## Windbg中使用 x \*! 命令

<0:000> x \*!  
start    end        module name  
00400000 00404000   temp         (deferred)  
77c10000 77c63000   MSVCRT       (pdb symbols)    
                                 c:\symbols\msvcrt.pdb\3D6DD5921\msvcrt.pdb  
77dd0000 77e5d000   ADVAPI32     (deferred)  
77e60000 77f46000   kernel32     (deferred)  
77f50000 77ff7000   ntdll        (deferred)  
78000000 78086000   RPCRT4       (deferred)  
这个命令展示所有模块的列表以及它们的开始和结束位置。我们的错误地址是77c3f10b，77c10000<=77c3f10b<=77c63000，因此可以确认错误发生在MSVCRT。下面我们来确定这个地址在哪。

## 栈结构

然后我们又把EBP设为ESP，我们可以把它看作是一个指针，栈就相当于一个DWORD类型的数组。因此，栈中各个变量与EBP的对应如下:  
[Parameter n]     ==  [EBP + n\*4 + 4] (The formula)  
...  
[Parameter 2]     ==  [EBP + 12]  
[Parameter 1]     ==  [EBP + 8]  
[Return Address]  ==  [EBP + 4]  
[Previous EBP]    ==  [EBP + 0]

## 栈回溯

0:000> kb  
ChildEBP RetAddr  Args to Child  
0012fd60 77c3e68d 77c5aca0 00000000 0012fdb0 MSVCRT!\_output+0x18  
0012fda4 0040102f 00000000 00000000 00403010 MSVCRT!printf+0x35  
0012ff4c 00401125 00000001 00323d70 00322ca8 temp!main+0x2f  
0012ffc0 77e814c7 77f944a8 00000007 7ffdf000 temp!mainCRTStartup+0xe3  
0012fff0 00000000 00401042 00000000 78746341 kernel32!BaseProcessStart+0x23  
0:000>  
“KB”命令能得到栈回溯。我们不会总是得到完整的栈回溯，我们会在更加深入的教程中讲解这点。在这篇简单的教程里，我们假定我们得到了完整的栈回溯。我们注意到，这个函数是printf，并且printf调用\_output。我们来反汇编printf，注

## 反汇编特定函数的代码

0:000> u MSVCRT!\_output  
MSVCRT!\_output:  
77c3f0f3 55               push    ebp  
77c3f0f4 8bec             mov     ebp,esp  
77c3f0f6 81ec50020000     sub     esp,0x250  
77c3f0fc 33c0             xor     eax,eax  
77c3f0fe 8945d8           mov     [ebp-0x28],eax  
77c3f101 8945f0           mov     [ebp-0x10],eax  
77c3f104 8945ec           mov     [ebp-0x14],eax

使用 u继续汇编未完成的代码；

## Gle找到最近出错的代码

在调用CreateFile函数之后，EAX中将存储返回值。我们注意到其值为ffffffff，也就是”Invalid Handle Value”，我们还想知道GetLastError的值，它存储在fs:34这个位置。FS是TEB选择子，我们可以把它dump出来。  
0:000> dd fs:34  
0038:00000034  00000002 00000000 00000000 00000000  
0038:00000044  00000000 00000000 00000000 00000000  
0038:00000054  00000000 00000000 00000000 00000000  
0038:00000064  00000000 00000000 00000000 00000000  
0038:00000074  00000000 00000000 00000000 00000000  
0038:00000084  00000000 00000000 00000000 00000000  
0038:00000094  00000000 00000000 00000000 00000000  
0038:000000a4  00000000 00000000 00000000 00000000  
CDB还有一种更快速的方式能做到这点,!gle:  
0:000> !gle  
LastErrorValue: (Win32) 0x2 (2) - The system cannot find the file specified.  
LastStatusValue: (NTSTATUS) 0xc0000034 - Object Name not found.

P命令是 单步执行 相当于ollydbg中 f7；

Fs:34 位置存储 getlasterror的值

然后，我们可以使用”da”，”dc”，“du”命令。”da”命令打印出ANSI字符串，”du”打印出Unicode字符串，”dc”和”dd”类似，不过它是打印出所有的字符，包括不可显示的。我们知道这是一个ANSI字符串，”da”命令:

Eax 用来存储返回的值，如果是引用则存储引用类型的地址值

eax=77c5c9e4 ebx=7ffdf000 ecx=00322cf8 edx=00322cf8 esi=00000000 edi=00000000  
eip=00401015 esp=0012ff28 ebp=0012ffc0 iopl=0         nv up ei pl zr na po nc  
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246  
temp!main+0x15:  
00401015 ff1504204000 call dword ptr [temp!\_imp\_\_CreateFileA (00402004)]{kernel3  
2!CreateFileA (77e7b476)} ds:0023:00402004=77e7b476  
0:000>  
eax=000007e8 ebx=7ffdf000 ecx=77f59037 edx=00140608 esi=00000000 edi=00000000  
eip=0040101b esp=0012ff44 ebp=0012ffc0 iopl=0         nv up ei ng nz ac pe cy  
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000293  
temp!main+0x1b:  
0040101b 8bf0             mov     esi,eax

eax有效句柄

eax=77c5c9e4 ebx=7ffdf000 ecx=00322cf8 edx=00322cf8 esi=00000000 edi=00000000  
eip=00401015 esp=0012ff28 ebp=0012ffc0 iopl=0         nv up ei pl zr na po nc  
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246  
temp!main+0x15:  
00401015 ff1504204000 call dword ptr [temp!\_imp\_\_CreateFileA (00402004)]{kernel3  
2!CreateFileA (77e7b476)} ds:0023:00402004=77e7b476  
0:000> p  
eax=ffffffff ebx=7ffdf000 ecx=77f939e3 edx=00000002 esi=00000000 edi=00000000  
eip=0040101b esp=0012ff44 ebp=0012ffc0 iopl=0         nv up ei ng nz na po nc  
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000286  
temp!main+0x1b:  
0040101b 8bf0             mov     esi,eax

eax无效句柄

每个线程都有自己的栈，因为多个线程公用一个栈会出现错乱，如果他们共享一个栈，将会出现数据覆盖的情况。

清理栈？  
  由谁来清理栈是很重要的一件事。首先就是保存内容。若果被调用者清理栈，就不需要在每个函数调用中产生一些代码来清理栈，缺点是你不能使用可变参数。像printf这样的函数就是使用可变参数，被调用者并不知道有多少参数压入了栈，它只能靠格式化的字符串猜测。如果这个字符串是printf  “%i  %i  %i”，它将适用3个栈中的参数来完成，而不管你是否真的压入了这个参数，这就可能产生错误。如果你压入了比这更多的参数，这将不会出现问题，因为调用者会负责清理栈。记住，可变参函数并不知道到底有多少参数，它们必须靠调用者告诉他们参数列表，Printf就是通过格式化字符串来知道到底有多少个参数的。  
  虽然它有可能可以清理栈，但是这还是不可行，因为函数在编译的时候不能确定有几个参数传给它，让调用者来清理栈会变得更加容易。  
  Intel提供了让被调用者清理栈的指令，这个指令是RET <Byte Count>，Byte Count就是压入栈中的参数的总字节数。

## 查看线程和线程栈空间

0:000> !teb  
TEB at 7ffde000  
    ExceptionList:        0012ffb0  
    StackBase:            00130000  
    StackLimit:           00031000  
    SubSystemTib:         00000000  
    FiberData:            00001e00  
    ArbitraryUserPointer: 00000000  
    Self:                 7ffde000  
    EnvironmentPointer:   00000000  
    ClientId:             00000928 . 00000898  
    RpcHandle:            00000000  
    Tls Storage:          00000000  
    PEB Address:          7ffdf000  
    LastErrorValue:       0  
    LastStatusValue:      0  
    Count Owned Locks:    0  
    HardErrorMode:        0  
0:000> ? 130000 - 31000  
Evaluate expression: 1044480 = 000ff000  
0:000>  
我使用!teb命令来显示TEB也就是Thread Environment Block。我们可以看到，1044480 bytes就是Windows给我们分配的栈空间

栈上溢的定义和原因  
  栈上溢和栈溢出是相反的。你以为你往栈中压入了很多数据，然后弹出了这么多数据，而事实上并没有这么多，所以你就超过了栈的上边界，从而造成了栈上溢。  
  
溢出和上溢  
  溢出和上溢其实就是对栈的处理不当，并且在最终使崩溃发生的不同的位置。如果你清理了过多并且返回将有可能发生上溢，出现这种情况的原因是，你以为你有这么多数据在栈中，而实际上没有。相反地，溢出和这类似。

## 遍历栈空间

调试器如何得到栈回溯  
  下一个话题就是，如何得到栈回溯？第一个答案就是符号。符号能告诉调试器栈中有多少参数，多少局部变量等等，因此调试器可以使用符号来确定怎样遍历栈以及显示信息。  
  如果没有符号，我们可以用基址指针。每个基址指针都指向以前的基址，基址+4指向返回地址，按这种规则我们就能遍历栈。虽然调试器不知道由多少参数，它只是Dump出来可能的参数的，最后依靠你自己来判定正确的参数。

我们当前的EBP值是0012fef4，这是栈中的一个指针，这个值指向以前的EBP，记住EBP+4==返回地址，EBP+8==参数。

## 栈和符号匹配，找到最近的call入口

我将会使用”DDS”命令，这个命令会将栈中的值和最近的符号匹配。

## Handle查看句柄

0:000> dd 0012f730  
0012f730  000007c4  
返回的值是7c4h，并不是程序中指向任何一个地址的指针，更不是内核内存中的指针。为了更深入，有一个调试器命令可以显示句柄信息，就是!handle。  
0:000> !handle 7c4 ff  
Handle 7c4  
  Type          File  
  Attributes    0  
  GrantedAccess 0x120089:  
         ReadControl,Synch  
         Read/List,ReadEA,ReadAttr  
  HandleCount   2  
  PointerCount  3  
  No Object Specific Information available  
我们可以看到，这确实是一个文件对象，还显示了它的属性和访问权限。但是我们并不知道这个句柄到底指向哪个文件

