第四章 软件漏洞

1 溢出漏洞基本概念

漏洞也称为**脆弱性（Vulnerability）**

缓冲区溢出漏洞：就是说，往系统中写入了超容量的数据，导致多余的数据将相邻的内存空间给覆盖了，导致了溢出

我们可以利用这个覆盖来完成缓冲区溢出攻击，就是去覆盖掉相邻内存空间的一些重要数据

为什么会产生缓冲区溢出？就是因为我们对自己程序的边界都没有进行检查，这样就会引发一系列的问题

缓冲区溢出通常包括**栈溢出、堆溢出、异常处理SEH结构溢出、单字节溢出**等

2 栈溢出漏洞

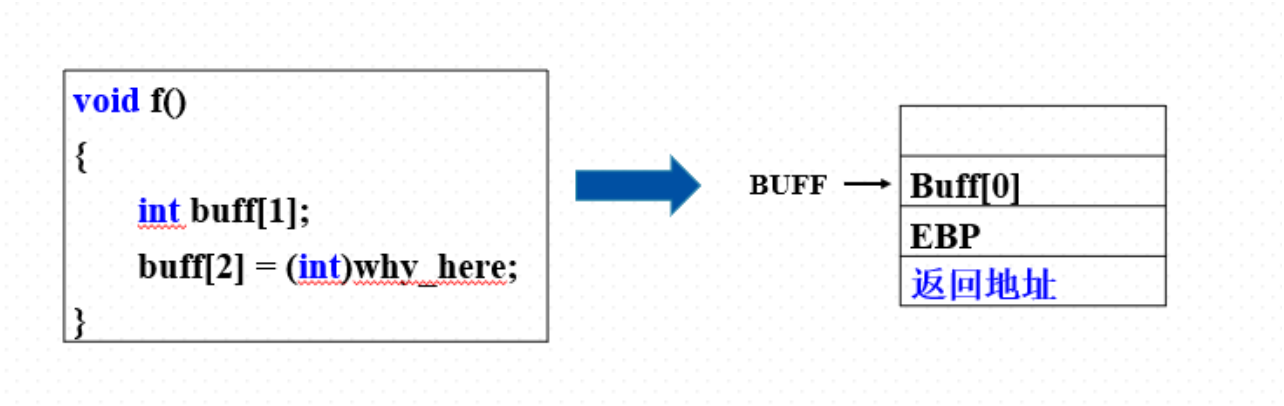
被调用的子函数中写入数据的长度，大于栈帧的基址到esp之间预留的保存局部变量的空间时，就会发生栈的溢出。要写入数据的填充方向是**从低地址向高地址增长**，多余的数据就会越过栈帧的基址，覆盖基址以上的地址空间

```
void why_here(void)
{    printf("why u r here?!\n");
      exit(0);
}
void f()
{     int buff[1];
       buff[2] = (int)why_here;
}
int main(int argc, char * argv[])
{      f();
        return 0;
}
```

在函数f中，所声明的数组buff长度为1，但是由于没有对访问下标的值的校验，程序中对数组外的内存进行了读写。

我们观察一下函数f的局部变量buff的内存示意。Buff是静态数组，buff的值就是数组在内存的首地址。而inf buff[1]意味着开辟了一个四字节的整数数组的空间。如图所示（动画）。

函数的栈区中，局部变量区域存的是数组元素buff[0]的值。而Buff[2]则指向了返回地址。而Buff[2]赋值为why_here，意味着返回地址被写入了4字节的函数why_here的地址。这样，在函数f执行完毕恢复到主函数main继续运行时，因为返回地址被改写成了why_here函数的地址，而覆盖了原来的主函数main的下一条指令的地址，因此，发生了执行跳转。



应该怎么修改呢?

