软安

其他常见漏洞

格式化字符串漏洞

数据泄露

第一类：`printf("%s %d %d %d %x\n",buf,a,b,c);`

需要注意 Debug 模式与 Release 模式的栈帧区别。

数据写入

第二类：

%n：将已经打印的字符数（或字节数）存储到对应的整型指针变量中。

```
int count;
printf("Hello, world!%n", &count);
```

那么依据此原理，执行 `printf("%100d%n\n", num, &num);` 就会将 num 置为 100。（这是补空格填充，想要补 0 的话使用 "0100d%n"）

Sprintf 函数：把格式化的数据写入某个字符串缓冲区。第一个参数是目标缓冲区，后面的参数集体作为格式化字符串。

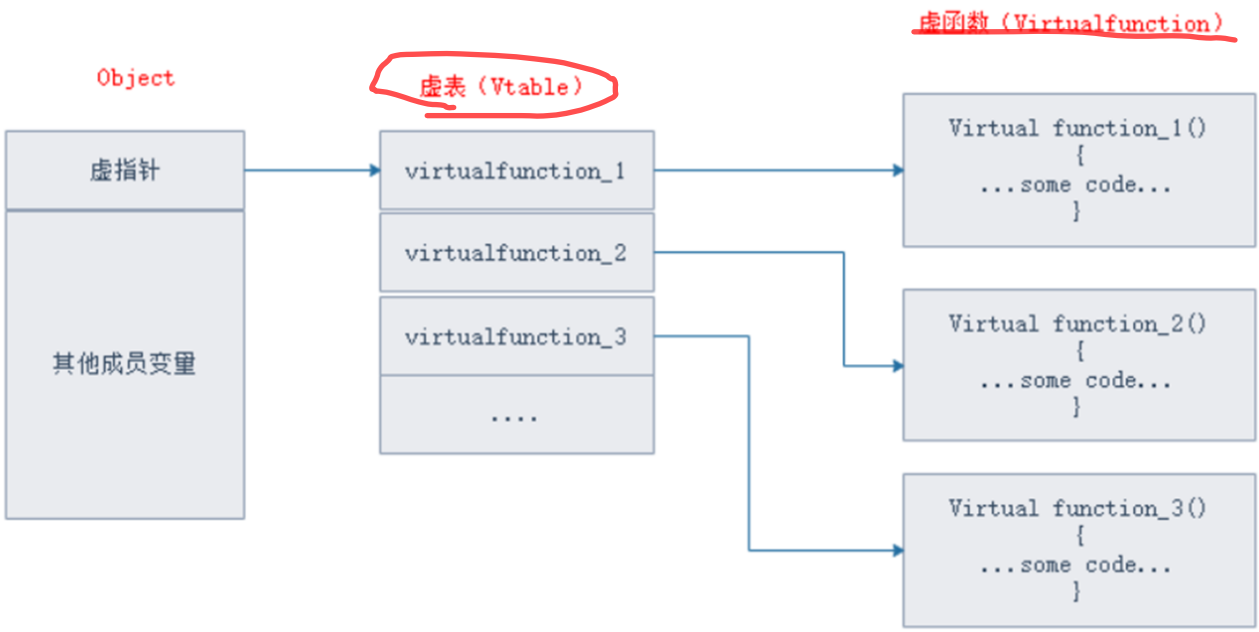
```
int formatstring_func2(int argc, char *argv[]) {
    char buffer[100];
    sprintf(buffer, argv[1]);
}
```