## 查看对应进程的句柄表

  在内核调试器中，我们使用!process 0 0，列举所有的进程，然后我们使用!handle命令，语法会有一些不一样，为了列举正确的句柄，我们需要指定相应的进程。  
kd> !process 0 0  
\*\*\*\* NT ACTIVE PROCESS DUMP \*\*\*\*  
...  
PROCESS fcc77200  SessionId: 0  Cid: 0338    Peb: 7ffdf000  ParentCid: 02c8  
    DirBase: 079de000  ObjectTable: fccc30c8  TableSize:  74.  
    Image: notepad.exe  
kd> !handle 120 ff fcc77200    
processor number 0  
PROCESS fcc77200  SessionId: 0  Cid: 0338    Peb: 7ffdf000  ParentCid: 02c8  
    DirBase: 079de000  ObjectTable: fccc30c8  TableSize:  74.  
    Image: notepad.exe  
  
Handle Table at e1e5f000 with 74 Entries in use  
0120: Object: fcd32448  GrantedAccess: 00120089  
Object: fcd32448  Type: (fced7c40) File  
    ObjectHeader: fcd32430  
        HandleCount: 1  PointerCount: 1  
        Directory Object: 00000000  Name: \TripItinerary.txt {HarddiskVolume1}  
第一个加粗的数字是进程对象，它是内存中的一个结构体，包含了进程中的一些信息。既然句柄是对应于特定进程的，我们就要告诉!handle它对应于哪个进程。”ff”就是说显示可能的所有信息，帮助文件中有具体的解释。我总是使用”ff”，因为我需要所有的信息。

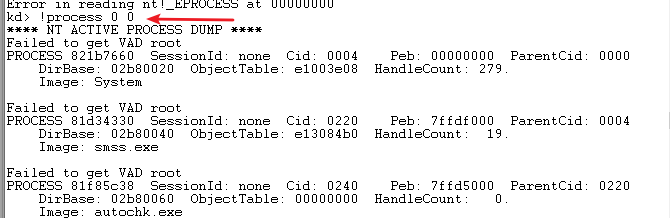
## 列举所有的句柄

0:001> !handle 0 0  
65545 Handles  
Type            Count  
Event           3  
Section         1  
File            1  
Port            1  
Directory       3  
WindowStation   2  
Semaphore       2  
Key             65530  
Desktop         1  
KeyedEvent      1  
在这个夸张的例子中，我们可以看到泄露的句柄类型是”Key”，也就是注册表句柄。下面我们要确定发生问题的位置。如果我们能找到这个键的信息，我们就能知道哪个键被打开了，然后缩小检查范围。

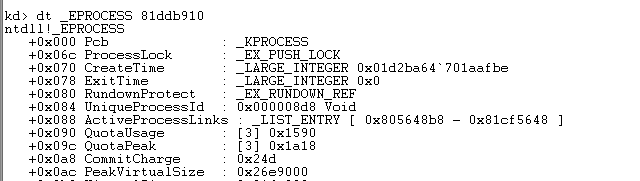
handle 0 ff Key  
Handle 2de0  
  Type          Key  
  Attributes    0  
  GrantedAccess 0xf003f:  
         Delete,ReadControl,WriteDac,WriteOwner  
         QueryValue,SetValue,CreateSubKey,EnumSubKey,Notify,CreateLink  
  HandleCount   2  
  PointerCount  3  
  Name          \REGISTRY\USER\S-1-5-21-789336058-706699826-1202660629-1003\Software  
  Object Specific Information  
    Key last write time:  01:10:03. 5/9/2004  
Key name Software  
我们看到，调试器给我们显示键的信息，并且HKCU\Software被打开的次数最多。我们可以确定这是一个key对象，也就是说可能使用RegOpenKey或RegOpenKeyEx打开的。我们已经确定了被打开的key，下面看第三步。  
  
第三步:浏览源码，调试程序  
  我们现在已经知道了类型，使用的API，还有被打开的资源。下面，我们就可以查看源码，找到这个地方或在RegOpenKey/RegOpenKeyEx上下断点来看栈回溯。然后我们就可以跟踪句柄，确定源头。Bounds Checker也能完成类似这样功能。  
  另外，我们还可以使用在”堆”那节讲述的寻找内存泄露的方法。我们可以将创建句柄的API包装成一些函数，然后将句柄放入链表中，释放的时候将其从链表中删除即可。缺点是，删除的时候遍历链表会很慢。  
//This method could return the actual key, but would require   
//a search of the linked list on a close.  
  
DWORD MyOpenKey(..., phKey)  
{  
    dwResult = RegOpenKey(... phKey);  
  
    pTemp = Allocate();  
    pTemp->pNext = gpHead;  
    gpHead = pTemp;  
    gpHead->hKey = phKey;  
    
    return dwResult;  
}  
  
//This method would allow faster look up in the close but   
//would also require a wrapper for all functions.  
  
PMYKEY MyOpenKey(...)  
{  
    hKey = RegOpenKey(...);  
  
    pTemp = Allocate();  
    pTemp->pNext = gpHead;  
    gpHead = pTemp;  
    gpHead->hKey = hKey;  
    
    return gpHead;  
}  
上面给出的只是一些伪代码。使用的时候必须加上临界区，或者其它一些方法。增加这个全局链表的目的是让调试扩展可以遍历它。这对大部分问题来说可能都太不需要了，并且可能使用在一些设置好的工程中。

