第四章　软件漏洞

知识点五：格式化字符串漏洞
知识点六：整数溢出漏洞
知识点七：攻击C++虚函数
知识点八：其他类型漏洞

知识点五：格式化字符串漏洞

1. 格式化字符串定义

格式化串漏洞和普通的栈溢出有相似之处，但又有所不同，都是利用了程序员的疏忽大意来改变程序运行的正常流程。

首先，**什么是格式化字符串呢**，print()、fprint()等*print()系列的函数可以按照一定的格式将数据进行输出，举个最简单的例子：

printf("My Name is: %s" , "bingtangguan")

执行该函数后将返回字符串：My Name is：bingtangguan

该printf函数的**第一个参数就是格式化字符串，它来告诉程序将数据以什么格式输出。**

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**

其中类型有以下常见的几种：
%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。

控制format参数之后结合**printf()函数特性**就可以进行相应攻击。

2. 格式化字符串漏洞的利用—数据泄露

特性一：格式化函数允许可变参数

C语言中的**格式化函数**（*printf族函数，包括printf，fprintf，sprintf，snprintf等）**允许可变参数**，它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。

当格式化符号为%x时以16进制的形式输出堆栈的内容，为%s时则输出对应地址所指向的字符串。

下面以下述程序样本为例，分析格式化字符串溢出的原理。

```
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。

泄露内存数据

```
#include <stdio.h>
int main(void)
{
int a=1,b=2,c=3;
char buf[]="test";
printf("%s %d %d %d\n",buf,a,b,c);
return 0;
}
```

编译之后运行（Debug模式）： test 1 2 3

增加一个printf()的format参数，改为：

printf("%s %d %d %d %x\n",buf,a,b,c),

编译后运行（Debug模式）：

test 1 2 3 12C62E

为什么输出了一个12C62E？

原因：函数调用，是要参数入栈的；printf函数会到入栈的参数位置去取参数；在没有给出%x的参数的时候，将自动将栈区参数的下一个地址作为参数输入。

读取任意内存地址的数据

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    char str[200];
    fgets(str,200,stdin);
    printf(str);
    return 0;
}
```