如果调用这段程序时用 "aaaabbbbcc%n" 作为命令行参数，数值10就会被写入地址为0x61616161（aaaa）的内存单元。

整数溢出漏洞

1. 存储溢出
2. 运算溢出
3. 符号问题

攻击C++虚函数



对象使用虚函数时：

1. 调用虚表指针找到虚表。
2. 然后从虚表中取出最终的函数入口地址进行调用。

如果虚表里存储的虚函数指针被篡改，程序调用虚函数的时候就会执行篡改后的指定地址的shellcode，就会发动虚函数攻击。

其他类型漏洞

注入类漏洞

注入类攻击都具备一个共同的特点：来自外部的输入数据被当作代码或非预期的指令、数据被执行，从而将威胁引入到软件或者系统。

根据应用程序的工作方式，将代码注入分为两大类：

- 二进制代码注入，即将计算机可以执行执行的二进制代码注入到其他应用程序的执行代码中。由于程序中某些缺陷导致程序的控制器被劫持，使得外部代码获得执行机会，从而实现特定的攻击目的；
- 脚本注入，即通过特定的脚本解释类程序提交可被解释执行的数据。由于应用在输入的过滤上存在缺陷，导致注入的脚本数据被执行。

下面介绍几种Web场景下的代码注入攻击。

1. SQL 注入
2. 操作系统命令注入
3. Web 脚本语言注入

4. SOAP 注入

权限类漏洞

1. **水平越权**就是相同级别（权限）的用户或者同一角色的不同用户之间，可以越权访问、修改或者删除的非法操作。
2. **垂直越权**又被分为向上越权与向下越权。
   向上越权是指一个低权限用户或者根本没权限也可以做高权限用户相同的事情；向下越权是一个高级别用户可以访问一个低级别的用户信息

漏洞利用基础

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。Exploit 的英文意思就是利用，它在黑客眼里就是漏洞利用。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

1. shell：命令解释器，它解释由用户输入的命令并且把它们送到内核。
2. shellcode：现在，“shellcode” 已经表达的是广义上的植入进程的代码，而不是狭义上的仅仅用来获得 shell 的代码。

漏洞利用的核心

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

要完成控制流劫持和达到不同攻击的目的，exploit 最终是需要执行 shellcode 的，但 exploit 中并不仅仅是 shellcode。exploit 要想达到攻击目标，需要做的工作更多，如对应的触发漏洞、将控制权转移到 shellcode 的指令一般均不相同，而且这些语句通常独立于 shellcode 的代码。这些能实现特定目标的 exploit 的有效载荷，称为 payload。

一个经典的比喻，将漏洞利用的过程可以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 是导弹发射装置，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 部分。

总的来说，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执行 shellcode。

> Exploit（利用）- 利用是攻击者或渗透测试人员利用系统、应用程序或服务中的漏洞的手段。攻击者使用利用来以开发人员未曾预料到的方式攻击系统，以实现特定的预期结果。常见的利用包括缓冲区溢出、Web应用程序漏洞（如SQL注入）和配置错误。

> Payload（有效载荷）- 有效载荷是攻击者希望系统执行的自定义代码，并由框架进行选择和传递。例如，反向 shell 是一种有效载荷，它在目标机器上创建一个连接，将其作为 Windows 命令提示符返回给攻击者，而绑定 shell 是一种有效载荷，它在目标机器上将命令提示符"绑定"到一个监听端口上，攻击者可以连接到该端口。有效载荷也可以是一些要在目标操作系统上执行的简单命令。

> Shellcode（外壳代码）- Shellcode 实际上是一系列精心设计的命令列表，一旦代码被注入到运行中的应用程序中，就可以执行。它是在利用漏洞时用作有效载荷的一系列指令。Shellcode 通常使用汇编语言编写。在大多数情况下，在目标机器执行一组指令后，将提供一个命令 shell 或 Meterpreter shell，因此得名"Shellcode"。

覆盖临接变量示例

代码中若存在 strcpy，那么可以通过缓冲区溢出覆盖临接变量。

```
#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);
}
```

Shellcode 代码植入示例

后面这三个部分可以参照软件安全实验 5：。

目标：植入一段代码，使其达到可以淹没返回地址，该返回地址将执行一个 MessageBox 函数，弹出窗体。

依旧是上面一段代码，buffer 缓冲区输入：[构造的机器码][填充对齐][覆盖地址]，具体构造的机器码这里略去。

Shellcode 编写

Shellcode 编写的难点：

1. **对一些特定字符需要转码**。比如，对于 strcpy 等函数造成的缓冲区溢出，会认为 NULL 是字符串的终结，所以 shellcode 中不能有 NULL，如果有需要则要进行变通或编码。
2. **函数 API 的定位很困难**。比如，在 Windows 系统下，系统调用多数都是封装在高级 API 中来调用的，而且不同的 Service Pack 或版本的操作系统其 API 都可能有所改动，所以不可能直接调用。因此，需要采用动态的方法获取 API 地址。

