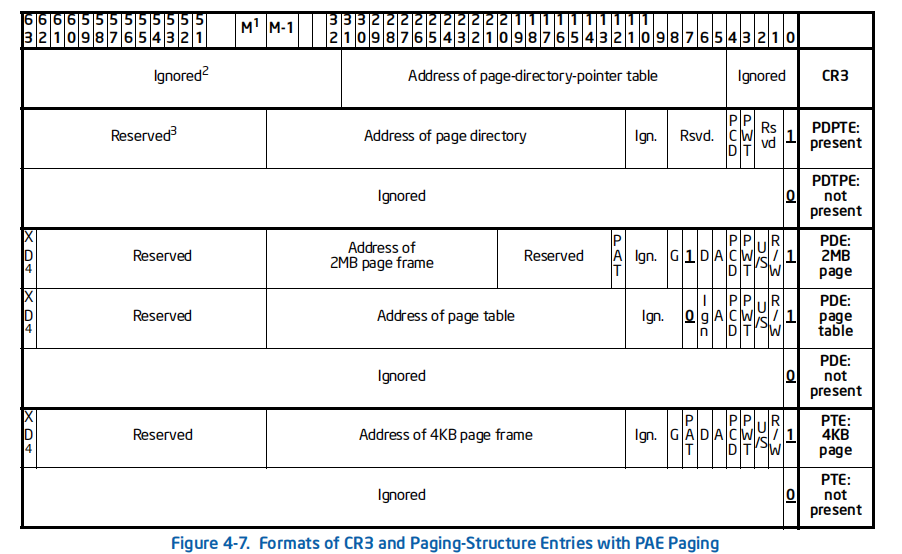
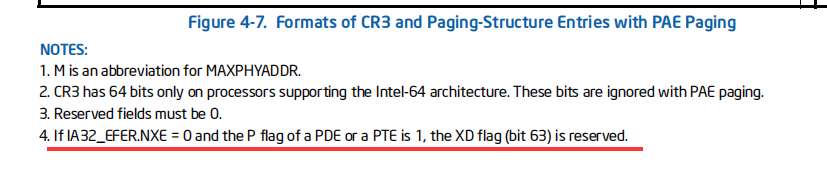
# 课堂笔记

### 1.XD/NX位实验



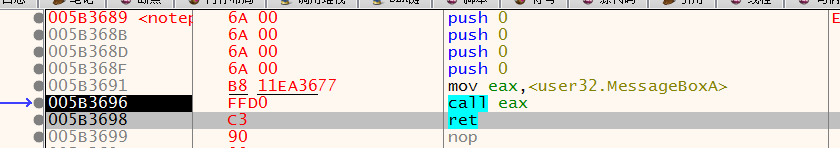
2-9-9-12分页提供了数据执行保护DEP机制，PDE和PTE的最高位是**XD位，又叫NX位**，置1表示不可执行。PDE.XD=1 表示这个页目录项指向的页表对应的所有物理页（最多512个）全部不可以执行；PTE.XD=1表示这个页表项映射的单个物理页不可执行。



不过凡事都可能有例外，如果IA32\_EFER.NXE = 0，那么XD位就没用了。这个位是MSR寄存器的第11位，用来启用DEP的。

一般是由PTE来控制XD，PDE.XD一般都设置为0，表示可执行，因为PDE控制范围太大了。

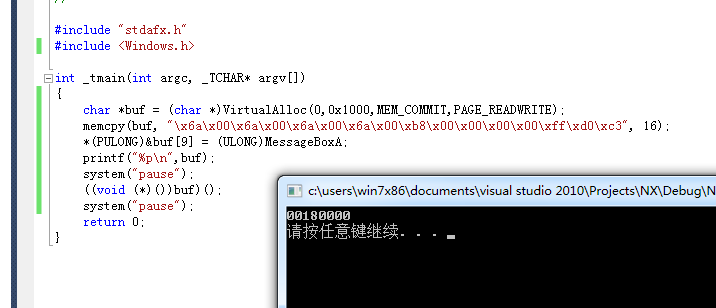
下面做NX位的实验，注意windbg改完NX位后，g放行后要多等一会，等TLB刷新。



