!Search to the Rescue!

ntdebug 13 Apr 2009 1:19 PM **6**

My name is **Trey Nash** and I am an Escalation Engineer on the Core OS team. My experience is as a software developer, and therefore, my blog posts tend to be slanted in the direction of helping developers during the feature development, testing, and the support phases.

Windbg is definitely a feature-rich debugger. Sometimes, reading the debugger help during idle time can provide some great insight into the capabilities of the debugger. However, the debugger help comes up short when you ask questions such as, "command *huh-huh* sure is cool, but when would I ever want to do that?!?" Besides, if you're caught reading the windbg help in your spare time, you may be on the receiving end of some ridicule form those in your social circle. In this post, I would like to speak a bit about the **!search** command, among others, and when you would want to use it. Additionally, I'll be demonstrating some related techniques germane to when you would use **!search** in the first place.

Not long ago, I was working with a dump from a machine that was hung and it was my job to find out why. After applying many of the techniques in **our hang dump blog post**, I discovered that there was a thread in particular that was stuck, which I show below:

```
0: kd> !thread fe016330
THREAD fe016330  Cid 0004.02e0  Teb: 00000000 Win32Thread: 00000000 WAIT: (Unknown) KernelMode Non-Alertable
    fcf1a698  SynchronizationEvent
    fe0163a8  NotificationTimer
IRP List:
    fcb47650: (0006,01d8) Flags: 00000404  Mdl: 00000000
Not impersonating
DeviceMap                 e18008e8
Owning Process            fe790648       Image:         System
Attached Process          N/A            Image:         N/A
Wait Start TickCount      75337682       Ticks: 145 (0:00:00:02.265)
Context Switch Count      31848752
UserTime                  00:00:00.000
KernelTime                00:25:38.000
Start Address Treyresearch (0xf45629e0)
Stack Init f50e1000 Current f50e05e8 Base f50e1000 Limit f50de000 Call 0
Priority 15 BasePriority 15 PriorityDecrement 0
ChildEBP RetAddr  Args to Child
f50e0600 e103d5b1 fe016330 fe0163d8 00000000 nt!KiSwapContext+0x26 (FPO: [Uses EBP] [0,0,4])
f50e062c e103df9e fe016330 fd0321f8 00000000 nt!KiSwapThread+0x2e5 (FPO: [0,7,0])
f50e0674 e101e05b fcf1a698 0000001b 00000000 nt!KeWaitForSingleObject+0x346 (FPO: [5,13,4])
f50e06b0 e102e00a e3faf6e0 f50e0900 00000000 nt!ExpWaitForResource+0xd5 (FPO: [0,5,4])
f50e06d0 f5a988cb fd0321f8 00000001 f50e08e4 nt!ExAcquireResourceExclusiveLite+0x8d (FPO: [2,3,0])
f50e06e0 f5ad81c4 f50e0900 e3faf6e0 00000001 Ntfs!NtfsAcquirePagingResourceExclusive+0x20 (FPO: [3,0,0])
f50e08e4 f5ad8909 f50e0900 fcb47650 fdcc3020 Ntfs!NtfsCommonCleanup+0x193 (FPO: [SEH])
f50e0a54 e1040153 fe00d718 fcb47650 fcb47650 Ntfs!NtfsFsdCleanup+0xcf (FPO: [SEH])
f50e0a68 f5b4fd28 fddbc818 fe6d1a28 00000000 nt!IofCallDriver+0x45 (FPO: [0,0,4])
f50e0a94 e1040153 fdcc3020 fcb47650 fcb47650 fltmgr!FltpDispatch+0x152 (FPO: [2,6,0])
f50e0aa8 f5b4fb25 fdde0cb0 fcb47650 fdd8dc18 nt!IofCallDriver+0x45 (FPO: [0,0,4])
f50e0acc f5b4fcf5 f50e0aec fdde0cb0 00000000 fltmgr!FltpLegacyProcessingAfterPreCallbacksCompleted+0x20b (FPO: [3,4,4])
f50e0b04 e1040153 fdde0cb0 fcb47650 fcb47650 fltmgr!FltpDispatch+0x11f (FPO: [2,6,0])
f50e0b18 e112ec0a fdfd9bd8 fe774730 fdfd9bf0 nt!IofCallDriver+0x45 (FPO: [0,0,4])
f50e0b48 e112b6af fe790648 fdde0cb0 00010003 nt!IopCloseFile+0x2ae (FPO: [5,7,0])
f50e0b78 e112b852 fe790648 00000001 fe774730 nt!ObpDecrementHandleCount+0xcc (FPO: [4,2,4])
f50e0ba0 e112b776 e1802e48 fdfd9bf0 00006e54 nt!ObpCloseHandleTableEntry+0x131 (FPO: [5,1,0])
f50e0be4 e112b7c1 00006e54 00000000 f50e0c00 nt!ObpCloseHandle+0x82 (FPO: [2,7,4])
f50e0bf4 e1033bdf 00006e54 f50e0cfc e103b00c nt!NtClose+0x1b (FPO: [1,0,0])
f50e0bf4 e103b00c 00006e54 f50e0cfc e103b00c nt!KiFastCallEntry+0xfc (FPO: [0,0] TrapFrame @ f50e0c00)
f50e0c70 f4562119 00006e54 00030000 00000068 nt!ZwClose+0x11 (FPO: [1,0,0])
WARNING: Stack unwind information not available. Following frames may be wrong.
f50e0cfc f456229f f50e0d34 f50e0d2c f4577f50 Treyresearch+0x11119
f50e0d38 f45626f9 fe016330 fc825368 00000000 Treyresearch+0x1129f
f50e0d70 f45629ae f1ed8000 00002000 00000000 Treyresearch+0x116f9
f50e0d90 f4562ba3 fc825318 fde59b38 00000003 Treyresearch+0x119ae
f50e0dac e1120833 f4577e20 00000000 00000000 Treyresearch+0x11ba3
```

```
f50e0ddc e103fe9f f45629e0 f4577e20 00000000 nt!PspSystemThreadStartup+0x2e (FPO: [SEH])
00000000 00000000 00000000 00000000 00000000 nt!KiThreadStartup+0x16
```