一种简单的编写 Shellcode 的方法的步骤如下：

1. 第一步：用 c 语言书写要执行的 Shellcode
2. 第二步：使用调试功能换成对应的汇编代码
3. 第三步：编写内嵌汇编代码，根据汇编代码在调试时的内存地址，找到对应地址中的机器码

构造任意字符串：

“hello world”的 ASCII 码为：\x68\x65\x6C\x6C\x6F\x20\x77\x6F\x72\x6C\x64\x20。注意小端字节序。

Shellcode 编码

Shellcode 代码编制过程通常需要进行编码，因为：

1. **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。
2. **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。
3. **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”。

编码方案：

对于网页 Shellcode，可以采用 **base64 编码**。

对于二进制 Shellcode 机器代码的编码，通常采用类似“加壳”思想的手段，采用：

1. 自定义编码（异或编码、计算编码、简单加解密等）的方法完成shellcode的编码
2. 通过精心构造精简干练的解码程序，放在 shellcode 开始执行的地方，完成 shellcode 的编解码；当 exploit 成功时，shellcode 顶端的解码程序首先运行，它会在内存中将真正的 shellcode 还原成原来的样子，然后执行。

异或编码是一种简单易用的 shellcode 编码方法，它的编解码程序非常简单。但是，它也存在很多限制，比如在选取编码字节时，不可与已有字节相同，否则会出现 0。

上述实验 5 就是使用异或编码。缓冲区结构：[解码程序][NOP 对齐][编码后的 Shellcode][覆盖返回地址（解码程序起始地址）]。

漏洞利用技术

Windows 安全防护技术

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。

   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。

   - （主体：操作系统；变化：加载基址）系统加载地址变化：PE 文件（可执行文件和动态链接库）在加载到内存时，其基址将会随机化。在原来的基址上加上一个随机数，从而使每次加载时的基址都不同，增加攻击者定位和利用特定函数或代码的难度。

- （主体：编译器；变化：内部结构）编译器选项-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。

2. **GS Stack protection**

（**用于缓冲区溢出的检测防护**）当启用 GS Stack Protection 时，编译器会为每个函数调用和返回生成一个 32 位的随机数，称为 security_cookie（安全令牌）。这个随机数会被插入到函数的栈帧中，并在函数返回时进行验证。

- 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。也就是：确保只有指定的代码区域（包含执行代码和 DLL 文件的 .text 段）才能被执行，而非代码区域则被禁止执行。

DEP（Data Execution Prevention）分为**软件 DEP 和硬件 DEP** 两种形式。硬件 DEP 需要 CPU 的支持，并通过在页表中添加一个保护位，即 NX（No Execute），来控制页面是否可执行。

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时启用 DEP 保护。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。

4. **SafeSEH**

SafeSEH（Safe Structured Exception Handling）是一项用于保护 SEH（Structured Exception Handling）函数不被非法利用的技术。SEH 是一种在 Windows 操作系统中用于异常处理的机制，用于处理软件中的异常情况，如访问冲突、除零等。然而，恶意攻击者可以利用缓冲区溢出等漏洞来劫持控制流并执行恶意代码，包括破坏 SEH 链表并绕过异常处理机制。

为了增强对 SEH 的保护，微软在其编译器中引入了 /SafeSEH 选项。使用该选项编译的程序将在 PE（Portable Executable）文件中创建一个 SEH 函数表。该表包含了所有合法的 SEH 异常处理函数的地址，它们在编译时被解析并存储在数据块中。在程序运行时，当发生异常时，系统会使用这张 SEH 函数表进行匹配和检查，以确保异常处理函数的合法性。