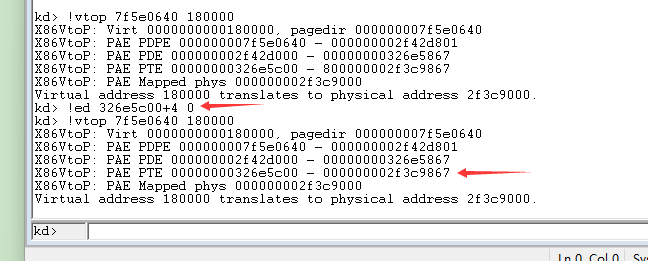
写一段shellcode

代码如下：

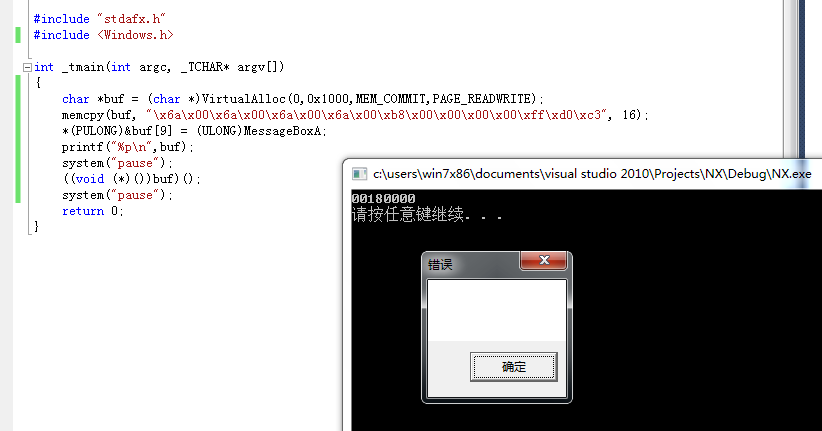
|  |
| --- |
| #include "stdafx.h"  #include <Windows.h>  int \_tmain(int argc, \_TCHAR\* argv[])  {  char \*buf = (char \*)VirtualAlloc(0,0x1000,MEM\_COMMIT,PAGE\_READWRITE);  memcpy(buf, "\x6a\x00\x6a\x00\x6a\x00\x6a\x00\xb8\x00\x00\x00\x00\xff\xd0\xc3", 16);  \*(PULONG)&buf[9] = (ULONG)MessageBoxA;  printf("%p\n",buf);  system("pause");  ((void (\*)())buf)();  system("pause");  return 0;  } |



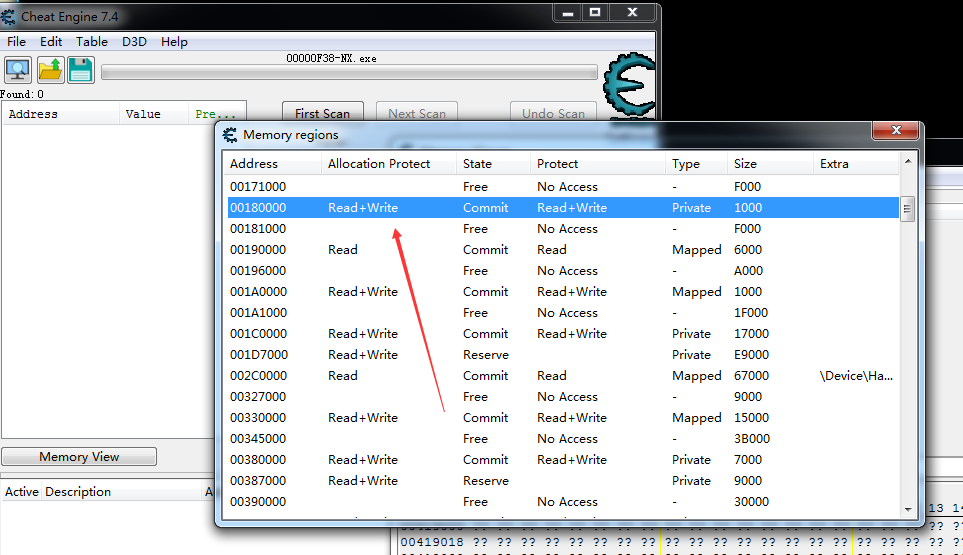
运行，在调用shellcode前断下，打印一下shellcode地址，接下来改shellcode PTE.XD=0