我们可以将数组改为指针，这样就不会出现数组溢出的情况了

```
*(p+2)或者p[2]= (int)why_here;
```

溢出之后，我们可以通过修改返回地址来使程序失效，或者转向执行恶意程序；也可以覆盖临近的变量，来达到更改程序的执行流程

3 堆溢出漏洞

堆溢出后，数据可以覆盖堆区的不同堆块的数据，带来安全威胁

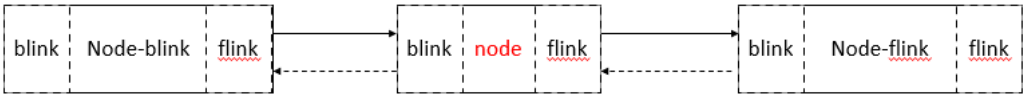
比如说，申请了两个堆，中间的内存距离是72字节，那么如果在第一个堆的输入中输入超过72字节，这样就会发生堆溢出，多出来的那些内容就被填充到了第二个堆块中

堆溢出的危害远远大于栈溢出，堆溢出可以在程序内存的任意位置写入数据

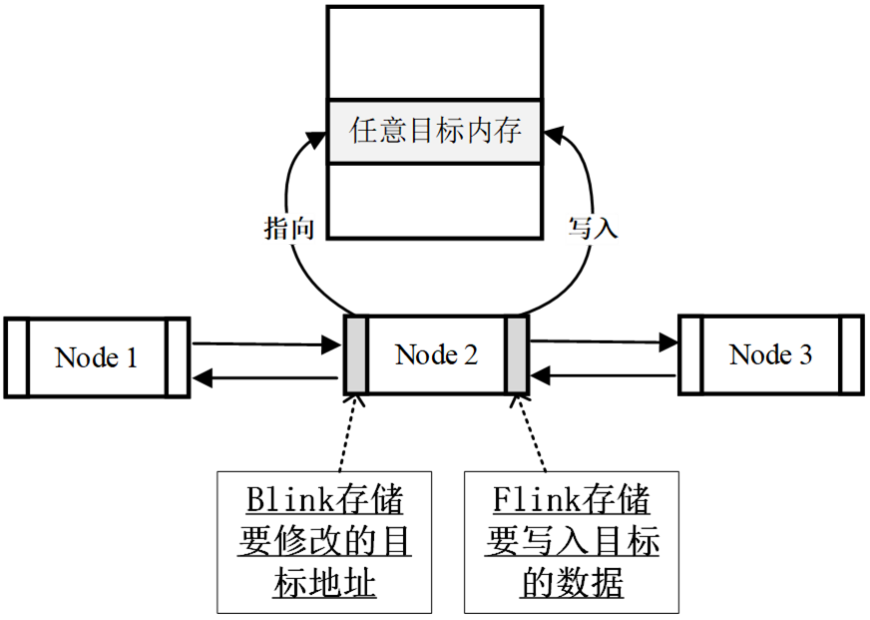
4 Dword Shoot攻击

从链表上卸载（unlink）一个节点的时候会发生如下操作：

```
node->blink->flink = node->flink ;
node->flink->blink = node->blink ;
```



如果我们通过堆溢出覆写了一个空闲堆块的块首的前向指针flink和后向指针blink，我们可以精心构造一个地址和一个数据，当这个空闲堆块从链表里卸下的时候，就获得一次向内存构造的任意地址写入一个任意数据的机会。