## 内核调试

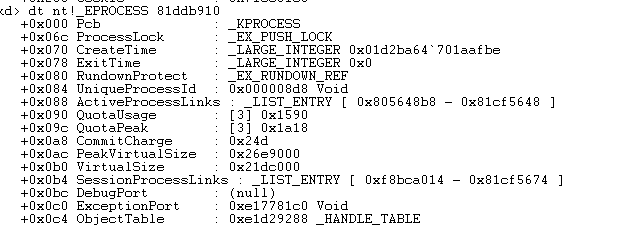
1.！process 0 0 命令测试结果



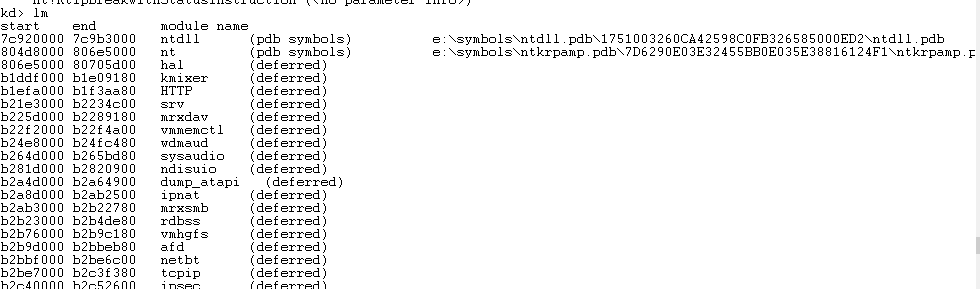
2. dt \_EPROCESS xxxxxxxx 地址测试结果



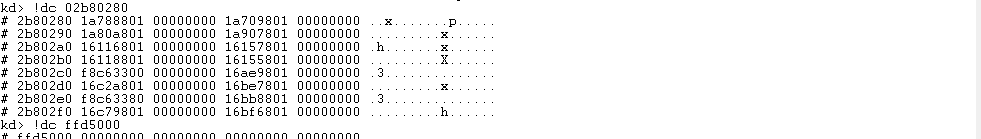
3. dt nt!\_EPROCESS XXXXXXXX 地址测试结果



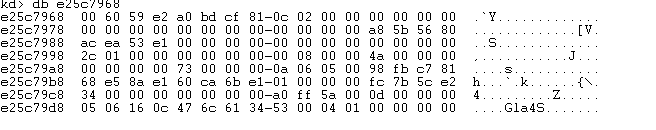
4. lm 测试结果



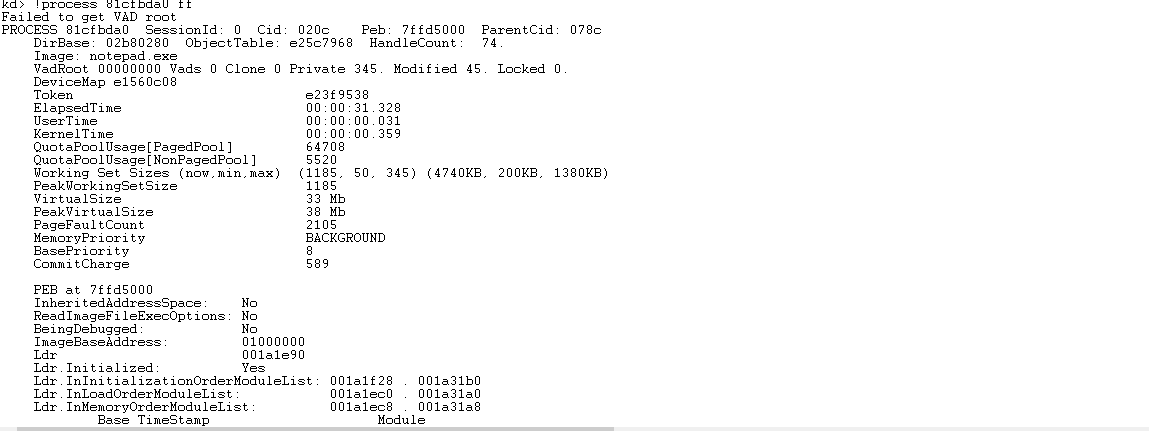
5.！dc命令测试 ，！db 对虚拟地址使用 ！开头



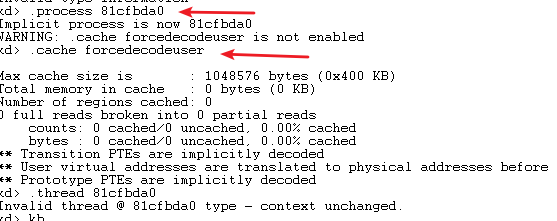
6.objecttable的数据测试



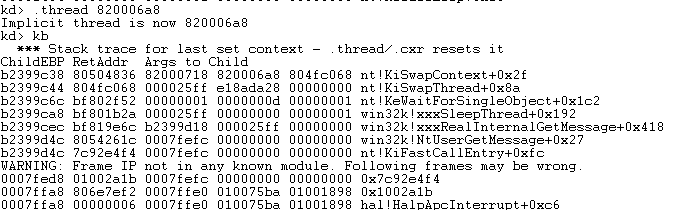
7.！process xxxxxxxx ff 详细显示进程结果



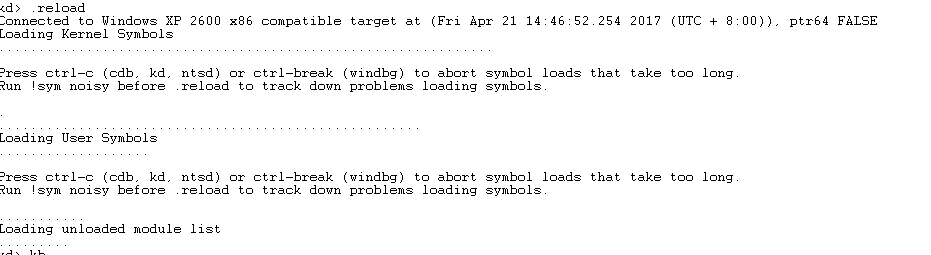
8 .process和 .cache forcedecodeuser的测试结果



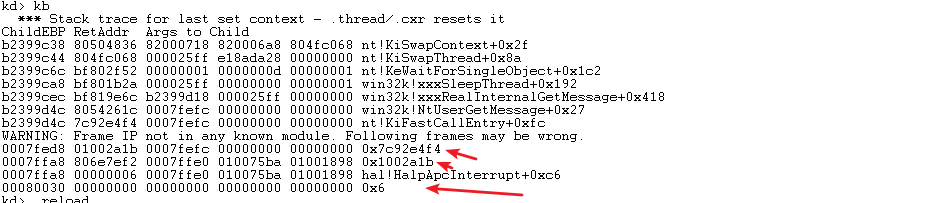
## 9 .thread xxxxxxxx 的结果测试 kb切换到线程栈

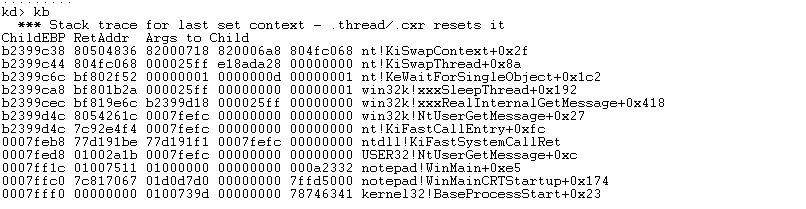


10 .reload 命令测试

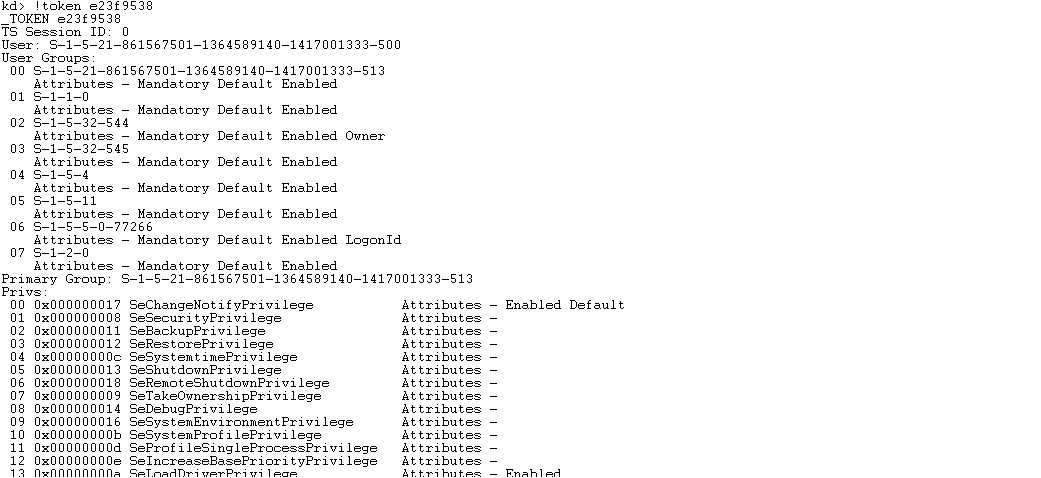


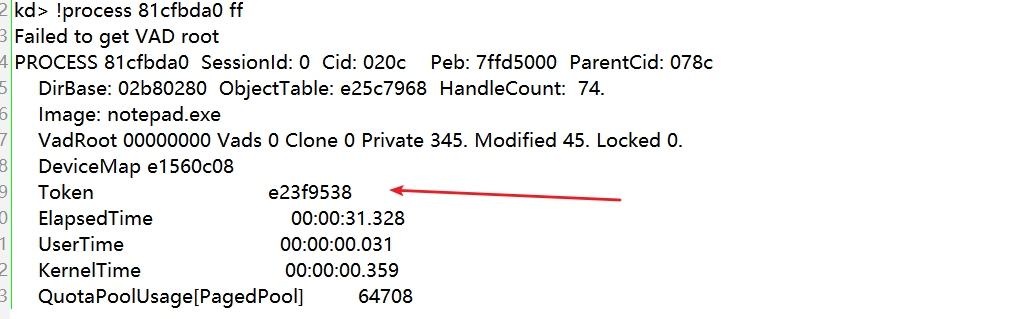
11 reload前后kb栈对比





12 进程令牌 ！token 命令测试

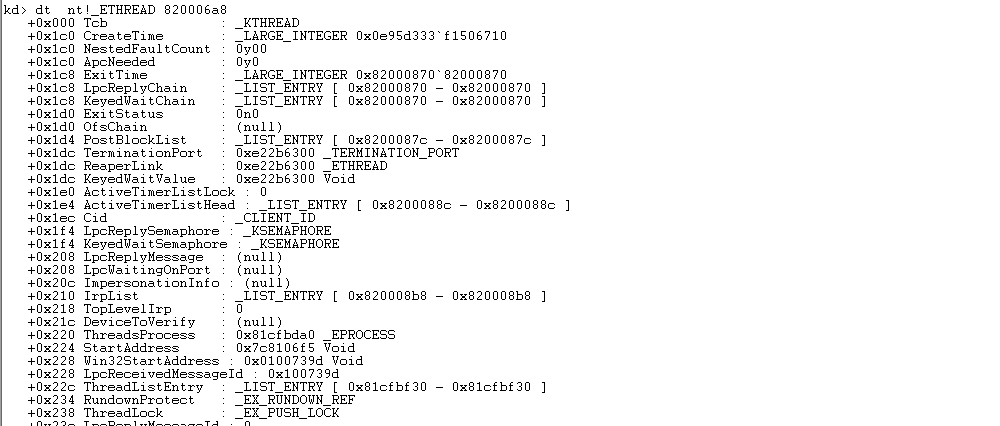


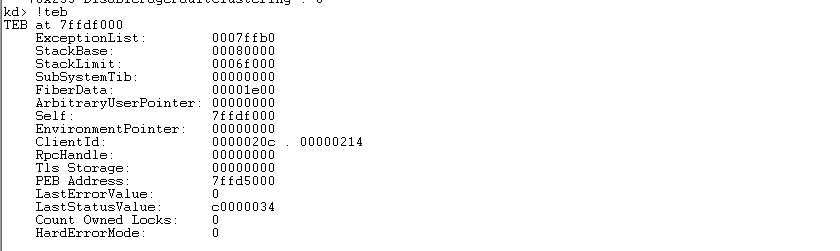


13 线程信息查看 ！thread

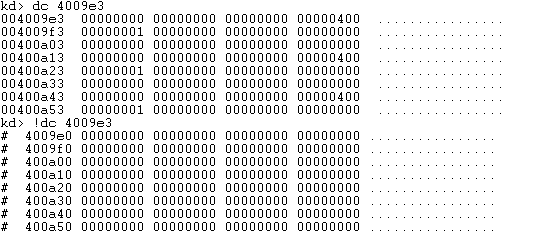


14 查看线程控制块 dt nt！\_ETHREAD XXXXXXXX 和！teb命令

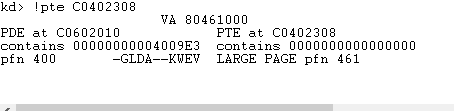




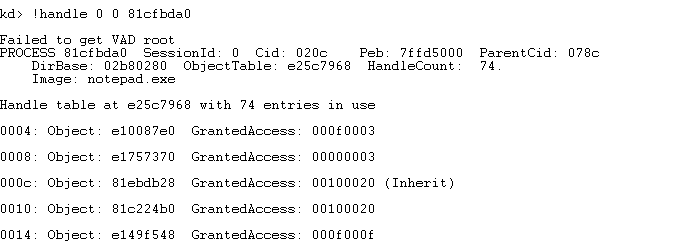
## 15 dc命令是否加！的区别



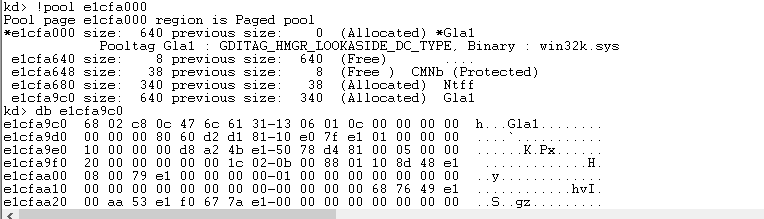