改PTE.XD=0



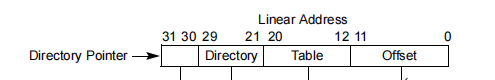
可执行了，msgbox成功弹出，并且此时这个地址的页属性依然是不可执行：

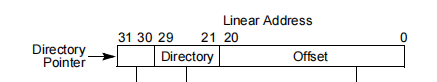


### TLB简介

每个核都有一套TLB。TLB的结构未公开，但大概长这样子：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 虚拟页帧 | 物理页帧 | Attribute属性 | 访问次数 | 可能还有类似64位PCID的字段？ |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |





所谓虚拟页帧，小页就是去掉低12位Offset，大页就是去掉低21位Offset

物理页帧应该是这个虚拟地址对应的物理页基址。

对线性地址寻址时，首先尝试在TLB中找，如果找到了，直接从TLB返回物理页，找不到才会继续去L1,L2,L3查找。从缓存中找到物理页之后，会记录到TLB中。每次从TLB中访问，该项的次数会加1.TLB满了之后会把访问次数少的优先移除。

TLB分为指令TLB（ITLB）和数据TLB（DTLB）。

TLB大概的结构

虚拟页针 物理页针 attribute 次数 （PCID 64位才有）

### 3.实验：验证TLB的存在

申请2块内存 A,B

先把A内存的PTE挂到0地址上

访问0地址，把结果保存在一个变量里

然后再把B内存的PTE挂到0地址上

再访问一次保存在另一个变量里

首先在GDT 48构造一个提权无参调用门



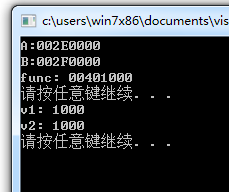
eq 80b99048 0040ec00`00081000

为了方便，实验前关闭增量链接 ASLR

代码如下：

|  |
| --- |
| #include "stdafx.h"  #include<Windows.h>  int \*A;  int \*B;  int v1,v2;  \_\_declspec(naked) void func()  {  \_\_asm  {  pushad;  pushfd;  push 0x30;  pop fs;  // calc A.PTE  mov eax, [A];  shr eax, 9;  and eax, 0x7FFFF8;  add eax, 0xC0000000;  mov edx, [eax];  mov ecx, [eax+4]; // ecx.edx now is A.PTE  // NULL.PTE = A.PTE  mov dword ptr ds:[0xC0000000], edx;  mov dword ptr ds:[0xC0000004], ecx;  // v1 = \*NULL  mov eax, dword ptr ds:[0];  mov v1, eax;  // calc B.PTE  mov eax, [B];  shr eax, 9;  and eax, 0x7FFFF8;  add eax, 0xC0000000;  mov edx, [eax];  mov ecx, [eax+4]; // ecx.edx now is B.PTE  // NULL.PTE = B.PTE  mov dword ptr ds:[0xC0000000], edx;  mov dword ptr ds:[0xC0000004], ecx;  // v2 = \*NULL  mov eax, dword ptr ds:[0];  mov v2, eax;  popfd;  popad;  push 0x3b;  pop fs;  retf;  }  }  int \_tmain(int argc, \_TCHAR\* argv[])  {  A = (int\*)VirtualAlloc(0,0x1000,MEM\_COMMIT,PAGE\_READWRITE);  B = (int\*)VirtualAlloc(0,0x1000,MEM\_COMMIT,PAGE\_READWRITE);  A[0] = 0x1000;  B[0] = 0x2000;  v1 = 0;  v2 = 0;  printf("A:%p\n",A);  printf("B:%p\n",B);  printf("func: %p\n", func);  system("pause");  \_\_asm  {  push 0x48;  push 0;  call fword ptr ds:[esp];  add esp,8;  }  printf("v1: %x\n", v1);  printf("v2: %x\n", v2);  system("pause");  return 0;  } |