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（可以采用简单的位运算、异或操作或其他加密算法来实现，将 SEH 函数表的地址与随机数进行混淆），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**

   这也是针对 SEH 攻击提出的一种安全防护方案。（微软提出）（注意与 SafeSEH 的区别：SafeSEH 主要使用 SEH 函数表，SafeSEH 表通常存储在 PE 头部的异常目录（Exception Directory）中）

   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 函数等。

接下来，介绍一些进一步的漏洞利用技术。

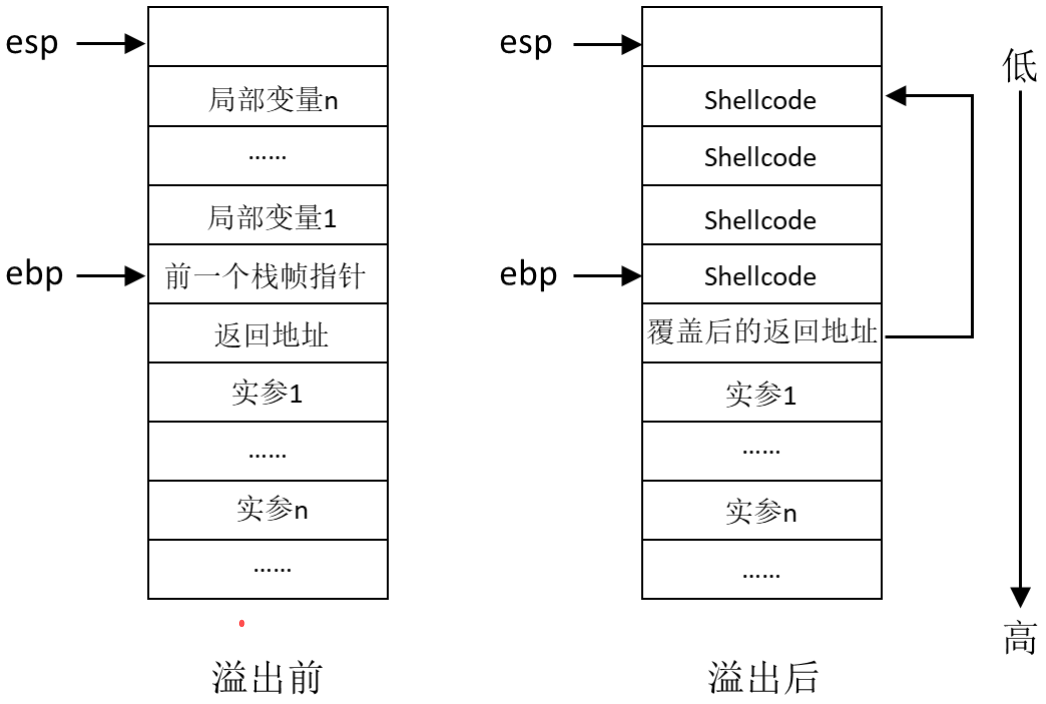
地址定位技术

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。



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。

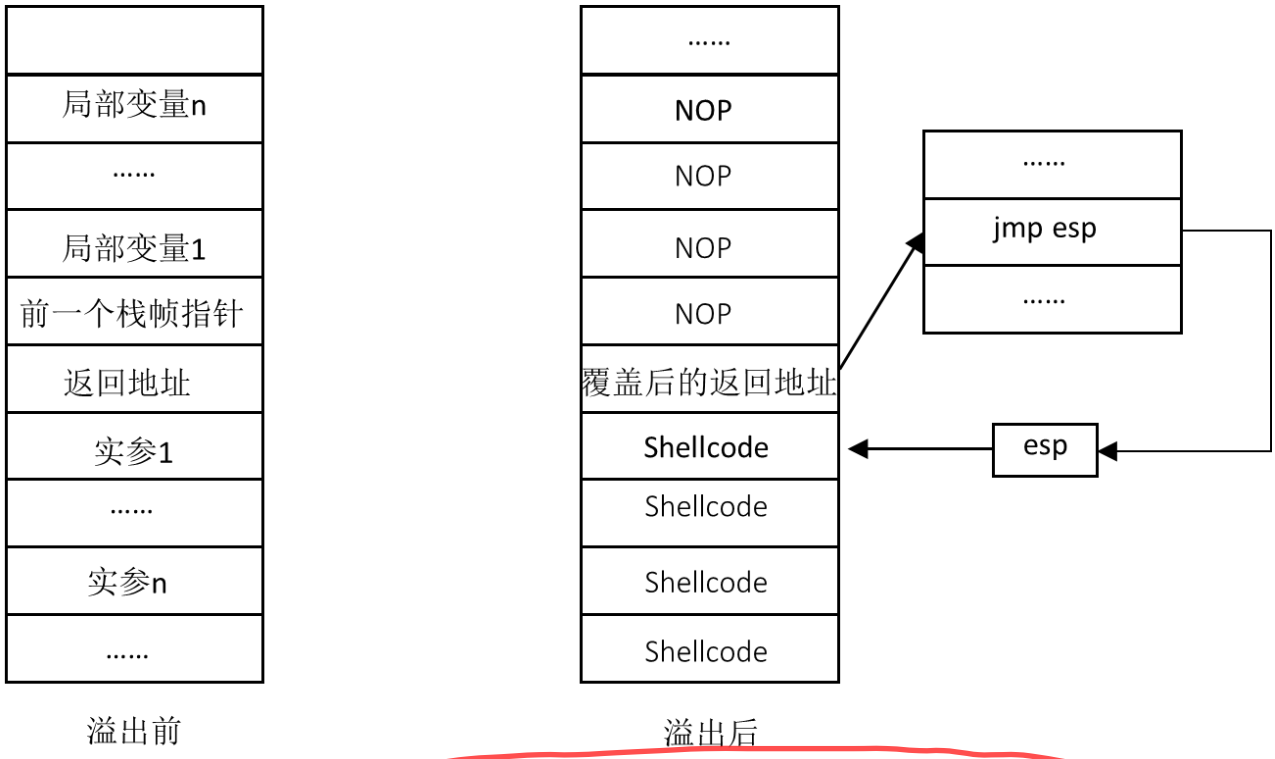
利用这种特性，可以实现对 shellcode 的动态定位，具体步骤如下：

- 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。

个人对于“攻击者会精心构造输入数据，使得缓冲区溢出后覆盖返回地址的位置正好是 jmp esp 指令的地址”的理解：**这里是因为是相对地址，所以可以轻易构造（通过偏移量覆盖返回地址）；而我们需要的 shellcode 的地址是绝对地址，由于 ASLR 技术，需要动态定位。**

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。**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 jmp esp 的指令地址**。这些动态链接库或者模块加载的基地址始终是固定的。

跳板示意图：



溢出前　　溢出后

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。

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。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。（更容易绕过检测）

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，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。

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

1. **调用 MessageBoxA 函数之前，需要先使用 LoadLibrary("user32.dll") 来加载 user32.dll**。以下是定位 LoadLibrary 函数的步骤：**第一步**：定位 kernel32.dll。需要找到 kernel32.dll 的基址，可以通过获取当前模块的句柄（GetModuleHandle(NULL)）来获取 kernel32.dll 的基址。

   第二步：解析 kernel32.dll 的导出表。导出表中包含了 kernel32.dll 中所有导出函数的名称和地址。

   第三步：搜索定位 LoadLibrary 等目标函数。遍历导出表，搜索函数名与目标函数名（如 LoadLibrary）匹配的项，获取其地址。

   第四步：基于找到的函数地址，完成 Shellcode 的编写。使用找到的 LoadLibrary 函数的地址作为参数，构造 Shellcode 来调用 LoadLibrary。