## 16.！pte命令测试



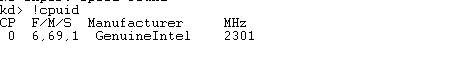
17 ！handle 0 0 xxxxxxxx 列举对应地址所有进程



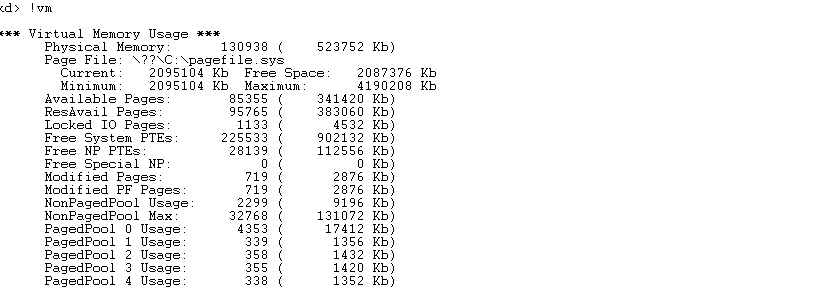
18.！pool指令测试



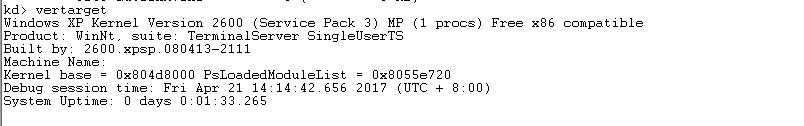
19 ！cpuid命令测试



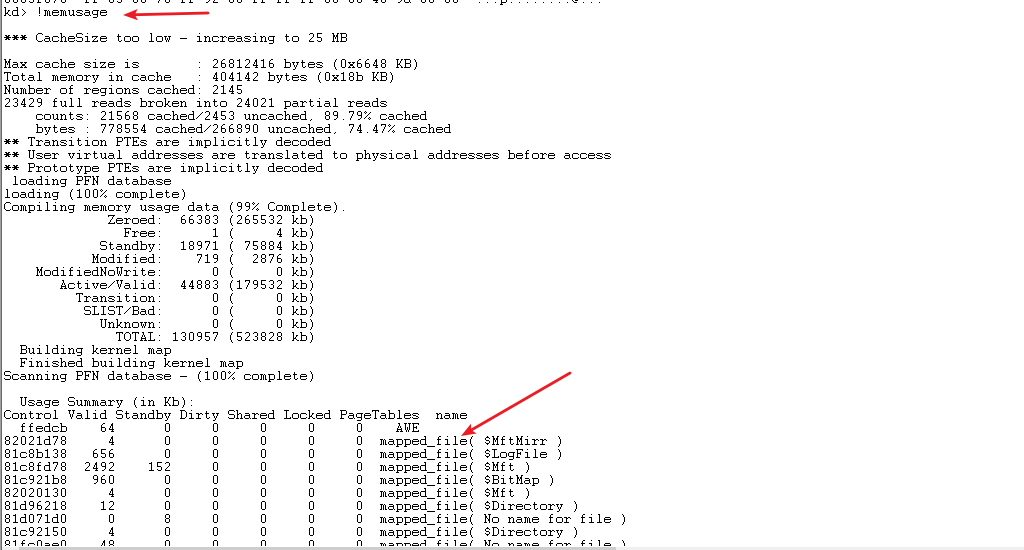
20 ！vm查看虚拟内存使用情况



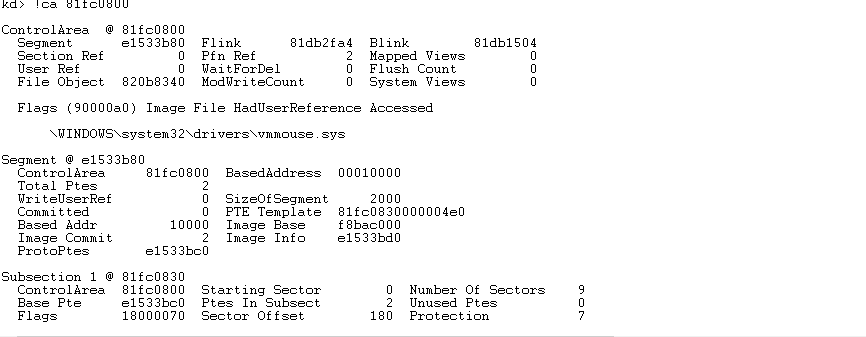
21 查看调试系统信息 vertarget



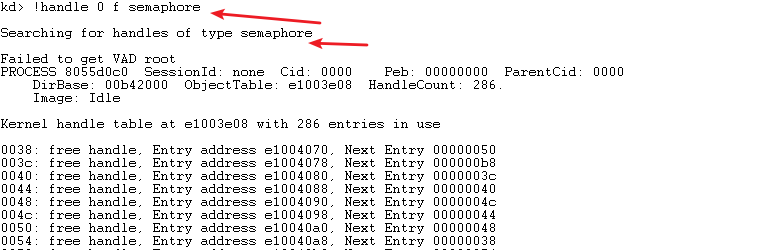
22 ！memusage 查看所有内存映射文件



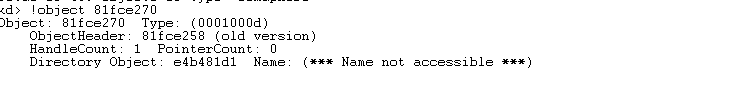
23 使用！ca命令查看具体的映射信息

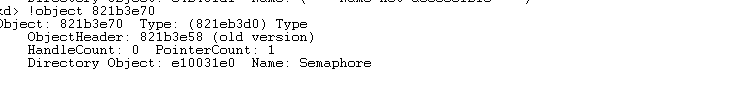


24 查看信号量 ！handle 0 f semaphore测试

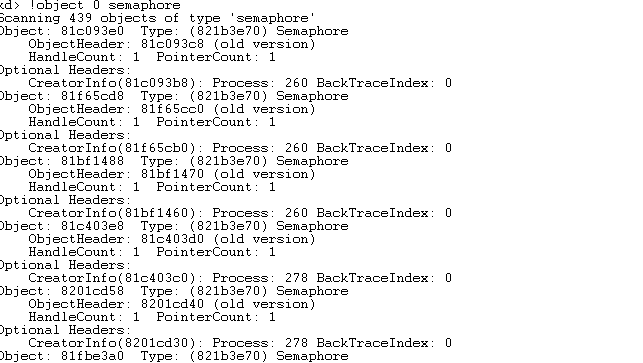


25 !object 访问特定地址的对象





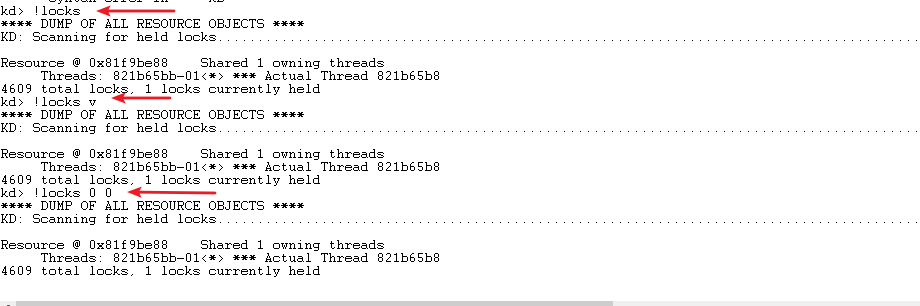
26 ！object 列举特定类型的对象



27. 根据栈信息，查看临界区，解除死锁 kb 和 ！cs 组合



28 ！lock显示临界区有关信息 ！lock v



29.内核态和用户态的调试

值得注意的是，内核态的命令用户态无法使用，用户态的命令内核态无法使用

内核态

Windbg 命令行 kb >> kernel debug

用户态 命令行 0001:x86 >>

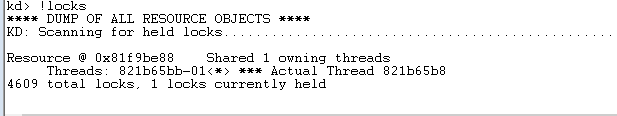
这是当临界区没有锁住时的输出。临界区相对于互斥量还有什么优势呢？因为它们是用户态的，因此它们不用进入内核，也就不用和其它进程去争抢。这让它实现起来更快，调试的时候更简单。对这个地址使用dd命令，你也可以看到这个数据结构很简单。  
  如果这个线程不存在，那么这个线程就是在释放这个锁之前就已经退出了。  
  
内核态可用的对象  
  这里我们讲述一些内核态下的对象。事件，信号量，互斥量都能在内核态下使用。事实上，用户态的程序也都是进入内核态，使用相同的函数然后创建这些对象的。另外还有其他一些内核态的对象。它们是自旋锁和ERESOURCE。