编译后运行（Release模式）并输入：AAAA%x%x%x%x

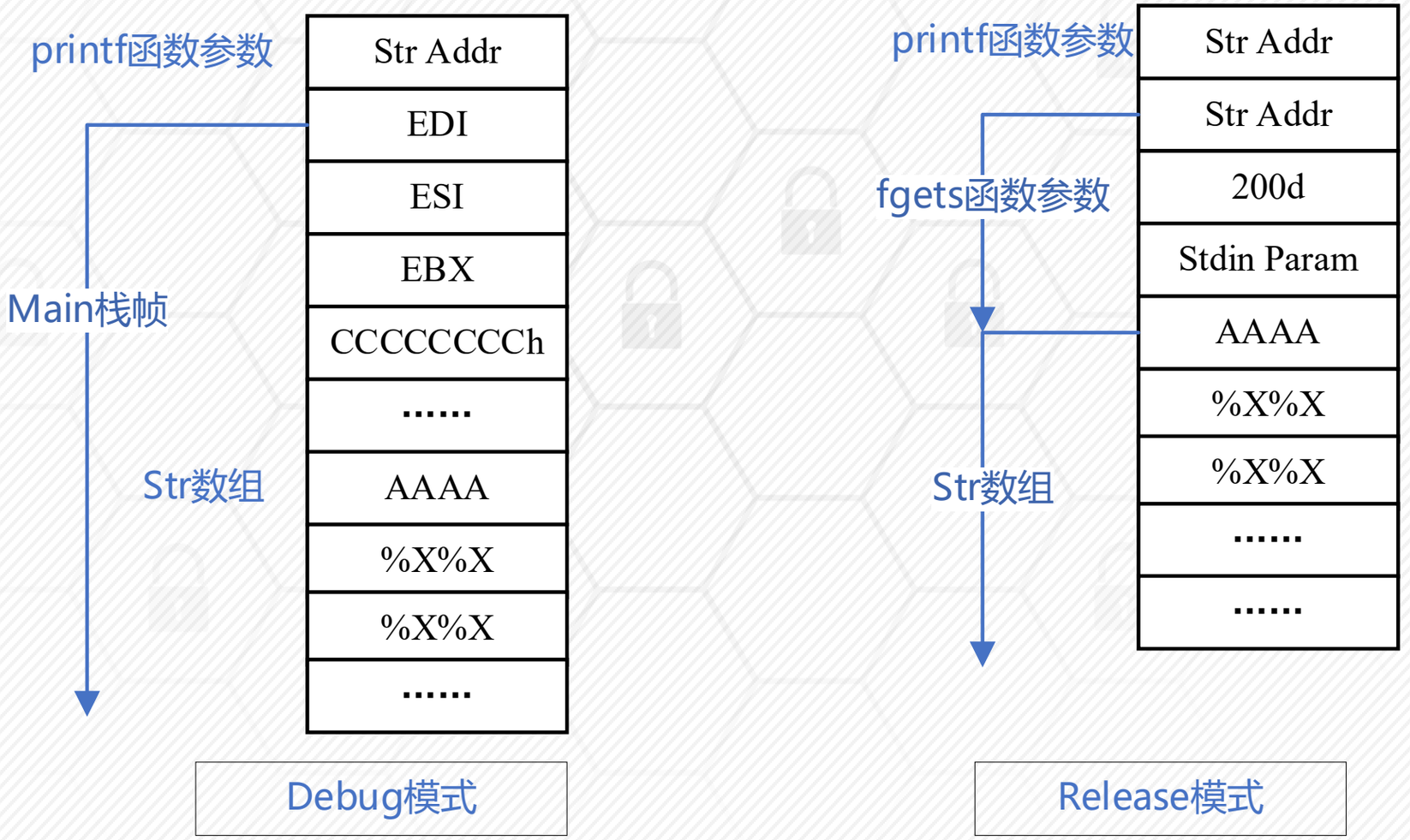
我们成功读到了AAAA：AAAA18FE84BB40603041414141

（0x41就是ASCII的字母A的值）。

思考：这个41414141是怎么读到的？

读取任意内存地址的数据

执行printf(str)语句的时候，对比Debug模式和Relase模式的栈帧结构：



3. 格式化字符串漏洞的利用—数据写入

 # 特性二：利用%n格式符写入数据

更危险的是格式化符号%n，它的作用是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。

%n不向printf传递格式化信息，而是令printf把自己到该点已打出的字符总数放到相应变元指向的整形变量中，比如

printf(“Jamsa%n”, &first_count)

将向整型变量first_count处写入整数5。

Sprintf函数的作用是**把格式化的数据写入某个字符串缓冲区**。函数原型为：

int sprintf(char *buffer, const char *format, [argument] …);

观察如下程序（Release模式）：

```
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，将会怎么样?

结果：数值10就会被写入地址为0x61616161（aaaa）的内存单元。

首先

“aaaabbbbcc”写入buffer；

然后

从堆栈中取下一个参数，**并将其当作整数指针使用**，由于调用sprintf时没有传入下一个参数，**因而buffer中的前四个字节被当作参数**，这样已输出字串的长度10就被写入内存地址0x61616161处。

通过这种格式化字符串的利用方式，可以实现向任意内存写入任意数值。

特性三：自定义打印字符串宽度

实验：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

```
#include <stdio.h>
main()
{
  int num=66666666;
  printf("Before: num = %d\n", num);
  printf("%d%n\n", num, &num);
  printf("After: num = %d\n", num);
}
```

运行：

```
Before: num = 66666666
66666666
After: num = 8
```

现在我们已经知道可以利用%n向内存中写入值，如果我们写的值(比如一个返回地址）非常大，怎么来构造这样的值?

关于打印字符串宽度的问题，在**格式符中间加上一个十进制整数来表示输出的最少位数**，若实际位数多于定义的宽度，则按实际位数输出，若实际位数少于定义的宽度则补以空格或0。我们把上一段代码做一下修改并看一下效果：