注意，**前向指针写数据，后向指针写地址**

具体操作可见实验部分Lab3

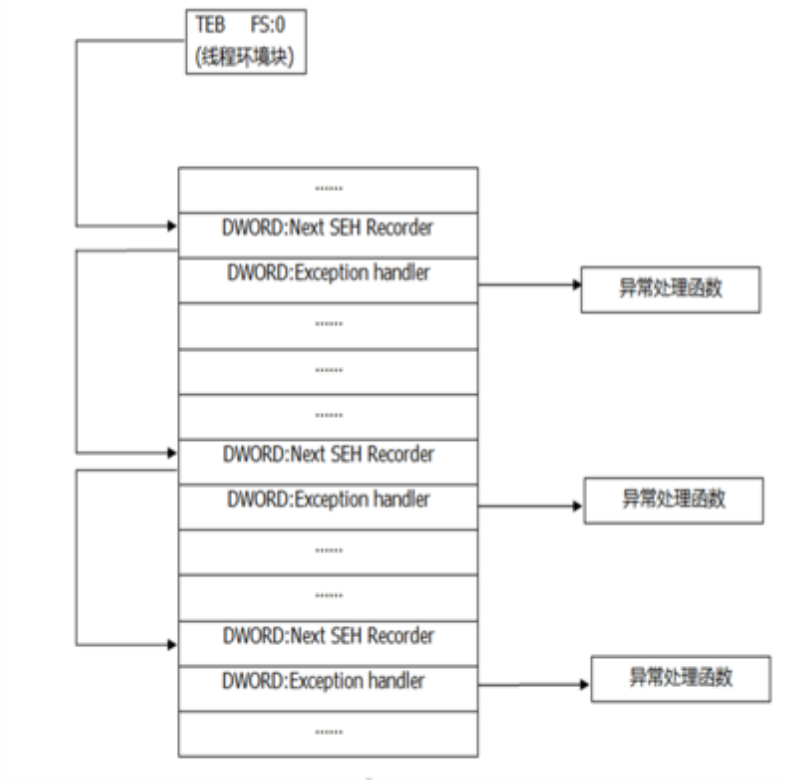
5 其他溢出漏洞

5.1 SEH结构溢出

为了保证系统在遇到错误时不至于崩溃，仍能够健壮稳定地继续运行下去，Windows会对运行在其中的程序提供一次补救的机会来处理错误，这种机制就是异常处理机制。

异常处理结构体SEH是Windows异常处理机制所采用的重要数据结构:

- SHE结构体存放在栈中，栈中的多个SEH通过链表指针在栈内由栈顶向栈底串成单向链表;
- 位于链表最顶端的SEH通过线程环境块（TEB，Thread Environment Block）0字节偏移处的指针标识;
- 每个SEH包含两个DWORD指针：SEH链表指针和异常处理函数句柄，共8个字节。



SEH结构用作异常处理，主要包括如下三个方面：

- 当线程初始化时，会自动向栈中安装一个SEH，作为线程默认的异常处理。如果程序源代码中使用了`_try{}_except{}`或者Assert宏等异常处理机制，编译器将最终通过向当前函数栈帧中安装一个SEH来实现异常处理。
- 当异常发生时，操作系统会**中断程序**，并首先从TEB的0字节偏移处取出距离栈顶最近的SEH，使用异常处理函数句柄所指向的代码来处理异常。当最近的异常处理函数运行失败时，将顺着SEH链表依次尝试其他的异常处理函数。
- 如果程序安装的所有异常处理函数都不能处理这个异常，系统会调用默认的系统处理程序，通常显示一个对话框，你可以选择关闭或者最后将其附加到调试器上的调试按钮。如果没有调试器能被附加于其上或者调试器也处理不了，系统就调用ExitProcess终结程序。

SHE攻击：指通过栈溢出或者其他漏洞，使用精心构造的数据覆盖SEH链表的入口地址、异常处理函数句柄或链表指针等，实现程序执行流程的控制

5.2 单字节溢出

就是程序的缓冲区只能溢出一个字节

```
void single_func(char *src){
char buf[256];
int i;
for(i = 0;i <= 256;i++)
         buf[i] = src[i];
}
```

使用条件：它溢出的一个字节必须与栈帧指针紧挨，就是要求必须是**函数中首个变量**，一般这种情况很难出现

6 格式化字符串漏洞

printf()函数的一般形式为：printf(“format”, 输出表列)， format的结构为：`%[标志][输出最小宽度][.精度][长度]类型`

常见的有以下几种：

- %d整型输出
- %ld长整型输出
- %o以八进制数形式输出整数
- %x以十六进制数形式输出整数
- %u以十进制数输出unsigned型数据(无符号数)
- %c用来输出一个字符
- %s用来输出一个字符串
- %f用来输出实数，以小数形式输出