> Note: Eagle-eye readers may have noticed that the debugger states every frame in the above thread uses **frame pointer optimization (FPO)**. This is a bug in version 6.11.0001.402 of the debugger.

I have highlighted the interesting bits above. It seems that this thread is some sort of worker thread, probably created by the Treyresearch driver. It is doing some work that includes closing a particular file. In the process of closing the file, NTFS wants to acquire the paging resource for this particular file, and that is where this thread gets stuck.

What is the paging resource? Many file systems have a per-file lock that one acquires when performing paging I/O such that other destabilizing activity cannot occur at the same time as a paging operation. The paging resource for the file is this lock.

To further illustrate the paging resource, let's check out the file in question. One handy things that you can do is follow the stack down to where you see the most recent call to nt!IofCallDriver. You can see in the MSDN documentation that **IoCallDriver** accepts two parameters, a **DEVICE_OBJECT*** and an **IRP***. However, nt!IofCallDriver is a **fastcall** function, so you cannot find its parameters on the stack. But since you know that nt!IofCallDriver is calling a driver dispatch routine, and since driver dispatch routines have the same prototype as IoCallDriver, you can easily find the IRP in question which I have highlighted in the Ntfs!NtfsFsdCleanup frame of the thread's stack above and dumped out below:

```
0: kd> !irp fcb47650
Irp is active with 10 stacks 10 is current (= 0xfcb47804)
 No Mdl: No System Buffer: Thread fe016330:  Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

                     Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

                     Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

                     Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

                     Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

                     Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

                     Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

                     Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

                     Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

                     Args: 00000000 00000000 00000000 00000000
>[ 12, 0]   0  0 fe00d718 fdfd9bf0 00000000-00000000
              \FileSystem\Ntfs
                     Args: 00000000 00000000 00000000 00000000
```