运行结果如图：



实验现象是，v1和v2都是A的值，因为给0地址挂了A.PTE后，访问0地址，会在TLB中缓存0地址到物理页帧A.PTE的记录。下次把B.PTE挂到0，再访问B时，由于TLB中已经有0地址的缓存，所以直接从TLB中取物理页帧，这就取到了A.PTE，然后给v2赋了A的值。

这样就能证明TLB确实存在。

### 4.实验：测试切换CR3对TLB的影响

刷一下CR3就能清空TLB中G=0的记录。

G位叫全局位，它只对指向2MB大页的PDE和指向4KB小页的PTE有效，指向页表的PDE则忽略G位。

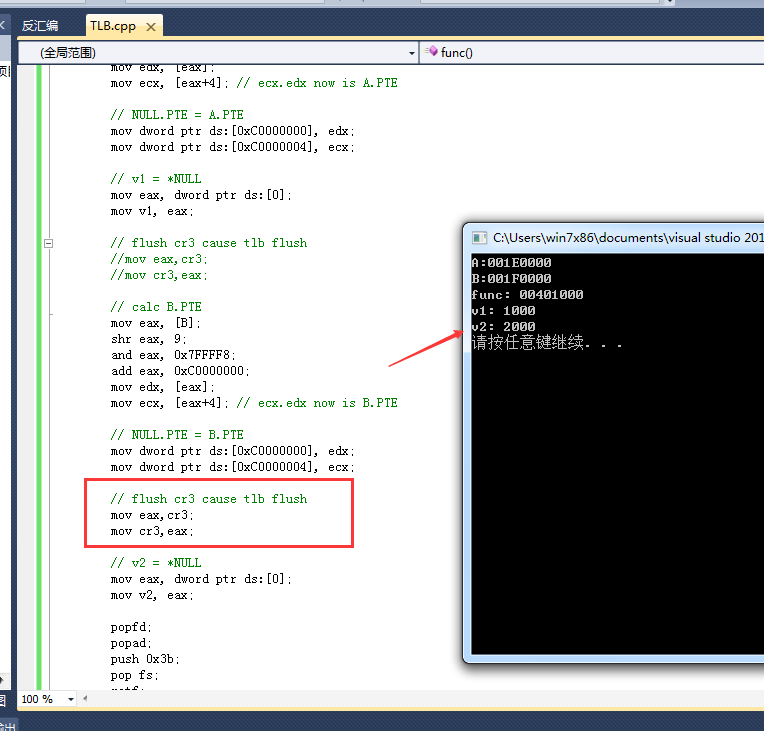


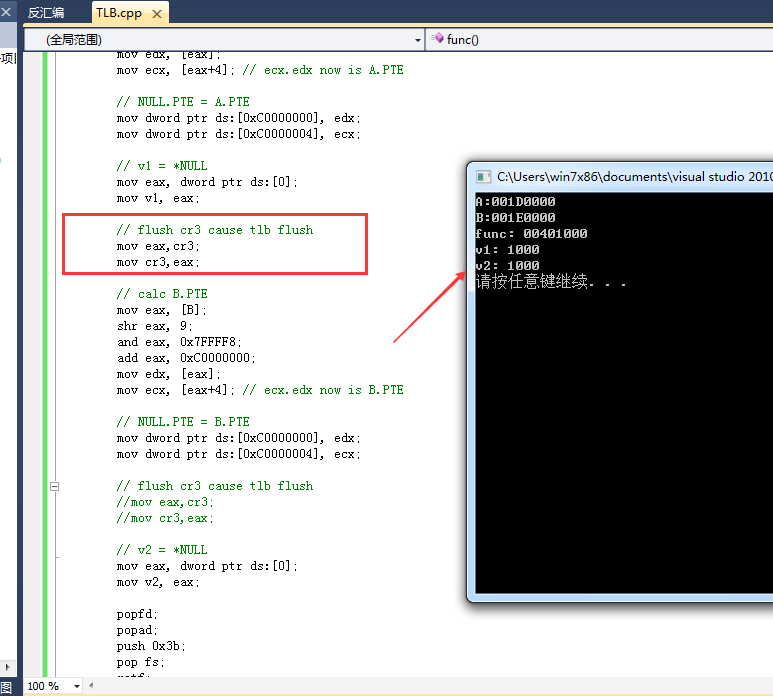
CR4.PGE是总开关，CR4.PGE=1即全局页开启时，如果G=1，那么刷新CR3时不会删除该项的记录。