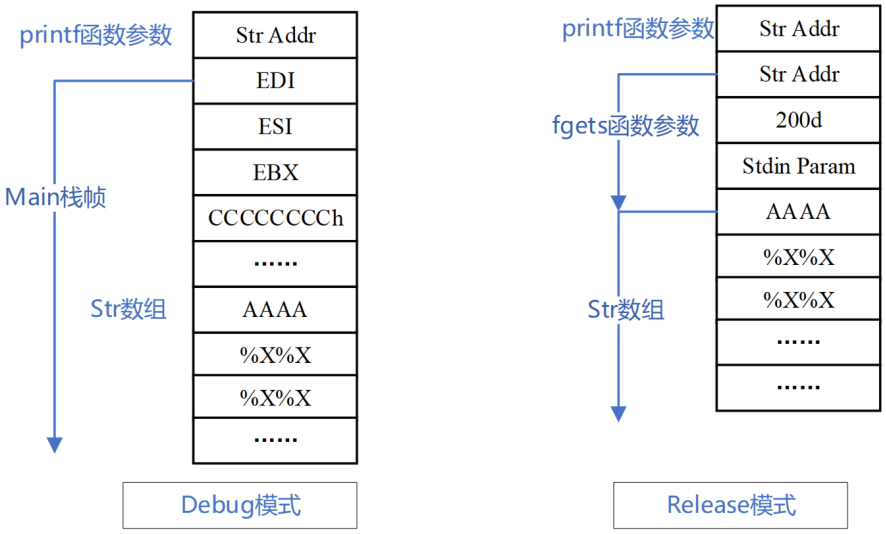
格式化函数允许可变参数，根据传入的格式化字符串获知可变参数的个数和类型，并依据格式化符号进行参数的输出，如果调用这些函数时，给出了格式化符号串，但没有提供实际对应参数时，这些函数会将格式化字符串**后面的多个栈中的内容**取出作为参数，并根据格式化符号将其输出

举个例子，下面的程序：

```
void formatstring_func1(char *buf)
{
     char mark[] = “ABCD”;
     printf(buf);
}
```

调用时如果传入”%x%x…%x”，则printf会**打印出堆栈中的内容**，不断增加%x的个数会逐渐显示堆栈中高地址的数据，从而导致堆栈中的数据泄漏。

对比debug模式和release模式的栈帧结构的不同：



我们还可以实现数据的写入，利用%n来写入数据，将格式化函数输出字符串的长度，写入函数参数指定的位置

还可以利用%n格式化符号和自定义打印字符串宽度，写入某内存地址任意数据

7 整数溢出漏洞

分为以下三类：

- **存储溢出**：存储溢出是使用另外的数据类型来存储整型数造成的。例如，把一个大的变量放入一个小变量的存储区域，最终是只能保留小变量能够存储的位，其他的位都无法存储，以至于造成安全隐患
- **运算溢出**：运算溢出是对整型变量进行运算时没有考虑到其边界范围，造成运算后的数值范围超出了其存储空间
- **符号溢出**：整型数可分为有符号整型数和无符号整型数两种。在开发过程中，一般长度变量使用无符号整型数，然而如果程序员忽略了符号，在进行安全检查判断的时候就可能出现问题

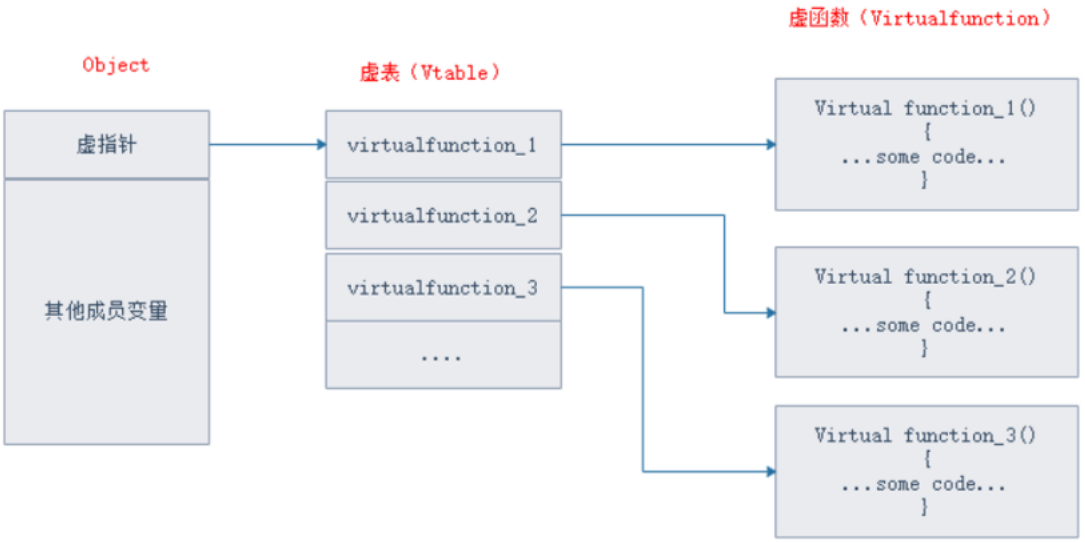
举个例子：

```
char* integer_overflow(int* data,
unsigned int len){
    unsigned int size = len + 1;
    char *buffer = (char*)malloc(size);
    if(!buffer)
                    return NULL;
    memcpy(buffer, data, len);
    buffer[len]='\';
      return buffer;
}
```