And from the IRP above, we can find the real file that the thread above is trying to acquire the paging lock for:

```
0: kd> !fileobj fdfd9bf0

\Program Files\Treyresearch\Treyresearch.data

Device Object: 0xfe6da738   \Driver\Ftdisk
Vpb: 0xfe791818
Access: Read Write Delete SharedRead SharedWrite SharedDelete

Flags:  0x43062
       Synchronous IO
       Sequential Only
       Cache Supported
       Modified
       Size Changed
       Handle Created

File Object is currently busy and has 0 waiters.

FsContext: 0xe3faf7a8      FsContext2: 0xe3faf8f0
Private Cache Map: 0xfccf1fa0
CurrentByteOffset: 6400164
Cache Data:
  Section Object Pointers: fc956f3c
  Shared Cache Map: fccf1ec8         File Offset: 6400164
  Vacb: fe77bd80
  Your data is at: cbe80164
```

The file object contains two fields named **FsContext** and **FsContext2** shown above. These fields are for the file system to store file system specific information. Most file systems would store the paging resource in these context fields somewhere. For example, NTFS uses FsContext to hold the stream control block (SCB) and you can surmise that somewhere down in the SCB is where NTFS stores the paging resource. (It's actually more complicated than that, but that's good enough for sake of this discussion)

