构建恶意代码。**

ROP的基本思想

借助已存在的代码块(也叫配件，Gadget)，这些**配件来自程序已经加载的模块；**
在已加载的模块中找到一些**以retn结尾的配件**，把这些配件的地址布置在堆栈上, 当控制EIP并返回时候, 程序就会跳去执行这些小配件；
这些小**配件是在别的模块代码段**, 不受DEP的影响。

对于ROP技术，可以总结为如下三点：

1. ROP通过ROP链（retn）实现有序汇编指令的执行。
2. ROP链由一个个ROP小配件（Gadget，相当于一个小节点）组成。
3. ROP小配件由“目的执行指令+retn指令组成”。

2 示例

??????=>表示当前返回地址里包含的指令及跳转到该指令处执行

示例5-11（**指针只执行retn**）：初始状态：即将执行retn命令，此时ESP指向返回地址，而且返回地址以及后面4个地址里都通过溢出覆写了多个RETN指令的地址。

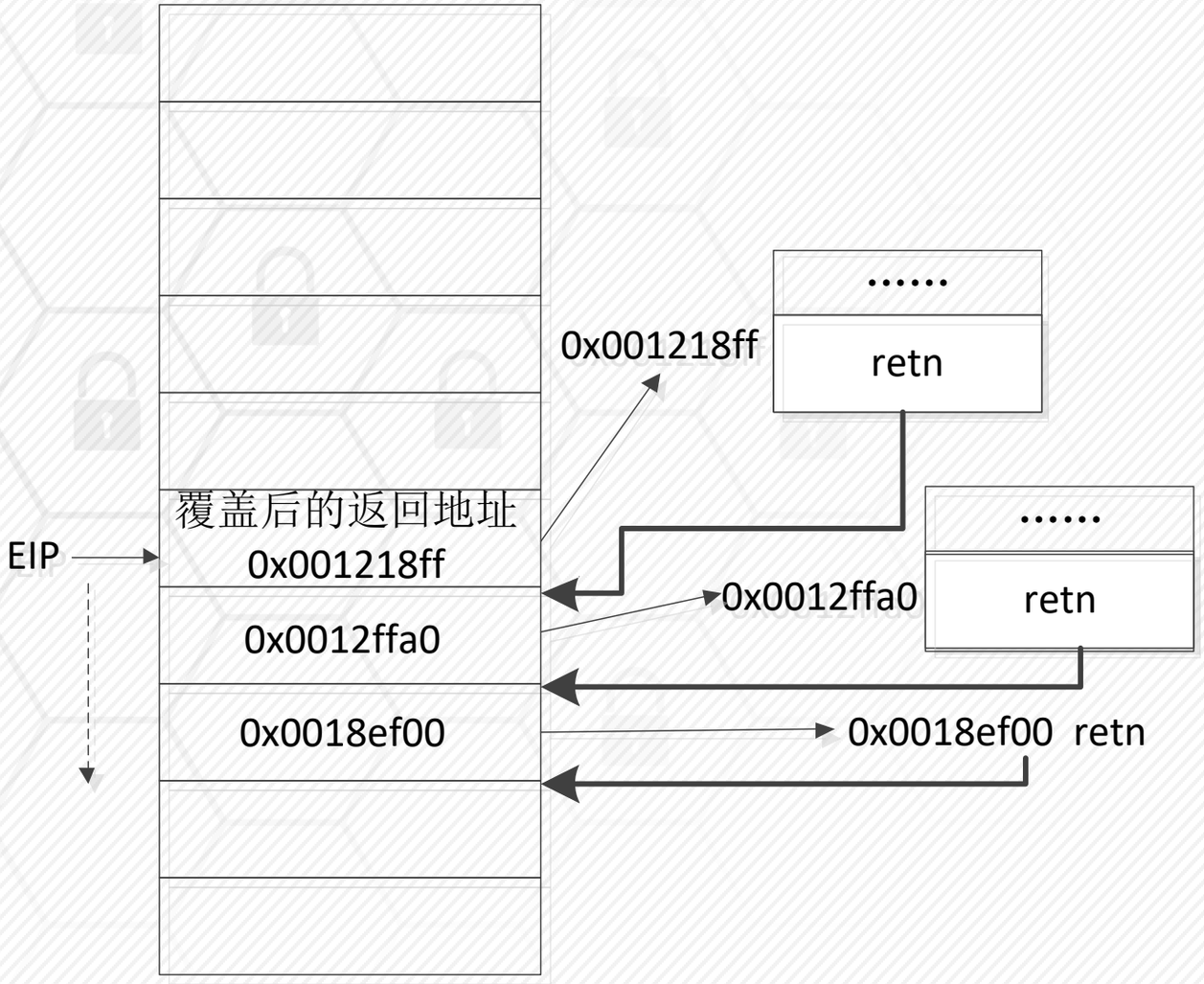
初始状态，EIP指针指向命令retn

ESP -> ???????? => RETN

???????? => RETN

???????? => RETN

???????? => RETN



下面来演示“指令+retn”的配件的用法，实现一定的逻辑。示例5-7（指针指向一些指令+retn）：

```
ESP -> ???????? => POP EAX # RETN
       ffffffff => we put this value in EAX
       ???????? => INC EAX # RETN
       ???????? => XCHG EAX,EDX # RETN
```