该函数将用户输入的数据拷贝到新的缓冲区，并在最后写入结尾符0。如果攻击者将0xFFFFFFFF作为参数传入len，当计算size时会发生整数溢出，malloc会分配大小为0的内存块（将得到有效地址），后面执行memcpy时会发生堆溢出

8 攻击C++虚函数

- 多态是面向对象的一个重要特性，在C++中，这个特性主要靠对虚函数的动态调用来实现。
- C++类的成员函数声明时，若使用关键字virtual进行修饰，则被称为虚函数。
- **虚函数的入口地址被统一保存在虚表（Vtable）中。**
- **对象在使用虚函数时，先通过虚表指针找到虚表，然后从虚表中取出最终的函数入口地址进行调用**



C++虚函数和类在内存中的位置关系如图所示：

- **虚表指针保存在对象的内存空间中，紧接着虚表指针的是其他成员变量；**
- **虚函数入口地址被统一存在虚表中**

一般来说，使用虚函数，需要通过调用虚表指针找到虚表，再从虚表中取出最终的入口地址进行调用，如果修改虚表里存储的虚函数指针被修改，就会发动虚函数攻击

攻击策略：

充分利用overflow.buf这个缓冲区：

- 修改虚表地址：将对象overflow的虚表地址修改为数组shellcode的倒数第四个字节开始地址。
- 修改虚函数指针：修改数组shellcode最后4位（虚表）来指向overflow.buf的内存地址，即让虚函数指针指向保存shellcode的overflow.buf区域。

9 其他类型漏洞

9.1 注入类漏洞

- 二进制代码注入
- 脚本注入

9.1.1 SQL注入

SQL注入是将Web页面的原URL、表单域或数据包输入的参数，修改拼接成SQL语句，传递给Web服务器，进而传给数据库服务器以执行数据库命令

9.1.2 操作系统命令注入

操作系统命令注入攻击（OS Command Injection）是指通过Web应用，执行非法的操作系统命令达到攻击的目的。大多数Web服务器都能够使用内置的API与服务器的操作系统进行几乎任何必需的交互，比如PHP中的system、exec和ASP中的wscript类函数

9.1.3 Web脚本语言注入

常用的ASP/PHP/JSP等web脚本解释语言支持动态执行在运行时生成的代码这种特点，可以帮助开发者根据各种数据和条件动态修改程序代码，这对于开发人员来说是有利的，但这也隐藏着巨大的风险。这种类型的漏洞主要来自两个方面：

- **合并了用户提交数据的代码的动态执行。**攻击者通过提交精心设计输入，使得合并用户提交数据后的代码蕴含设定的非正常业务逻辑来实施特定攻击。
- **根据用户提交的数据指定的代码文件的动态包含。**多数脚本语言都支持使用包含文件（include file），这种功能允许开发者把可重复使用的代码插入到单个文件中，在需要的时候再将它们包含到相关代码文件中。如果攻击者能修改这个文件中的代码，就让受此攻击的应用执行攻击者的代码。

9.1.4 SOAP注入

基于XML的协议，让应用程序跨HTTP进行信息交换

如果用户提交的数据中包含这些字符，并被直接插入到SOAP消息中，攻击者就能够破坏消息的结构，进而破坏应用程序的逻辑或造成其他不利影响

9.2 权限类漏洞

- **水平越权**：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。水平权限漏洞一般出现在一个用户对象关联多个其他对象（个人资料、修改密码，订单信息，等）、并且要实现对关联对象的CURD的时候
- **垂直越权**
  - 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**