   难点在于第一步到第三步，即如何实现 API 函数的自动搜索。
2. 后面的步骤较多，代码较难，见实验 6 实验报告 。

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。

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。

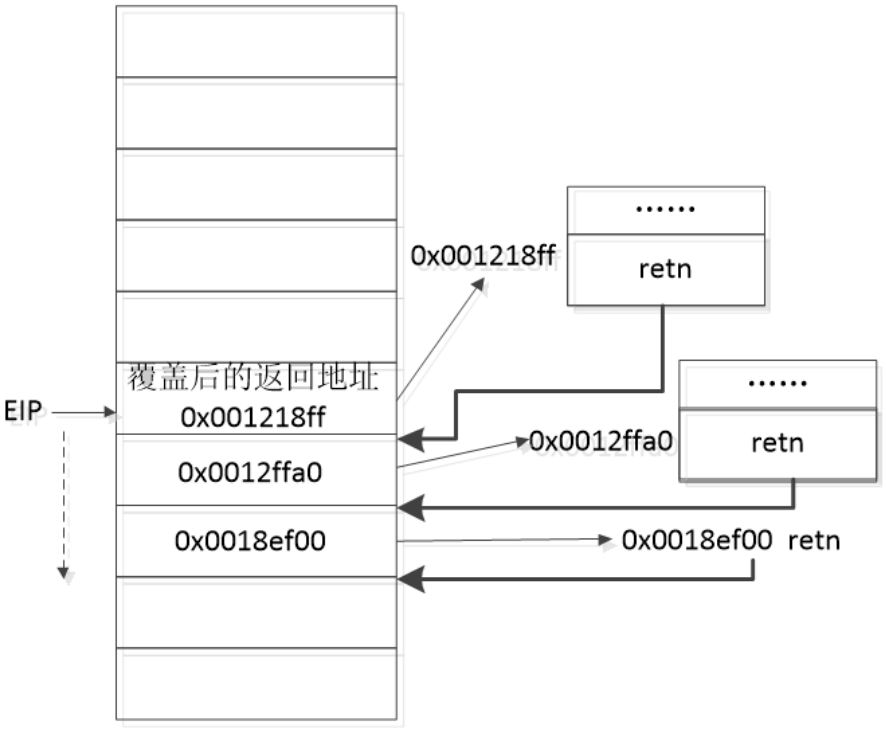
基本思想：借助已存在的代码块（也叫配件，Gadget），这些配件来自程序已经加载的模块。
在已加载的模块中找到一些以 retn 结尾的配件，把这些配件的地址布置在堆栈上，当控制EIP并返回的时候，程序就会跳去执行这些小配件，
这些小配件是在别的模块代码段, 不受 DEP 的影响。

总结：

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。

示例图：



核心理解：retn = pop eip，会链式执行配件。

实例：

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

通过上面的ROP指令段，我们实现了将EDX置0的结果。

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（个人理解：也是 retn），完成漏洞利用。

绕过其它安全防护

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。

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。

其中，David Litchfield 在 2003 年提出了一种技术，通过利用异常处理机制 SEH 来绕过 GS 保护：

> 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。GS 安全机制的一个主要缺陷是没有保护异常处理器。尽管系统提供了 SEH 保护机制，但它也可能被攻击者绕过，导致安全漏洞的利用。

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：

- 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，并且**随机化发生在地址前两个最有意义的字节上**。
- 很多应用程序和 DLL 模块**并没有采用 /DYNAMICBASE 的编译选项**。
- 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。

SEH 保护机制缺陷和绕过方法

当一个进程中存在一个未使用 /SafeSEH 编译的 DLL 或库文件时，整个 SafeSEH 机制就可能失效。因为 /SafeSEH 编译选项需要 .NET 编译器的支持，目前仍有许多第三方库和程序没有使用该编译器编译或没有启用 /SafeSEH 选项。

目前，一些可行的绕过 SafeSEH 机制的方法包括：

1. **利用未启用 SafeSEH 的模块作为跳板绕过**：可以在未启用 SafeSEH 的模块中找到一些跳转指令，并覆盖 SEH 函数指针。由于这些指令在未启用 SafeSEH 的模块中存在，当异常触发时，可以执行这些指令。
2. **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特定内存绕过。具体的方法不展开介绍。