Now, let's take a look at the paging resource itself. You can see from the documentation of **ExAcquireResourceExclusiveLite**, the first parameter is an **ERESOURCE** and I have highlighted it in our thread stack above. Given that, we can use the **!locks** command to get a better idea of what's going on:

```
0: kd> !locks -v fd0321f8

Resource @ 0xfd0321f8    Shared 1 owning threads
    Contention Count = 2
    NumberOfSharedWaiters = 1
    NumberOfExclusiveWaiters = 1
     Threads: fe77f1e0-07<*>      ← This thread is the owner

     THREAD fe77f1e0  Cid 0004.0064  Teb: 00000000 Win32Thread: 00000000 WAIT: (Unknown) KernelMode Non-Alertable
         fc825320  NotificationEvent
     Not impersonating
     DeviceMap                 e18008e8
     Owning Process            fe790648       Image:         System
     Attached Process          fced2d88       Image:         store.exe
     Wait Start TickCount      75281842       Ticks: 55985 (0:00:14:34.765)
     Context Switch Count      4440231
     UserTime                  00:00:00.000
     KernelTime                00:01:51.171
     Start Address nt!MiMappedPageWriter (0xe101962c)
     Stack Init f6137000 Current f61366ac Base f6137000 Limit f6134000 Call 0
     Priority 17 BasePriority 8 PriorityDecrement 0
     ChildEBP RetAddr
     f61366c4 e103d5b1 nt!KiSwapContext+0x26 (FPO: [Uses EBP] [0,0,4])
     f61366f0 e103df9e nt!KiSwapThread+0x2e5 (FPO: [0,7,0])
     f6136738 f4562e28 nt!KeWaitForSingleObject+0x346 (FPO: [5,13,4])
WARNING: Stack unwind information not available. Following frames may be wrong.
     f6136760 f4563229 Treyresearch+0x11e28
     f6136788 f4559d8f Treyresearch+0x12229
     f61367e4 f4570b95 Treyresearch+0x8d8f
     f613684c f4570e39 Treyresearch+0x1fb95
     f6136898 f4570f4b Treyresearch+0x1fe39
     f61368c4 f5b4cb73 Treyresearch+0x1ff4b
     f613692c f5b4efc2 fltmgr!FltpPerformPostCallbacks+0x1c5 (FPO: [1,17,4])
     f6136940 f5b4f4f1 fltmgr!FltpProcessIoCompletion+0x10 (FPO: [1,0,0])
     f6136950 f5b4fb83 fltmgr!FltpPassThroughCompletion+0x89 (FPO: [3,0,4])
     f6136980 f5b4fcf5 fltmgr!FltpLegacyProcessingAfterPreCallbacksCompleted+0x269 (FPO: [3,4,4])
     f61369b8 e1040153 fltmgr!FltpDispatch+0x11f (FPO: [2,6,0])
     f61369cc f452b2f8 nt!IofCallDriver+0x45 (FPO: [0,0,4])
     f6136a18 f452b6d3 exifs!NtSystemWrite+0x1ff (FPO: [Non-Fpo])
     f6136ab0 f452ae1d exifs!IfsInternalWrite+0x1a0 (FPO: [Non-Fpo])
     f6136b24 f4549e02 exifs!MRxIfsWrite+0x333 (FPO: [Non-Fpo])
     f6136b44 f4541a8e exifs!RxLowIoSubmit+0x180 (FPO: [Uses EBP] [2,2,4])
     f6136b54 f45427ed exifs!RxLowIoWriteShell+0x2e (FPO: [1,0,1])
     f6136c64 f452fbe3 exifs!RxCommonWrite+0xcc1 (FPO: [Non-Fpo])
     f6136cf8 f453dffd exifs!RxFsdCommonDispatch+0x2c4 (FPO: [Non-Fpo])
     f6136d24 f452439a exifs!RxFsdDispatch+0x93 (FPO: [Non-Fpo])
     f6136d40 e1040153 exifs!MRxIfsFsdDispatch+0x6c (FPO: [Non-Fpo])
     f6136d54 e101c5b4 nt!IofCallDriver+0x45 (FPO: [0,0,4])
     f6136d68 e101971d nt!IoAsynchronousPageWrite+0xd0 (FPO: [8,0,4])
     f6136dac e1120833 nt!MiMappedPageWriter+0x12e (FPO: [1,4,0])
     f6136ddc e103fe9f nt!PspSystemThreadStartup+0x2e (FPO: [SEH])
     00000000 00000000 nt!KiThreadStartup+0x16

fe78eb40-01

     THREAD fe78eb40  Cid 0004.001c  Teb: 00000000 Win32Thread: 00000000 WAIT: (Unknown) KernelMode Non-Alertable
         fcea3890  Semaphore Limit 0x7fffffff
         fe78ebb8  NotificationTimer
     Not impersonating
     DeviceMap                 e18008e8
     Owning Process            fe790648       Image:         System
     Attached Process          N/A            Image:         N/A
     Wait Start TickCount      75337718       Ticks: 109 (0:00:00:01.703)
     Context Switch Count      3480314
     UserTime                  00:00:00.000
     KernelTime                00:01:35.875
     Start Address nt!ExpWorkerThread (0xe102da4b)
     Stack Init f60ef000 Current f60eebf0 Base f60ef000 Limit f60ec000 Call 0
     Priority 14 BasePriority 13 PriorityDecrement 1
     ChildEBP RetAddr
     f60eec08 e103d5b1 nt!KiSwapContext+0x26 (FPO: [Uses EBP] [0,0,4])
     f60eec34 e103df9e nt!KiSwapThread+0x2e5 (FPO: [0,7,0])
     f60eec7c e101e05b nt!KeWaitForSingleObject+0x346 (FPO: [5,13,4])
     f60eecb8 e1024ba8 nt!ExpWaitForResource+0xd5 (FPO: [0,5,4])
     f60eecd8 f5a98915 nt!ExAcquireResourceSharedLite+0xf5 (FPO: [2,3,4])
     f60eece8 f5ae198a Ntfs!NtfsAcquirePagingResourceShared+0x20 (FPO: [3,0,0])
     f60eed04 e1044997 Ntfs!NtfsAcquireScbForLazyWrite+0x7a (FPO: [2,0,0])
     f60eed40 e104328e nt!CcWriteBehind+0x27 (FPO: [0,8,4])
     f60eed80 e102db08 nt!CcWorkerThread+0x15a (FPO: [SEH])
     f60eedac e1120833 nt!ExpWorkerThread+0xeb (FPO: [1,5,0])
     f60eeddc e103fe9f nt!PspSystemThreadStartup+0x2e (FPO: [SEH])
     00000000 00000000 nt!KiThreadStartup+0x16

     Threads Waiting On Exclusive Access:
```

```
                fe016330
```

Now this is some juicy output. I used the –v option to also expand some of the threads related to this lock. The owner thread is the one with the asterisk (*) next to it and you can see from the expanded thread listing and based on the fact that the function at the bottom of the stack is nt!MiMappedPageWriter, that the thread in question is the **mapped page writer**. This thread is a system thread that periodically sweeps through a list of dirty pages flushing them out to disk. Interestingly, the mapped page writer has acquired the ERESOURCE seven times. That is shown by the -07 next to the owner thread in the above output. The second thread is waiting for shared access and it is a system file cache thread. And finally, the third thread is our initial hung thread that is waiting on exclusive access.