```
#include <stdio.h>
main()
{
  int num=66666666;
  printf("Before: num = %d\n", num);
  printf("%100d%n\n", num, &num);
  printf("After: num = %d\n", num);
}
```

运行：

```
Before: num = 66666666

                                                                                                    66666666
After: num = 100
```

我们也可以使用%02333d这种形式。在打印数值右侧用0补齐不足位数的方式来补齐，而不是空格。

知识点六：整数溢出漏洞

整数溢出

高级程序语言中，整数分为无符号数和有符号数两类，其中有符号负整数最高位为1，正整数最高位为0，无符号整数则无此限制。常见的整数类型有8位、16位、32位以及64位等，对应的每种类型整数都包含一定的范围。当对整数进行加、乘等运算时，计算的结果如果大于该类型的整数所表示的范围时，就会发生整数溢出。

根据溢出原理的不同，整数溢出可以分为以下三类：

(1) 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。

(2) 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。

(3) 符号问题

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。**

整数溢出的样例可通过下面的代码了解。

```
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。

整数溢出一般不能被单独利用，而是用来绕过目标程序中的条件检测，进而实现其他攻击。

分析如下实例：

```
#include<iostream>
#include<windows.h>
#include<shellapi.h>
#include<stdio.h>
#include<stdlib.h>
#define MAX_INFO 32767
using namespace std;
void func()
{
ShellExecute(NULL,"open","notepad",NULL,NU
LL,SW_SHOW);
//打开记事本
}
void func1()
{
ShellExecute(NULL,"open","calc",NULL,NULL,
SW_SHOW);
//打开计算器
}
```

```
int main()
{
        void (*fuc_ptr)() = func;
        char info[MAX_INFO];
        char info1[30000];  char info2[30000];

        freopen("input.txt","r",stdin);
        cin.getline(info1,30000,' ');
        cin.getline(info2,30000,' ');

        short len1 = strlen(info1);
        short len2 = strlen(info2);
        short all_len = len1 + len2;

        if(all_len<MAX_INFO)
        {
                strcpy(info,info1);
                strcat(info,info2);
        }
        fuc_ptr();
        return 0;
}
```

short型整数表示范围为-32768~32767，当len1+len2超过了short型整数的最大范围后会变为一个负数，将满足all_len<MAX_INFO的判断条件，进而进入if的分支语句。于是继续执行if语句的时候，将info1与info2的内容都写进info中。

思考：如何实现fuc_ptr的覆盖，改变程序执行?