30 自旋锁  
  自旋锁是用来在多处理器系统中保护资源的一种同步对象。自旋锁和临界区的不同之处在于，第二个处理器会在这个锁上一直自旋知道获取它，而不是让其他线程去运行。  
  在单处理器系统上，自旋锁只是将IRQL增高，以使线程不能切换，让代码运行。但是你也不能使用分页内存，并且你只能进行一小部分操作。  
hal!KfAcquireSpinLock:  
80069850 33c0             xor     eax,eax  
80069852 a024f0dfff       mov     al,[ffdff024]  
80069857 c60524f0dfff02   mov     byte ptr [ffdff024],0x2  
8006985e c3               ret

这是指向全局IRQL的一个地址。调用KeAcquireSpinLock会将IRQL置为2。然后它保存以前的IRQL，用来在KeReleaseSpinLock中恢复。  
  IRQL表示操作系统当前处于的一个中断级。下面是NTDDK中的定义。  
#define PASSIVE\_LEVEL 0             // Passive release level  
#define LOW\_LEVEL 0                 // Lowest interrupt level  
#define APC\_LEVEL 1                 // APC interrupt level  
#define DISPATCH\_LEVEL 2            // Dispatcher level  
因此，自旋锁会把操作系统升到DISPATCH\_LEVEL。你可以在MSDN中找到更多关于IRQL的信息。  
自旋锁函数在多处理器系统中会有一些不同。它们会一直自旋直到获得它。  
“LOCK”这个汇编指令会锁住总线，防止其他处理器读或写同一块内存区域。以0为参数使用BTS指令，就是把0放入一个carry flag，然后把这个0位置为1。  
JB指令会在carry flag为1时(也就是说它以前是1)跳转。如果0位是1，它会做一个test，如果0位不是1，它就会跳回去然后再继续。如果0位是1，就意味着它正在被占有，因此它会做一个”pause”。  
hal!KfAcquireSpinLock:  
80065420 8b158000feff     mov     edx,[fffe0080]  
80065426 c7058000feff41000000 mov dword ptr [fffe0080],0x41  
80065430 c1ea04           shr     edx,0x4  
80065433 0fb68280a30680   movzx   eax,byte ptr [edx+0x8006a380]  
8006543a f00fba2900       lock    bts dword ptr [ecx],0x0  
8006543f 7203             jb      hal!KfAcquireSpinLock+0x24 (80065444)  
80065441 c3               ret  
80065442 8bff             mov     edi,edi  
0: kd> u  
hal!KfAcquireSpinLock+0x24:  
80065444 f70101000000     test    dword ptr [ecx],0x1  
8006544a 74ee             jz      hal!KfAcquireSpinLock+0x1a (8006543a)  
8006544c f390             pause  
8006544e ebf4             jmp     hal!KfAcquireSpinLock+0x24 (80065444)  
这是自旋锁工作的本质，事实上并没有很多工作，并且大部分程序都不会用到自旋锁。一般情况下，信号量或互斥量已经足够了。如果你想使用自旋锁，可以去阅读MSDN。另外还有”queued”自旋锁，可以提供更好的表现。

## 31 ERESOURCE

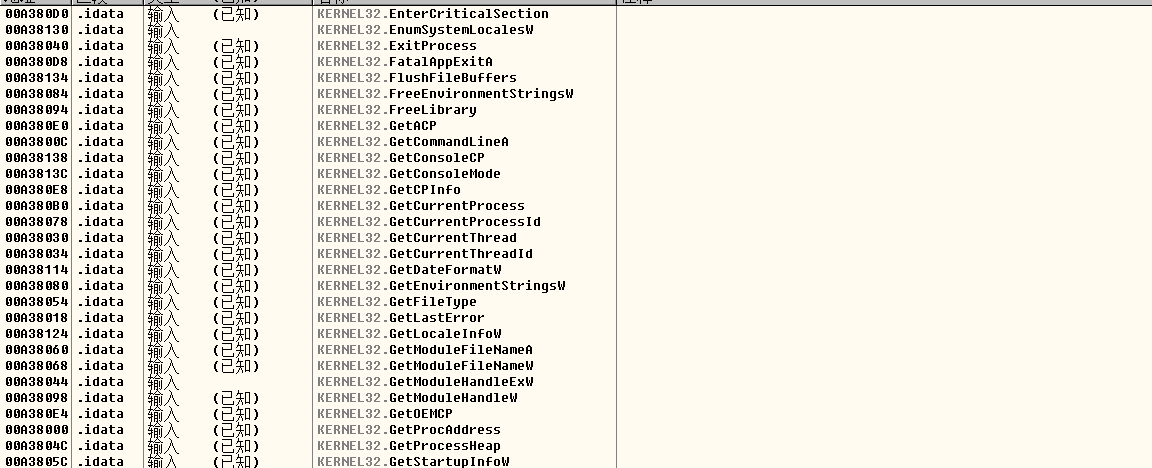
。ERESOURCE是允许你在内核中共享去独占访问的一个数据结构。共享的意思是多个线程可以获得它，独占的意思是只有一个线程可以获得它。  
  需要注意的是，ERESOURCE存在于非分页内存的一个全局链表中。这就是说，如果你释放这块内存或覆盖掉这个数据结构，可能会造成崩溃。  
  在内核调试器中，你可以使用!locks来Dump出所有的系统中所有的锁。



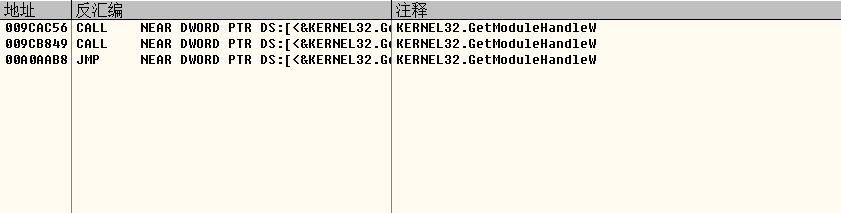
LEA和MOV命令的区别

LEA指令的功能是取偏移地址，例如LEA AX,[1000H]，作用是将源操作数[1000H]的偏移地址1000H送至AX；  
MOV指令的功能是传送数据，例如MOV AX,[1000H]，作用是将1000H作为偏移地址，寻址找到内存单元，将该内存单元中的数据送至AX。

## 32 Ollydbg使用ctrl+n 查看输入 输出表，查看选用函数引用



鼠标左键单击目标函数，按enter键 查看目标函数的引用情况



如果想要返回 到 输入输出表界面的操作步骤是

1. 点击菜单栏中 c 返回到汇编窗口
2. 使用ctrl+n 打开输入输出表

## 33 Windbg设置条件断点

bp kernel32!CreateFileW "r $t1=poi(esp+4);as /mu $FileName $t1;.block{.if($sicmp(\"${$FileName}\",\"C:\\haha.txt\")==0){.echo hahaha}.else{.echo nonono}}"

bp kernel32!CreateFileW 设置函数断点

"r $t1=poi(esp+4); 取esp+4的值，复制给windbg的t1虚拟寄存器，t1-t19

as /mu $FileName $t1; 给t1起一个别名，因为寄存器无法直接和定值进行比较

.block{ 执行一个代码块的语法

.if($sicmp(\"${$FileName}\",\"C:\\haha.txt\")==0) if-else结构

$sicmp 字符串比较

{.echo hahaha}

.else

{.echo nonono}

}"

34 使用 ！prcoess address 7 查看进程详细信息

kd> !process 81f86520   7  
PROCESS 81f86520  SessionId: 0  Cid: 05a8    Peb: 7ffde000  ParentCid: 05e8  
    DirBase: 04f002e0  ObjectTable: e143e718  HandleCount:   7.  
    Image: dbgTCB.exe  
    VadRoot 820674b8 Vads 22 Clone 0 Private 43. Modified 0. Locked 0.  
    DeviceMap e1dfb5c8  
    Token                             e14988c0  
    ElapsedTime                       00:00:00.046  
    UserTime                          00:00:00.015  
    KernelTime                        00:00:00.000  
    QuotaPoolUsage[PagedPool]         12364  
    QuotaPoolUsage[NonPagedPool]      880  
    Working Set Sizes (now,min,max)  (174, 50, 345) (696KB, 200KB, 1380KB)  
    PeakWorkingSetSize                174  
    VirtualSize                       7 Mb  
    PeakVirtualSize                   7 Mb  
    PageFaultCount                    167  
    MemoryPriority                    BACKGROUND  
    BasePriority                      8  
    CommitCharge                      51  
    
  //得到TCB地址  
        THREAD 81f862a8  Cid 05a8.0588  Teb: 7ffdd000 Win32Thread: 00000000 RUNNING on processor 0  
        Not impersonating  
        DeviceMap                 e1dfb5c8  
        Owning Process            0       Image:         <Unknown>  
        Attached Process          81f86520       Image:         dbgTCB.exe  
        Wait Start TickCount      9170           Ticks: 1 (0:00:00:00.015)  
        Context Switch Count      23               
        UserTime                  00:00:00.000

## 35 esi edi

寄存器ESI、EDI、SI和DI称为变址寄存器(Index Register)，它们主要用于存放存储单元在段内的偏移量，用它们可实现多种存储器操作数的寻址方式，为以不同的地址形式访问存储单元提供方便。  
变址寄存器不可分割成8位寄存器。作为通用寄存器，也可存储算术逻辑运算的操作数和运算结果。它们可作一般的存储器指针使用。在字符串操作指令的执行过程中，对它们有特定的要求，而且还具有特殊的功能。