As a sanity check, let's make sure that the ERESOURCE and the file in question are related. Previously, I stated that in the NTFS file system the FsContext field of the file object contains an SCB. Let's pass that pointer to !pool and get some more information about it:

```
0: kd> !pool 0xe3faf7a8 2
Pool page e3faf7a8 region is Paged pool
*e3faf6d8 size:  330 previous size:   20  (Allocated) *Ntff
            Pooltag Ntff : FCB_DATA, Binary : ntfs.sys
```

Now, we can use the search command (**s**), to search the pool memory above and see if our ERESOURCE is in there. If so, that would satisfy our sanity check:

```
0: kd> s -d e3faf6d8 L 330/4 fd0321f8
e3faf72c  fd0321f8 0c9013aa 01c9969b 42002f46  .!.........F/.B
e3faf7b4  fd0321f8 06410000 00000000 06400164  .!....A.....d.@.
```

Now that we are satisfied that we have matched up the ERESOURCE with the file that owns it, let's move on. At first glance of the mapped writer thread, it looks like the offending entity is exifs. After all, it's the most interesting component on the mapped page writer stack. But don't be fooled. What you see in the mapped page writer stack is a snapshot of what it was doing when the dump was taken, and that's not necessarily the work that caused things to go bad in the first place. Even though exifs is a file system, it is not NTFS. And we know an NTFS file's paging resource is locked. Keep in mind that the mapped page writer is processing a list. So the items on the list that have caused the contention may have long been taken off the list and processed.

So what do you do? Unfortunately, the badness happened some time ago. We don't have a stack to look at to show who did this and when. But what we can do is perform a search of memory to see if there are any references to the ERESOURCE elsewhere in memory. If we find some hits, maybe they will shed some more light on what is going on. So, let's go ahead and do that:

```
0: kd> !search fd0321f8
Searching PFNs in range 0000000B - 000DFFF9 for [FFFFFFFFFD0321F8 - FFFFFFFFFD0321F8]

Pfn      Offset   Hit      Va       Pte
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
00007AB7 000004E8 FD0321F8 FCAB74E8 C03F2ADC
       fcab74e0+0x8     : Ntfr  -- ERESOURCE
00007CDF 00000950 FD0321F8 FCCDF950 C03F337C
       fccdf938+0x18    : NpFc  -- CCB, client control block - Process: fdd70248
0000803A 0000022C FD03A1F8 FD03A22C C03F40E8
       fd03a220+0xc     : Vad   -- Mm virtual address descriptors
0000977F 0000003C FD0321F8 FE77F03C C03F9DFC
       fe77f000+0x3c    : MmWe  -- Work entries for writing out modified filesystem pages.
0000977F 00000514 FD0321F8 FE77F514 C03F9DFC
       fe77f4d8+0x3c    : MmWe  -- Work entries for writing out modified filesystem pages.
0000977F 000005BC FD0321F8 FE77F5BC C03F9DFC
       fe77f580+0x3c    : MmWe  -- Work entries for writing out modified filesystem pages.
0000977F 00000904 FD0321F8 FE77F904 C03F9DFC
       fe77f8c8+0x3c    : MmWe  -- Work entries for writing out modified filesystem pages.
0000977F 00000BA4 FD0321F8 FE77FBA4 C03F9DFC
       fe77fb68+0x3c    : MmWe  -- Work entries for writing out modified filesystem pages.
0000978A 00000324 FD0321F8 FE78A324 C03F9E28
       fe78a2e8+0x3c    : MmWe  -- Work entries for writing out modified filesystem pages.
0000978A 0000051C FD0321F8 FE78A51C C03F9E28
       fe78a4e0+0x3c    : MmWe  -- Work entries for writing out modified filesystem pages.
00060B7B 0000072C FD0321F8 E3FAF72C C038FEBC
       e3faf6d8+0x54    : Ntff  -- FCB_DATA
00060B7B 000007B4 FD0321F8 E3FAF7B4 C038FEBC
       e3faf6d8+0xdc    : Ntff  -- FCB_DATA
000D9653 00000638 FD0321F8 F50E0638 C03D4380
000D9653 00000694 FD0321F8 F50E0694 C03D4380
000D9653 000006D8 FD0321F8 F50E06D8 C03D4380
000DE415 0000054C F50321F8 F503254C C03D40C8
000DFBF4 00000C40 FD0321F8 F60EEC40 C03D83B8
000DFBF4 00000C9C FD0321F8 F60EEC9C C03D83B8
000DFBF4 00000CE0 FD0321F8 F60EECE0 C03D83B8
Search done.
```