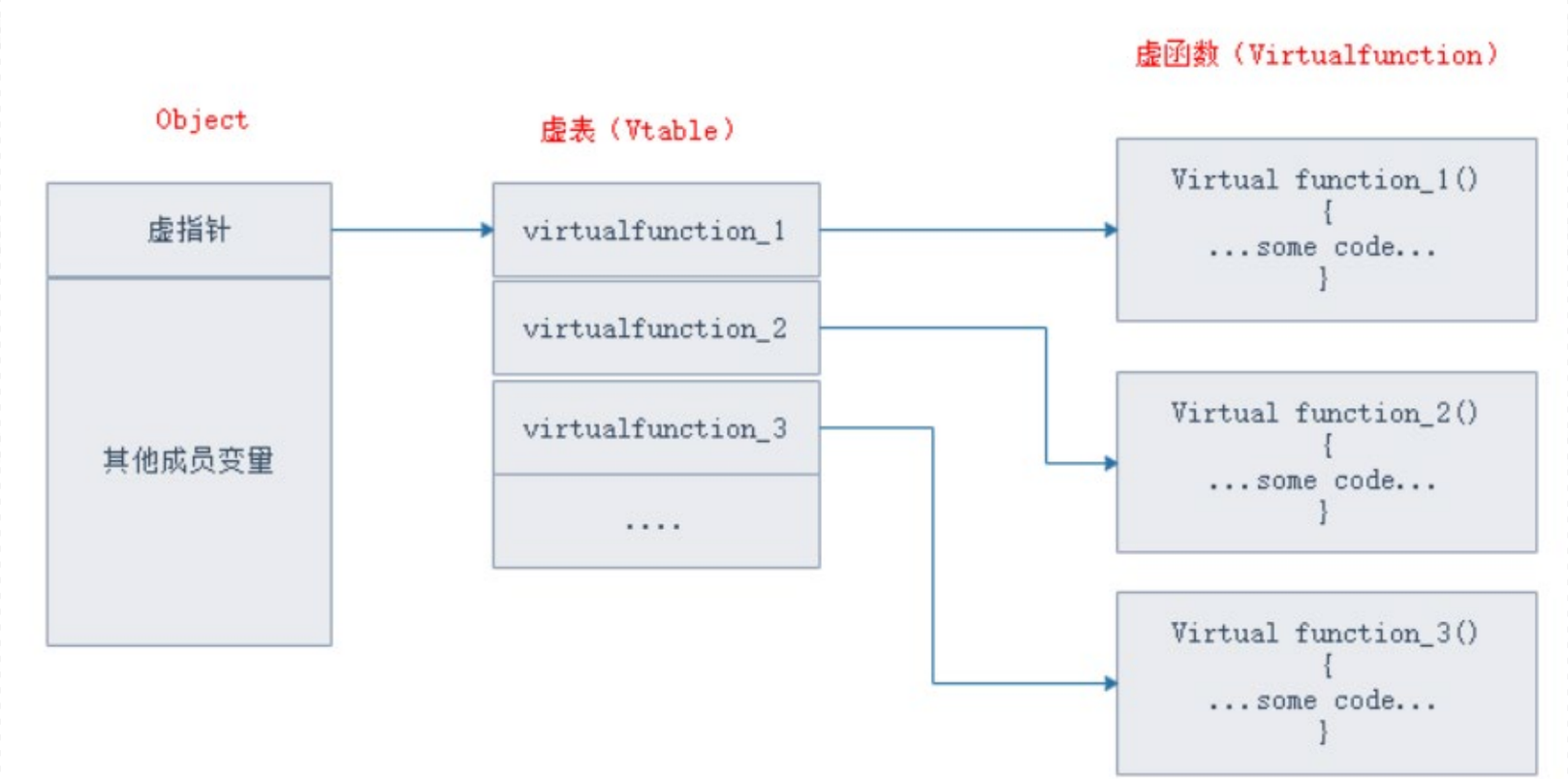
知识点七：攻击C++虚函数

C++面向对象语言的漏洞

- 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++面向对象语言的漏洞

C++虚函数和类在内存中的位置关系如图所示：（1）**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；（2）虚函数入口地址被统一存在虚表中。**



攻击虚函数

对象使用虚函数时通过（1）调用虚表指针找到虚表，然后（2）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。

通过下述代码来复现虚函数攻击。

```
char shellcode[] =“xFC\x68\x6A….. \xA4\x8B\x42\x00”;
class Failwest{
public:
   char buf[200];
   virtual void test(void)
   {
      cout<<"Class Vtable::test()"<<endl;
   };
};
Failwest overflow, *p;
void main(void){
   char *p_vtable;
   p_vtable = overflow.buf – 4;
   int len = strlen(shellcode);
   __asm int 3;  //人为增加一个断点
   p_vtable[0] = 0x54;
   p_vtable[1] = 0x8c;
   p_vtable[2] = 0x42;
   p_vtable[3] = 0x00;
   strcpy(overflow.buf, shellcode);
   p = &overflow;
   p->test();
}
```