DDk 的含义

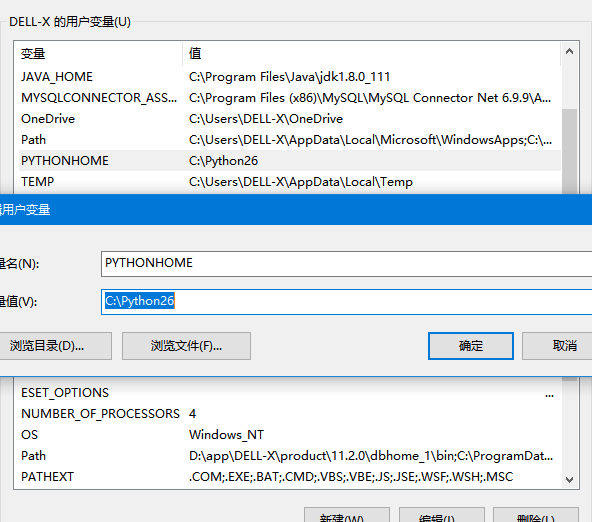
Ddk是windows develop kit wdk的旧称；

IDA 出现

# [idapython import site failed](http://blog.csdn.net/x356982611/article/details/51527450)

的警告的解决办法

设置环境变量 PYTHONHOME



设置完成后，重启软件即可

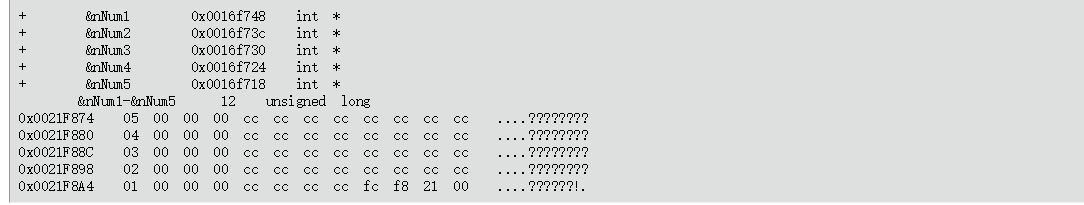
Windows可执行文件的载入过程：  
  
1、windows为主线程创建默认的堆、栈和线程局部存储空间。  
2、windows将可执行文件得主体和他的所有DLL载入虚拟存储空间。如果有必要，DLL会被重定位。  
3、windows解释所有的函数和数据引入符号。  
4、对所有DLL\_PROCESS\_ATTACH符号中带有该文件的DLL，windows调用其DLLMain函数。  
5、windows调用C的运行时刻函数库中的WinMainCRTStartup启动代码。  
6、运行时刻函数库分解命令行，并设置环境变量。  
7、运行时刻函数库调用主程的WinMain函数。对于MFC程序，WinMain立即调用AfxWinMain。该函数调用AfxWinInit  
初始化MFC自身，然后调用CWinAPP::InintInstance初始化应用程序。

## Win32控制台main方法参数的意义

int WINAPI WinMain（hInstance，hPrevInstance，lpComdLine，nCmdShow）；

HInstance 该进程所对应的应用程序当前实例的句柄。WinMainCRTStartup函数通过调用GetStartupInfo函数获得该参数的值。该参数实际上是应用程序被加载到进程虚拟地址空间的地址，通常情况下，对于大多数进程，该参数总是0X00400000。  
  
hPrevInstance 应用程序前一实例的句柄。由于Win32应用程序的每一个实例总是运行在自己的独立的进程地址空间中，因此，对于Win32应用程序，WinMainCRTStartup函数传给该参数的值总是NULL。如果应用程序希望知道是否有另一个实例在运行，可以通过线程同步技术，创建一个具有唯一名称的互斥量，通过检测这个互斥量是否存在可以知道是否有另一个实例在运行。  
  
lpCmdLine 命令行参数的指针。该指针指向一个以0结尾的字符串，该字符串不包括应用程序名。  
  
nCmdShow 指定如何显示应用程序窗口。如果该程序通过在资源管理器中双击图标运行，WinMainCRTStartup函数传给该参数的值为SW\_SHOWNORMAL。如果通过在另一个应用程序中调用CreatProcess函数运行，该参数由CreatProcess函数的参数lpStartupInfo(STARTUPINFO.wShowWindow)指定。

## Int在栈中的存储问题



关于内存中上述数据的说明

1. 由于存储的是int 基本类型，所以数据存储在栈中，而栈是32位对齐的，所以导致四个数据每个栈一行，
2. 由于未初始化的数据是cc，所以出了初始化的1，2，3，4，5之外其他的都是cc ，但是值得桌椅的是01那行的末尾。是有数据的

Fc f8 21 00 即 0021f8fc

而变量在栈中的地址是 0021f8a4

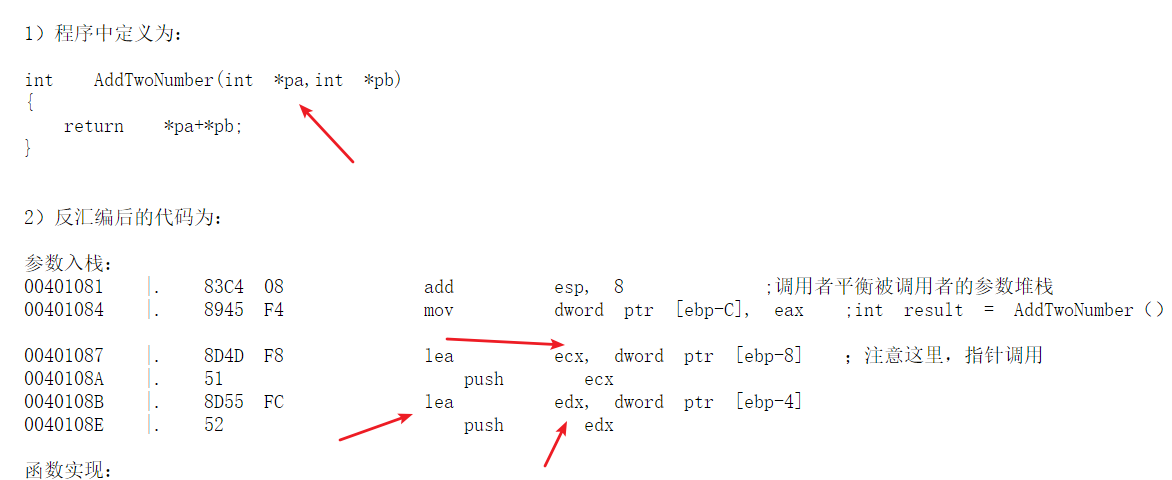
说明0024f8fc是一个栈地址 0021f8fc-0021f8a4=0x58h byte