我们申请的内存G=0，所以不影响，在给0地址挂B.PTE之前，刷一下CR3看看，代码修改如下：

|  |
| --- |
| #include "stdafx.h"  #include<Windows.h>  int \*A;  int \*B;  int v1,v2;  \_\_declspec(naked) void func()  {  \_\_asm  {  pushad;  pushfd;  push 0x30;  pop fs;  // calc A.PTE  mov eax, [A];  shr eax, 9;  and eax, 0x7FFFF8;  add eax, 0xC0000000;  mov edx, [eax];  mov ecx, [eax+4]; // ecx.edx now is A.PTE  // NULL.PTE = A.PTE  mov dword ptr ds:[0xC0000000], edx;  mov dword ptr ds:[0xC0000004], ecx;  // v1 = \*NULL  mov eax, dword ptr ds:[0];  mov v1, eax;  // flush cr3 cause tlb flush  // doesnt work here!  //mov eax,cr3;  //mov cr3,eax;  // calc B.PTE  mov eax, [B];  shr eax, 9;  and eax, 0x7FFFF8;  add eax, 0xC0000000;  mov edx, [eax];  mov ecx, [eax+4]; // ecx.edx now is B.PTE  // NULL.PTE = B.PTE  mov dword ptr ds:[0xC0000000], edx;  mov dword ptr ds:[0xC0000004], ecx;  // flush cr3 cause tlb flush  mov eax,cr3;  mov cr3,eax;  // v2 = \*NULL  mov eax, dword ptr ds:[0];  mov v2, eax;  popfd;  popad;  push 0x3b;  pop fs;  retf;  }  }  int \_tmain(int argc, \_TCHAR\* argv[])  {  A = (int\*)VirtualAlloc(0,0x1000,MEM\_COMMIT,PAGE\_READWRITE);  B = (int\*)VirtualAlloc(0,0x1000,MEM\_COMMIT,PAGE\_READWRITE);  A[0] = 0x1000;  B[0] = 0x2000;  v1 = 0;  v2 = 0;  printf("A:%p\n",A);  printf("B:%p\n",B);  printf("func: %p\n", func);  //system("pause");  \_\_asm  {  push 0x48;  push 0;  call fword ptr ds:[esp];  add esp,8;  }  printf("v1: %x\n", v1);  printf("v2: %x\n", v2);  system("pause");  return 0;  } |