得到虚表指针：为虚表指针位于对象overflow成员变量char buf[200]之前，程序中通过p_vtable=overflow.buf-4定位到这个指针。

这是我们能利用的缓冲区，这意味着，恶意代码shellcode被存储到了overflow.buf位置。

我们**希望通过调用test虚函数的时候，跳转到这个位置去执行恶意代码**。但是，怎么让调用test虚函数的时候，通过虚表指针找到的虚函数指针就是我们期待的目标呢？

```
char shellcode[] =“xFC\x68\x6A....\xA4\x8B\x42\x00”;
class Failwest{
public:
    char buf[200];
    virtual void test(void)
    {
        cout<<"Class Vtable::test()"<<endl;
    };
};
Failwest overflow, *p;
void main(void){
    char *p_vtable;
    p_vtable = overflow.buf - 4;
    ……

    p_vtable[0] = 0x54;
    p_vtable[1] = 0x8c;
    p_vtable[2] = 0x42;
    p_vtable[3] = 0x00;

    …….
}
```

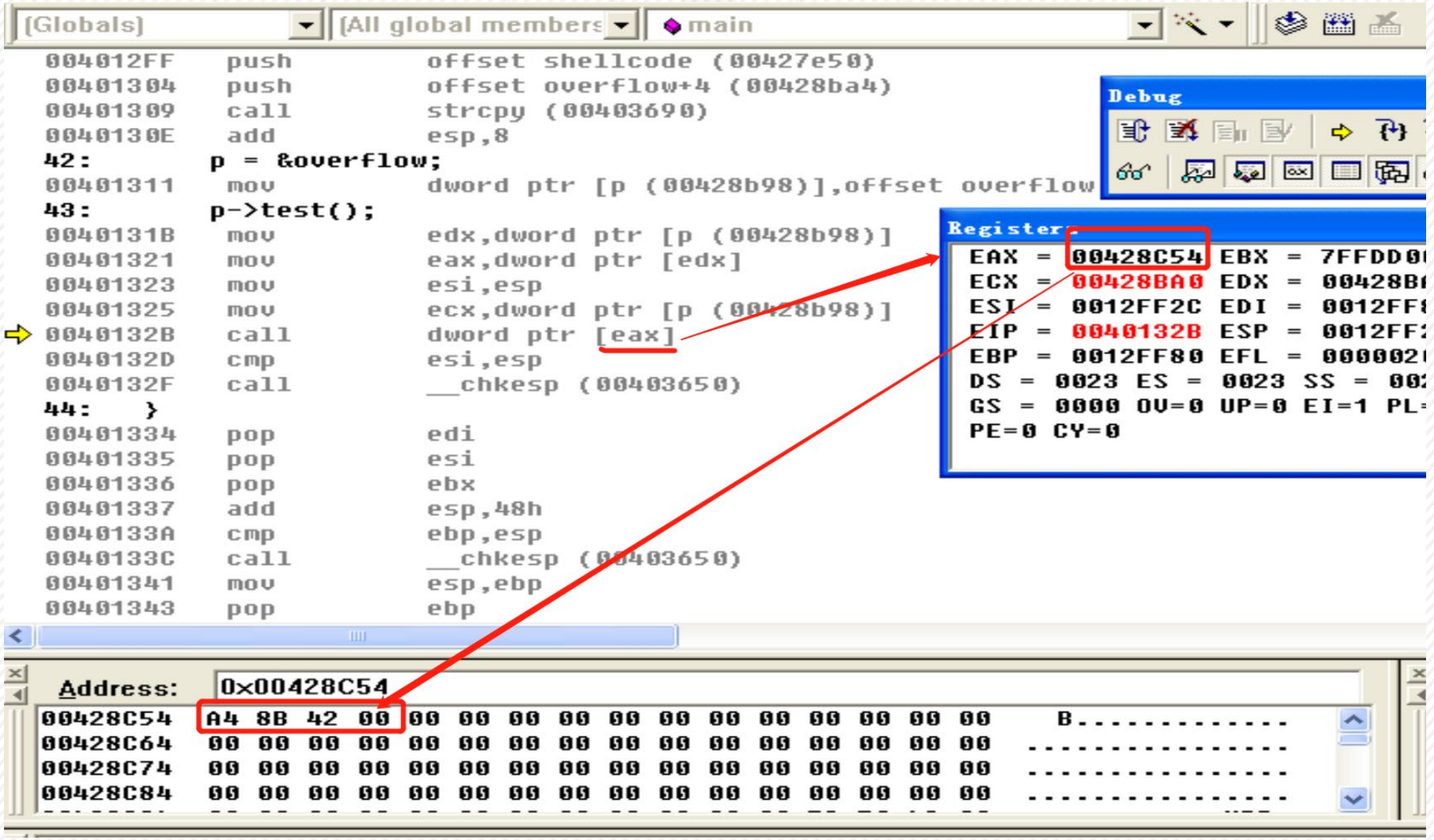
攻击策略

充分利用overflow.buf这个缓冲区：overflow.buf的地址为0x00428ba4，其倒数第四个字节开始地址为0x00428c54。Strlen里最后一个字符是0x00，需要加上。

1. 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。

2. 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。

VC IDE，进行实际调试的时候，在语句“p->test();”处转入反汇编，继续单步调试，可以看到攻击成功，弹出failwest的对话框：



知识点八：其他类型漏洞

1. 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。

（1）SQL注入

下面介绍几种Web场景下的代码注入攻击。

SQL（Structured query language，结构化查询语言）是操作数据库数据的结构化查询语言，用于读取、更新、增加或删除数据库中保存的信息。应用程序通过SQL语言来完成后台数据库中的数据的增加、删除、修改和查询。

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。

如果Web应用程序的开发人员对用户所输入的数据不进行过滤或验证就直接传输给数据库，就可能导致拼接的异常SQL语句被执行，获取对数据库的信息以及提权，发生SQL注入攻击。

（1）SQL注入


```
strKeyword = Request[“keyword”];
sqlQuery = “SELECT * FROM Articles WHERE Keywords LIKE ‘%” +strKeyword+ “%’ ”;