猜测 0021f8fc 可能是先前方法的ebp的值

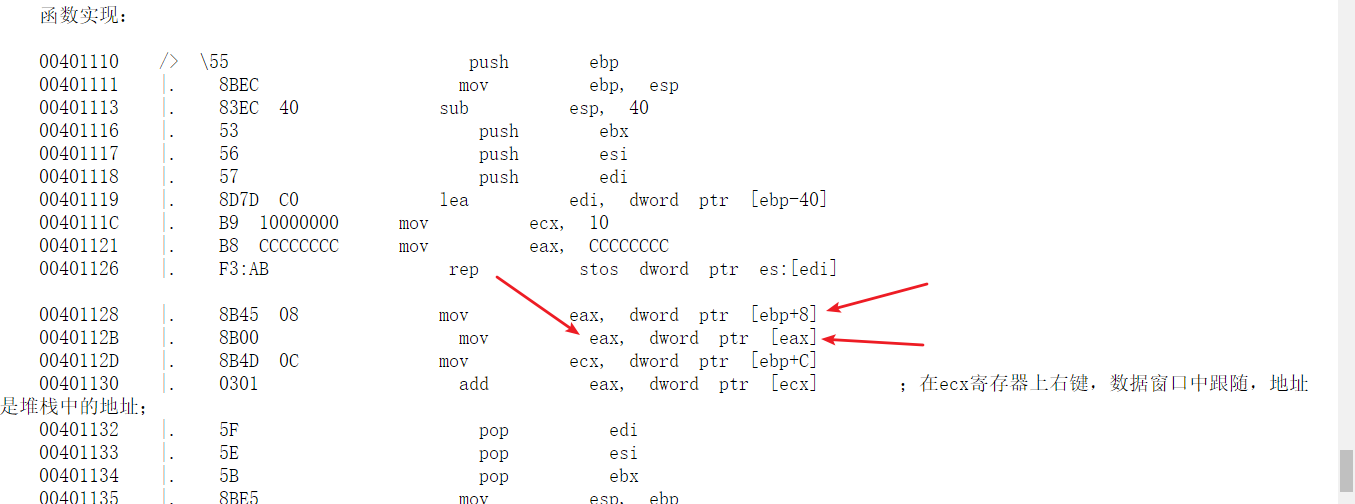
### LEA命令的作用

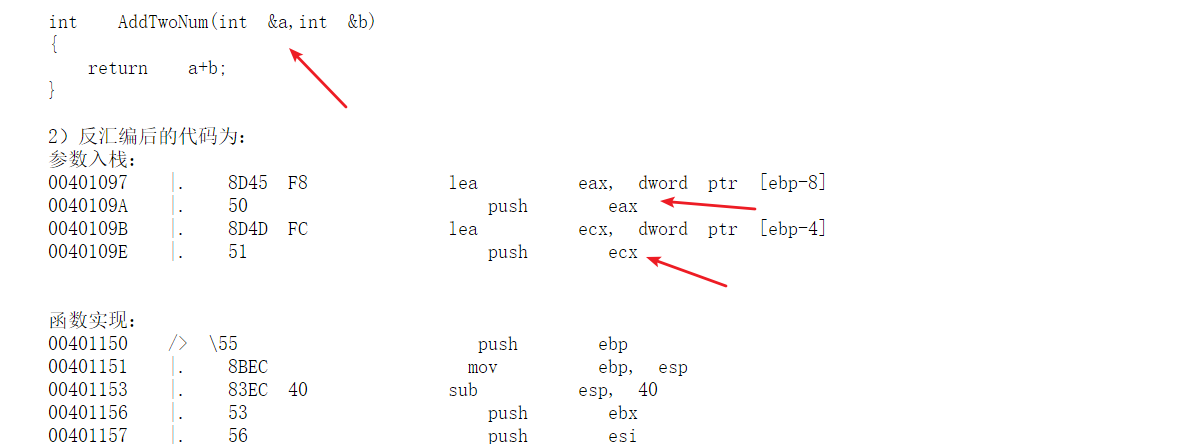
Lea的命令是将有效的地址或者是立即数，从源操作数送到目的操作数

通常反汇编中使用lea，是出现了函数的指针

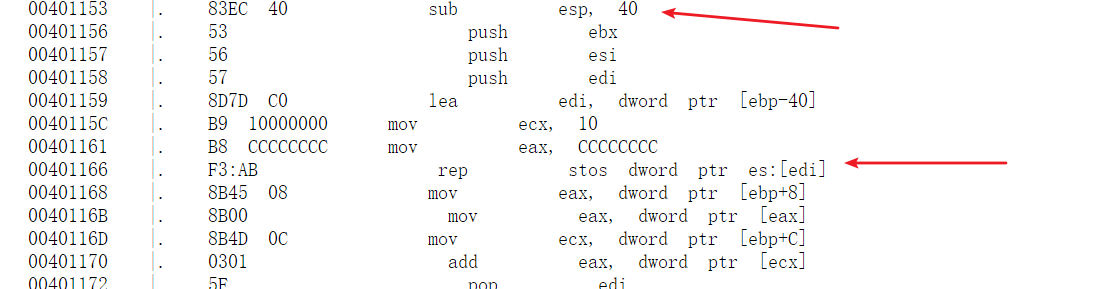


使用了ecx和edx作为指针的寄存器





一个是传入指针，另一个是传入地址



Sub esp，40

值得说明的是

1. push执行只和esp指令有关和ebp指令无关，通过sub esp，40进行了堆栈控件的申请

push指令相当与 sub esp，4

也就是说sub esp，40相当于申请10个 栈空间，而使用push 指令是无法使用这些申请空间的，如果想要访问这些被申请的空间的操作是

Mov eax,ebp

Sub eax,12

Mov eax,500;

或者是

Mov dword ptr [ebp-10],500;

虽然栈是32位对齐的，但是mov操作是操作内存的，所以可以避开对齐的约束，而对某个栈帧的某些位赋值。

此外，这些申请的栈空间的作用应该是存储方法的临时变量的作用。

栈和堆 在实际中的区别是 大的对象，或者不经常移动的数据通常存储在堆中。在内存中 当内存地址是由低到高时，栈是向上增长，堆是向下增长。每个线程都有一个独立栈，防止数据混乱，每个程序一个堆。

 从上述我们可以看出，三种函数调用方式上 ，传值调用和后两种在参数的使用和功能实现上都有不同，由于是指针，  
  所以在参数入栈时用了lea指令，并且，传址调用和引用调用的功能性代码完全相同，而且，用引用调用能简化书写，  
  它们的不同，在这个程序中，唯一的区别就是参数入栈时，使用的寄存器不同；由于使用了Lea指令，传址调用和引用调用  
  函数执行的时间肯定要比传值调用的时间长，所以如果是在函数的递归调用和嵌套调用中，如果为了效力考虑，在完成  
  相同的功能的函数设计上，尽量使用传值调用；另外在传址调用和引用调用上，建议大家使用引用调用，因为它们在实现  
  上一样的，而引用调用使用起来更加方便；  
  并且有一点想让大家注意，大家在想栈这个数据结构，不要想着Push， pop后栈中的数据就不再了，  
  物理内存一般为动态RAM，一般为MOS型，它具有客观实在性，因此，即使你pop平衡了栈后，原来的数据仍然存放在内存中  
  也就是，除非断电或再一次被使用，否则内存中的数据不会改变；

### Window64 反汇编

1. window64汇编的参数寄存器是rcx，rdx，r8，r9作为通用寄存器，rsp 和esp的作用相同
2. 如果函数的传入参数多于4个那么，多余的参数则利用栈传递
3. 64汇编时，当小于4个参数的时候不需要压栈，直接使用rcx,rdx,r8,r9这四个寄存器传值
4. 当反汇编64位程序的时候，IDA 分为32位程序和64位程序
5. Jne 跳转 依据的是 z 寄存器 z寄存器不等于0则跳。Xor只是异或，没有栈操作；

64位汇编可以使用的寄存器 rcx，rdx，r8，r9，rsp，edx，rdx，rsi，eax，ecx

### 汇编程序编写

比如 我们有这样一个C函数  
#include<stdio.h>  
long test(int a,int b)  
{  
    a = a + 1;  
    b = b + 100;  
    return a + b;  
}  
void main()  
{    
  printf("%d",test(1000,2000));  
}  
  
写成32位汇编就是这样  
;//////////////////////////////////////////////////////////////////////////////////////////////////////  
.386  
.model flat,stdcall           ;这里我们用stdcall 就是函数参数 压栈的时候从最后一个开始压，和被调用函数负责清栈  
option casemap:none            ;区分大小写  
  
includelib msvcrt.lib          ;这里是引入类库 相当于 #include<stdio.h>了         
printf  PROTO C:DWORD,:VARARG  ;这个就是声明一下我们要用的函数头，到时候 汇编程序会自动到msvcrt.lib里面找的了   
                                ;:VARARG 表后面的参数不确定 因为C就是这样的printf(const char \*, ...);  
                               ;这样的函数要注意 不是被调用函数负责清栈 因为它本身不知道有多少个参数  
                               ;而是有调用者负责清栈  下面会详细说明  
.data  
szTextFmt  BYTE '%d',0        ;这个是用来类型转换的，跟C的一样,字符用字节类型  
a          dword 1000         ;假设  
b          dword 2000         ;处理数值都用双字 没有int 跟long 的区别  
  
;/////////////////////////////////////////////////////////////////////////////////////////  
.code  
  
\_test proc ;A:DWORD,B:DWORD   
      push ebp  
      mov  ebp,esp  
      mov  eax,dword ptr ss:[ebp+8]  
      add  eax,1  
      mov  edx,dword ptr ss:[ebp+0Ch]  
      add  edx,100  
      add  eax,edx  
      pop  ebp        
      retn 8  
\_test endp  
  
\_main proc   
      push dword ptr ds:b       ;反汇编我们看到的b就不是b了而是一个[\*\*\*\*\*]数字 dword ptr 就是我们在ds(数据段)把[\*\*\*\*\*]  
                                ;开始的一个双字长数值取出来  
      push dword ptr ds:a       ;跟她对应的还有 byte ptr \*\*\*\*就是取一个字节出来 比如这样 mov  al,byte ptr ds:szTextFmt   
                                ;就把 % 取出来 而不包括 d  
      call \_test                    
      push eax                  ;假设push eax的地址是×××××  
      push offset szTextFmt  
      call printf  
      add  esp,8  
      ret               
\_main endp  
end  \_main

### 栈对齐方式

1. 栈是4个栈帧对齐的，1个栈帧是32位，4个字节， 4个栈帧是128位，16个字节
2. 当进入方法入口的时候，如果方法里面 有 3个局部变量，那么根据对齐的方式需要 3\*4=12 个字节，但是 实际申请的时候是申请12+64 =76 个字节

多出这64个字节的作用是 防止栈溢出，和一些结构化异常的处理代码

### 关于add esp，8 的解释

1. 由于平衡堆栈可以在调用的方法内平衡，也可以由调用者平衡
2. 平衡堆栈的时候可以使用push pop相对应，也可以使用add esp，x 16进制 ，采用add 进行平衡堆栈的原因是，如果压入栈的参数很多，那么pop的字节数就很多，这个采用release的方式的时候就没有使得程序最优化。
3. 采用add还是pop是在编译阶段由编译器动态决定的。由于压入栈中的参数是由方法调用的，所以可以由方法自己平衡堆栈。
4. 压入栈的dubug 时，由于栈对齐，和调试的愿意，所以需要申请占空间的 大小应该是

申请空间= x \* 4 + 64；

64也是4的倍数

### 计算机存储FLOAT类型

27.5的二进制为11011.1

将二进制 转换成科学计数法 也就是小数点左移4位

1.10111\*2^4

尾数（小数点后的数）10111，补够23位 1011 1000 0000 0000 0000 000

指数：4，加上127，就是131，二进制1000 0011

用二进制表示就是 (符号数位1位)0 （指数位8位）1000 0011 （尾数位23位）1011 1000 0000 0000 0000 000

写成二进制标准形式：0100 0001 1101 1100 0000 0000 0000 0000

写成16进制就是41 DC 00 00

### Rep和 [stosd](http://blog.csdn.net/u010874011/article/details/38070277) 的区别

Rep是重复指令，重复的次数由ecx寄存器决定

stosb指令，将AL寄存器的值取出来赋给edi所指向的地址处。mov [edi]，AL；edi=edi+1；

stosw指令去的是一个字。

stosd指令，取得是双字节，mov [edi]，eax；edi=edi+4；

.text:00401030                   \_main\_0 proc near             ; CODE XREF: \_mainj