One thing to note is that there are some hits that are not exactly what we were looking for. That's because **!search** also looks for values that are one bit off from what you requested. Check out the help for how you can adjust this behavior. Also, **!search** performs some extra work along the way. If it notices that the virtual address found is in the pool, it displays information about that pool entry.

Do you spot the curiosity? Remember that the ERESOURCE had been acquired seven times. Correspondingly, there are seven hits in the **!search** list with the MmWe tag! And not surprisingly, the description of that tag pulled from pooltag.txt file in the triage directory where the debugger is installed reveals that these are paging work entries. Now we're on to something.

> Note: Incidentally, if you want to determine where a virtual address in the list above resides, you can always pass it to **!address**. The addresses at the end of the !seach output are often addresses on some thread's stack. If you pass those addresses to **!thread**, it is smart enough to find the thread that is associated with that stack and display it for you.

"OK, but how do I find the real culprit?", you then say. Well, you have to continue to dig with what you have. Unfortunately, the contents of the pool entries with MmWe tags are not documented, although, we definitely know what they are. Let's take a look at one of them using **!pool**:

```
0: kd> !pool FE77F03C
Pool page fe77f03c region is Nonpaged pool
*fe77f000 size:   a8 previous size:    0  (Allocated) *MmWe
            Pooltag MmWe : Work entries for writing out modified filesystem pages., Binary : nt!mm
```

Now, we see where the pool entry starts and how big the entry is. So, let's take a look at the contents of the memory:

```
0: kd> dps fe77f000 L a8/@$ptrsize
fe77f000  0a150000
fe77f004  65576d4d
fe77f008  fe78a4e8
fe77f00c  e10b3af0 nt!MmMappedPageWriterList
fe77f010  06140000
fe77f014  00000000
fe77f018  06150000
fe77f01c  00000000
fe77f020  fcc20788
fe77f024  00000000
fe77f028  e10b3f20 nt!MmMappedFileHeader
fe77f02c  00000000
fe77f030  00000000
fe77f034  fdfd9bf0
fe77f038  fdc6f008
fe77f03c  fd0321f8
fe77f040  00000000
fe77f044  00000000
fe77f048  00000000
fe77f04c  0002005c
fe77f050  00000000
fe77f054  f1d20000
fe77f058  00000000
fe77f05c  00010000
fe77f060  00000000
fe77f064  0007036c
fe77f068  000cb59d
fe77f06c  000a97fe
fe77f070  0000d7ef
fe77f074  00021c90
fe77f078  0005d6b1
fe77f07c  000a5642
fe77f080  0004d5c3
fe77f084  000ae354
fe77f088  00038249
fe77f08c  000509ea
fe77f090  0009c915
fe77f094  00018dd6
fe77f098  000b94d7
fe77f09c  0006ca58
fe77f0a0  00091e29
fe77f0a4  00000000
```

I used the **dps** command so that it would check to see if any of the values matched to any known symbols. As you can see, there are a couple of symbols in there. Notice that I divided the size of the block by the **pseudo register $ptrsize** as well. The symbols that dps found validate that this block of memory is associated with the mapped page writer.