这里有个奇怪的现象，在我的机器上，如果像火哥课上那样，在给v1赋值后马上刷CR3，则不起作用，只有在第二次访问0地址之前刷CR3，才有用，对比效果如下图：

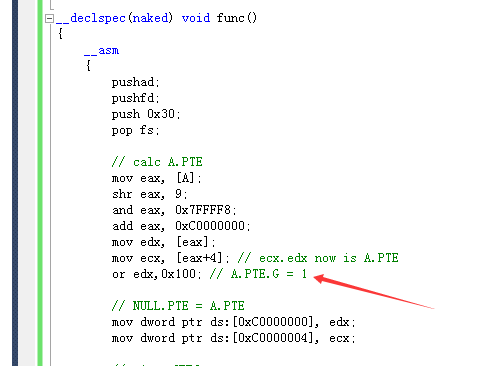




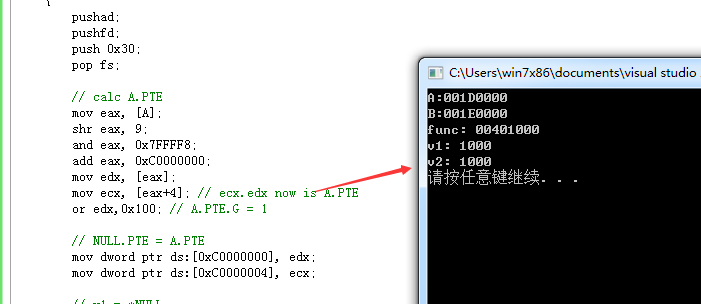
### 实验：G=1的页切CR3时TLB不会刷新

当 CR4.PGE=1时，启用全局页机制，此时G=1的页切换CR3时不会刷TLB

修改裸函数如下：



把A.PTE复制给0地址前，把G位置1



则切CR3后并不刷新TLB。

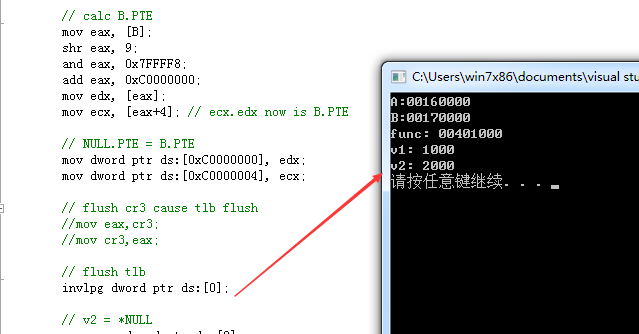
### 实验：invlpg 强刷TLB

有一个指令可以强刷线性地址TLB

invlpg dword ptr ds:[0]

在上一个实验的基础上，不刷CR3了，直接用invlpg指令强刷TLB，修改裸函数代码如下：

|  |
| --- |
| \_\_declspec(naked) void func()  {  \_\_asm  {  pushad;  pushfd;  push 0x30;  pop fs;  // calc A.PTE  mov eax, [A];  shr eax, 9;  and eax, 0x7FFFF8;  add eax, 0xC0000000;  mov edx, [eax];  mov ecx, [eax+4]; // ecx.edx now is A.PTE  or edx,0x100; // A.PTE.G = 1  // NULL.PTE = A.PTE  mov dword ptr ds:[0xC0000000], edx;  mov dword ptr ds:[0xC0000004], ecx;  // v1 = \*NULL  mov eax, dword ptr ds:[0];  mov v1, eax;  // flush cr3 cause tlb flush  // doesnt work here!  //mov eax,cr3;  //mov cr3,eax;  // calc B.PTE  mov eax, [B];  shr eax, 9;  and eax, 0x7FFFF8;  add eax, 0xC0000000;  mov edx, [eax];  mov ecx, [eax+4]; // ecx.edx now is B.PTE  // NULL.PTE = B.PTE  mov dword ptr ds:[0xC0000000], edx;  mov dword ptr ds:[0xC0000004], ecx;  // flush cr3 cause tlb flush  //mov eax,cr3;  //mov cr3,eax;  // flush tlb  invlpg dword ptr ds:[0];  // v2 = \*NULL  mov eax, dword ptr ds:[0];  mov v2, eax;  popfd;  popad;  push 0x3b;  pop fs;  retf;  }  } |



确实把TLB刷了。

### 7.杂项

PCID 类似于进程ID

还有缓存作用

win10 双页表，R3一个CR3 R0一个CR3

TLB缓存

保证同一进程ID 切换CR3时不会刷掉

hambaga

G=1的页会一直占用TLB吗，只有invlpg才能把它刷走

hambaga

所以G=1的页不能太多，不然TLB就废了

锁页 不会换到磁盘

某些游戏会检测到

R3是VirtualLock

R0是ExLockUserBuffer

# 作业

### 1.TLB上课实验 动手 做总结

已完成，在上面课堂笔记部分。

### 2.请用调用门 修改内核高2G的所有物理地址U/S位为1，然后返回R3的时候 随意访问一个高地址，打印出结果，修改地址 从 0x80010000-0xFFFF0000

### 3.思考题 利用调用门 如何实现访问其他进程的内存