.text:00401030

.text:00401030                   var\_44= byte ptr -44h

.text:00401030                   var\_4= dword ptr -4

.text:00401030

.text:00401030 55                push    ebp

.text:00401031 8B EC             mov     ebp, esp

.text:00401033 83 EC 44          sub     esp, 44h

.text:00401036 53                push    ebx

.text:00401037 56                push    esi

.text:00401038 57                push    edi

.text:00401039 8D 7D BC          lea     edi, [ebp+var\_44]

.text:0040103C B9 11 00 00 00    mov     ecx, 11h

.text:00401041 B8 CC CC CC CC    mov     eax, 0CCCCCCCCh

.text:00401046 F3 AB             rep stosd

注意的是

1. 有sub esp，44 申请栈空间的汇编指令
2. 存在mov ecx，11h 则下方应该会有rep指令
3. 存在lea edi，[ebp+var\_44] 则下方应该会有stosd 将eax指令指向的值 初始化对应的地址处，eax里面存储的值是0XCCCCCCCh, edi=ebp-44；

循环之后

Rep指令执行时 ebp=ebp+4； 即 Edi =ebp-40；

循环的次数是11h 即 17次，44h =68 =17\*4；

这段代码的含义是被调用的方法申请一段栈空间并使用 cc初始化内存地址

使用cc初始化内存地址的含义是

1. 如果正常访问程序初始化的内存空间，因为对应地址在初始化的过程中已经被覆盖为有效的16进制数值，所以访问正常，而由于未初始化的地址仍然是 cc，而 cc在 程序中又被称为 软中断，如果没有结构化异常处理SHE，那么程序就会调用系统的程序异常处理。所以程序会弹出 程序异常的提示框。

### 关于AFX\_PMSG类型的解释

在上述结构的六个域中，pfn是一个指向CCmdTarger成员函数的指针。函数指针的类型定义如下：  
  
typedef void (AFX\_MSG\_CALL CCmdTarget::\*AFX\_PMSG)(void);

其中typedef 表示 定义类型的关键字

第一个void 表示函数的返回值是void

第二个void表示函数的参数是void

AFX\_MSG\_CALL CCmdTarget：： \* AFX\_PMSG的含义是

AFX\_PMSG是 AFX\_MSG\_CALL类型的指针 AFX\_MSG\_CALL \*

并且AFX\_PMSG是 CCmdTarget 结构的一个成员

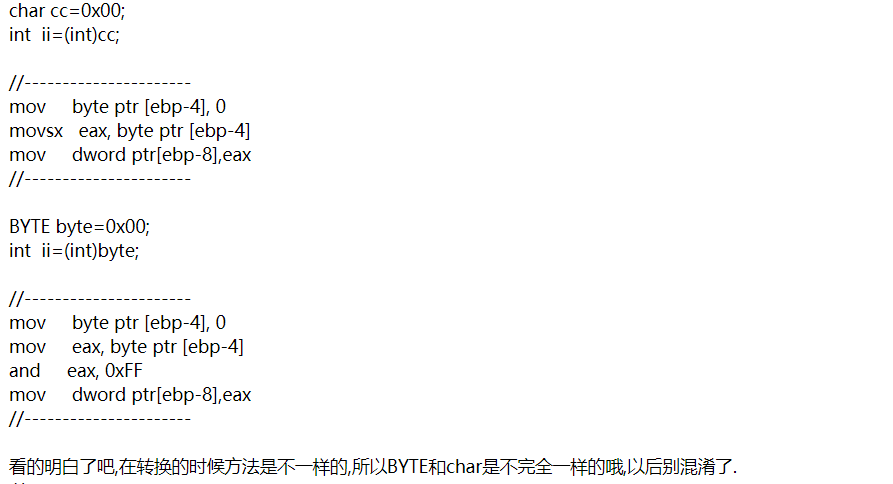
也就是说AFX\_PMSG的类型是指针类型，同时AFX\_PMSG这个指针类型也被定义在CCmdTarget这个结构体中。

### 关于反汇编中的变量的说明

013E3424  mov         eax,dword ptr [c]   
013E3427  add         eax,dword ptr [i]   
013E342A  mov         dword ptr [c],eax

实际中出现的是 相应的地址，但是分析的时候，因为为了方便理解所以可以使用变量代替，ollydbg和ida中都可以设置相应的变量标志。

### C++中char 和byte 类型的区别



      byte 是字节数据类型 ，是有符号型的，占1 个字节；大小范围为-128—127 。char 是字符数据类型 ，是无符号型的，占2字节(Unicode码 ）；大小范围 是0—65535 ；char是一个16位二进制的Unicode字符，JAVA用char来表示一个字符 。

### 汇编语言中SAR和SHR指令的区别

汇编语言中SAR和SHR指令都是右移指令，SAR是算数右移指令（shift arithmetic right），而SHR是逻辑右移指令（shift logical right）。

两者的区别在于SAR右移时保留操作数的符号，即用符号位来补足，而SHR右移时总是用0来补足。

例如10000000算数右移一位是11000000，而逻辑右移一位是01000000。

相关文章:

### Word文件格式

Word文件虽然是以docx格式结尾，但是可以在docx结尾后边加上.zip结尾，也就是将word文件变成了一个压缩包文件。

值得注意的是 当word文件为空文件的时候，会出现压缩包错误，只有当word文件非空时才能解压压缩包。

压缩包的文件格式文本文件是xml格式的，微软扩展为office open xml 所以简称为ooxml，压缩包中存在单独的图片文件夹保存当前文档中的所有的图片。

猜想

1. 由于docx文档实际上是zip压缩包，所以是否可以在压缩包中放置一些程序。当出现点击事件或者是自动运行程序可以启动。如果程序无法直接启动可以采用脚本进行二次启动。
2. 是否可以动态的修改word文档里面的内容，如使用一个脚本替换word文档中的图片或者是文字。

来看看C++标准库的一些内置数据类型:  
  
string：　这个也是以类的形式实现的．　string str("test"); string str2(str); string str; 这三种方式都是利用string的构造函数进行赋值的.   
string str ="test" ; 这是利用运算符重载进行赋值的.  
const char \*cstr = str.c\_str(); 返回c字符数组. 必须是const.   
vector : 这个是以模板和类的形式实现的.下一节具体分析.

.NET Reflector 查看函数的引用关系

点击某个函数邮件 analayz 分析 可以查看当前函数的引用函数的情况和被引用的情况。

### SSDT 和 SSPT

SSDT：系统服务调度表（System Service Dispatch Table），该表可以基于系统服务  
编号进行索引，来定位函数内存地址。  
SSPT：系统服务参数表（System Service Parameter Table），指定系统服务函数的  
参数字节数。  
系统有 2 个 SSDT 表， 一个是 KeServiceDescriptorTable(ntoskrnl.exe 导出)，一个是  
KeServieDescriptorTableShadow(ntoskrnl.exe 未 导 出 ) 。 两 者 的 区 别 是 ，  
KeServiceDescriptorTable 仅 有 ntoskrnel.exe 中 的 函 数 一 项 ，  
KeServieDescriptorTableShadow 包含了 ntoskrnel.exe 以及 win32k.sys 中包含的函数。  
一般的 ntdll.dll 中的 Native API 的函数地址由 KeServiceDescriptorTable 分派，  
gdi.dll、 user.dll 的内核 API 调用服务地址由 KeServieDescriptorTableShadow 分派。

### TEST EAX,EAX 和 跳转指令的组合用法

1. test eax，eax

je XXXXXXXX 判断依据 ZF=0 then jump

含义 测试eax的值是否为0，为0则进行跳转。

1. test eax，eax

jl XXXXXXXX 判断依据 SF ^OF 两者与操作

### 符号标志SF

(Sign Flag)

符号标志SF用来反映运算结果的符号位，它与运算结果的最高位相同。运算结果为正数时，SF的值为0，否则其值为1。

### 溢出标志OF

(Overflow Flag)

运算结果超过当前运算位数所能表示的范围，OF的值被置为1，否则，OF的值被清为0

含义 测试eax，如果eax不为0，如果eax的最高位的值为1

则跳转。

相关的例子是 eax存着枚举值 as =00 ad= 01 af=10 ad=11;

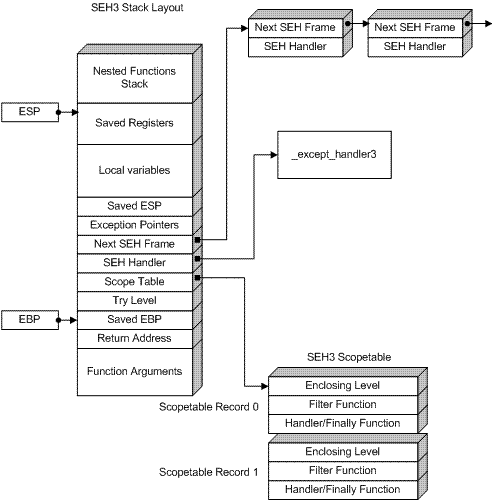
如果eax的值是af，ad，则发生跳转。

1. test eax,eax

jle XXXXXXXX 跳转条件 （SF^OF）| ZF

含义 如果 最高位为1 或者 eax为0 则跳转。

### 内存中c++函数的地址结构图



Scope table 范围表 用来表示 try结构的

Enclosing level 封闭等级

更多的SEH实现细节请看[1]。为了恢复try块，请注意观察try块的层次变量是如何更新的。每一个try块都分配了一个唯一的数作为标识，scopetable表中条目（entry）间的关系则描述了try块的嵌套关系。例如，如果scopetable的第i项的EnclosingLevel等于j，则表示try块j包围了try块i。 函数体自身被认为拥有级别-1。请参看附录1作为例子。