在这个例子中：

[1].执行retn之后，EIP指向了指令段“POP EAX #RETN”，ESP指向了高地址中的“0xffffffff”。此时，执行POP EAX，结果是将0xffffffff赋值给EAX；

[2].然后执行RETN的时候，EIP指向了“INC EAX # RETN”，ESP指向了下一个高地址。此时，执行INC EAX，这样EAX的值将由0xffffffff变为0；

[3].再执行RETN的时候，EIP指向了“XCHG EAX,EDX # RETN”，ESP指向了下一个高地址。此时，执行XCHG EAX,EDX，这样EDX的值就变为0。

可见，通过上面的ROP指令段，我们实现了将EDX置0的结果。

3 基于ROP的漏洞利用

ROP可以通过一些小配件构建期待的目标指令序列，但是因为它严重依赖内存中已存在的代码序列，因此，构建复杂和大规模的代码序列是非常难的。

在实际应用中，**基于ROP编写的代码序列可以利用有限的编码完成下述目标来达到攻击的目的**：

1. **调用相关API关闭或绕过DEP保护。**相关的API包括SetProcessDEPPlolicy、VirtualAlloc、NtSetInformationProcess、VirtualProtect等，比如VirtualProtect函数可以将内存块的属性修改为Executable。
2. **实现地址跳转**，直接转向不受DEP保护的区域里保存的shellcode执行。
3. **调用相关API将shellcode写入不受DEP保护的可执行内存。**进而，配合基于ROP编写的地址跳转指令，完成漏洞利用。

知识点十：绕过其它安全防护

漏洞又称为脆弱性，本书的一个观点就是只要有不健壮的地方，就存在被利用的可能。正所谓道高一尺、魔高一丈，接下来，我们简要介绍对于GS安全机制、ASLR机制、SEH保护机制等安全防护策略的绕过策略。

1 绕过GS安全机制

Visual Studio在实现**GS安全机制的时候，除了增加Cookie，还会对栈中变量进行重新排序**，比如：将字符串缓冲区分配在栈帧的最高地址上，因此，当字符串缓冲区溢出，就不能覆盖本地变量了。

但是，**考虑到效率问题，它仅按照函数隐患及危害程度进行选择性保护，因此有一部分函数可能没有得到有效的保护。**比如：**结构成员因为互操作性问题而不能重新排列**，因此当它们包含缓冲区时，这个缓冲区溢出就可以将之后其它成员覆盖和控制。

1 绕过GS安全机制

正是因为GS安全机制存在这些缺陷，所以聪明的攻击者构造出了各种办法来绕过GS保护机制。David Litchfield在2003年提出了一个技术来绕过GS保护机制：如果Cookie被一个不同的值覆盖了，代码会检查是否安装了安全处理例程，如果没有，系统的异常处理器就将接管它。

如果黑客**覆盖掉了一个异常处理结构，并在Cookie被检查前触发一个异常，这时栈中虽然仍然存在Cookie，但是还是可以被成功溢出。**这个方法相当于是利用SEH进行漏洞攻击。可以说，**GS安全机制最重要的一个缺陷是没有保护异常处理器**，但这点上虽然有SEH保护机制作为后盾，但SEH保护机制也是可以被绕过的。

2 ASLR缺陷和绕过方法

ASLR通过增加随机偏移使得攻击变得非常困难。但是，**ASLR技术存在很多脆弱性**：

（1）为了减少虚拟地址空间的碎片，操作系统把随机加载库文件的**地址限制为8位**，即地址空间为256，而且**随机化发生在地址前两个最有意义的字节**上；

（2）**很多应用程序和DLL模块并没有采用/DYNAMICBASE的编译选项**；

（3）很多**应用程序使用相同的系统DLL文件**，这些系统DLL加载后地址就确定下来了，对于**本地攻击，攻击者还是很容易就能获得所需要的地址**，然后进行攻击。

针对这些缺陷，还有一些其他绕过方法，比如**攻击未开启地址随机化的模块（作为跳板）、堆喷洒技术、部分返回地址覆盖法**等。

3 SEH保护机制缺陷和绕过方法

当一个进程中存在一个不是/SafeSEH编译的DLL或者库文件的时候，整个SafeSEH机制就可能失效。因为/SafeSEH编译选项需要.NET的编译器支持，**现在仍有大量第三方库和程序没有使用该编译器编译或者没有启动/SafeSEH选项。**

目前，较为可行的绕过SafeSEH的方法有：

- **利用未开启SafeSEH的模块作为跳板绕过**：可以在未启用SafeSEH的模块里找一些跳转指令，覆盖SEH函数指针，由于这些指令在未启用SafeSEH的模块里，因此异常触发时，可以执行到这些指令。
- **利用加载模块之外的地址进行绕过：**可以利用加载模块之外的地址，包括从堆中进行绕过或者其他一些特定内存绕过，具体不展开介绍。