```

Hack’; DROP TABLE Aritcles; --

SELECT * FROM Aritcles WHERE Keywords LIKE ‘%**hack’ ; DROP TABLE Aritcles; --**%’

--是注释符，**结果是以中间的分号为标志分成两个部分**，执行完 “SELECT * FROM Aritcles WHERE Keywords LIKE ‘%hack’ ” 后，将执行” DROP TABLE Aritcles;”

（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。**如果正确使用，这些API可以丰富Web应用的功能。但是，如果应用程序向操作系统命令程序传送用户提交的输入，而且没有对输入进行过滤和检测，就可能遭受命令注入攻击。

许多定制和非定制web应用程序中都存在这种命令注入缺陷。在为企业服务器或防火墙、打印机和路由器之类的设备提供管理界面的应用程序中，这类缺陷尤其普遍。

（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：

（1）**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。

（2）**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。

（4）SOAP注入

SOAP（Simple Object Access Protocol，简单对象访问协议），是一个简单的基于XML的协议，它让应用程序跨HTTP进行信息交换。它主要用在Web服务中，**通过浏览器访问的Web应用程序常常使用SOAP在后端应用程序组件之间进行通信。**

由于**XML也是一种解释型语言**，因此SOAP也易于遭受代码注入攻击。XML元素通过元字符<>和/以语法形式表示。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。

2. 权限类漏洞

绝大多数系统，都具备基于用户角色的访问控制功能，根据不同用户对其权限加以区分。但攻击者为了访问受限资源或使用额外功能，会利用系统存在的缺陷或漏洞，进行自身角色的权限提升或权限扩展。

权限越权又可以分为两种：**水平越权与垂直越权**。

2. 权限类漏洞

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。如果出现此类漏洞，那么将可能会造成大批量数据泄露，严重的甚至会造成用户信息被恶意篡改。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。比如，当web应用程序接收到用户请求时，没有判断数据的所属人，或者在判断数据所属人时是从用户提交的参数中获取了userid，导致攻击者可以自行修改userid修改不属于自己的数据。

2. 权限类漏洞

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。

比如，在web应用中，如果后台应用没有做权限控制，或仅仅在菜单、按钮上做了权限控制，导致恶意用户只要猜测其他管理页面的URL或者敏感的参数信息，就可以访问或控制其他角色拥有的数据或页面，达到权限提升的目的。