You may also ask yourself, "are any of these values pointers to other pool blocks?" If you wanted to know that, you could iterate over each one of them passing them to !pool or !address. That sounds tedious to do manually. But thankfully, the debugger has some nice command tokens such as **.foreach** that make this a breeze. If you want to pass each of the above values to !pool, you can perform the following in the debugger:

```
.foreach /pS 1 /ps 1 ( value { dp /c 1 fe77f000 L a8/@$ptrsize } ) { .if( value != 0 ) {.printf "**** %p ****\n", ${value}; !pool
${value} 0x2} }
```

The address highlighted above is the address of the pool block revealed by the previous !pool command. a8 is the size of the block in bytes and since dp lists the memory in units of pointer size, I divide a8 by $ptrsize.

I won't show the full output here, because it's rather verbose. But I have duplicated the output from the dps command above with added pool tags next to items that are pool entries:

```
0: kd> dps fe77f000 L a8/@$ptrsize
fe77f000  0a150000
fe77f004  65576d4d
fe77f008  fe78a4e8   Pooltag MmWe : Work entries for writing out modified filesystem pages., Binary : nt!mm
fe77f00c  e10b3af0 nt!MmMappedPageWriterList
fe77f010  06140000
fe77f014  00000000
fe77f018  06150000
fe77f01c  00000000
fe77f020  fcc20788   Pooltag Irp  : Io, IRP packets
fe77f024  00000000
fe77f028  e10b3f20 nt!MmMappedFileHeader
fe77f02c  00000000
fe77f030  00000000
fe77f034  fdfd9bf0   Pooltag File : File objects
fe77f038  fdc6f008   Pooltag MmCa : Mm control areas for mapped files, Binary : nt!mm
fe77f03c  fd0321f8   Pooltag Ntfr : ERESOURCE, Binary : ntfs.sys
fe77f040  00000000
fe77f044  00000000
fe77f048  00000000
fe77f04c  0002005c

<snip>
```

Sure enough, the file object above is identical to the file we identified earlier on as the one that the worker thread was attempting to close a handle to. Also, you can see that our ERESOURCE is in there as well. And even more, now we have an IRP that may reveal even more information. Let's see:

```
0: kd> !irp fcc20788 1
```

```
Irp is active with 10 stacks 12 is current (= 00000000)
 Mdl=fe77f048: No System Buffer: Thread fd089e6c:  Irp is completed.  Pending has been returned
Flags = 00000003
ThreadListEntry.Flink = fcc20798
ThreadListEntry.Blink = fcc20798
IoStatus.Status = 00000000
IoStatus.Information = 00010000
RequestorMode = 00000000
Cancel = 00
CancelIrql = 0
ApcEnvironment = 00
UserIosb = fe77f018
UserEvent = 00000000
Overlay.AsynchronousParameters.UserApcRoutine = e101c95b
Overlay.AsynchronousParameters.UserApcContext = fe77f008
Overlay.AllocationSize = 00000000 - 00000000
CancelRoutine = 00000000
UserBuffer = 00000000
&Tail.Overlay.DeviceQueueEntry = fcc207c8
Tail.Overlay.Thread = fd089e6c
Tail.Overlay.AuxiliaryBuffer = e101cb86
Tail.Overlay.ListEntry.Flink = 00000000
Tail.Overlay.ListEntry.Blink = 00000000
Tail.Overlay.CurrentStackLocation = 00000000
Tail.Overlay.OriginalFileObject = 00000000
Tail.Apc = 00300012
Tail.CompletionKey = 00300012
     cmd  flg cl Device   File     Completion-Context
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

                    Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

                    Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

                    Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

                    Args: 00000000 00000000 00000000 00000000
 [  4, 0]   0  0 fe755ab8 00000000 f5ce857e-fe709df8
              \Driver\Disk  PartMgr!PmIoCompletion
                    Args: 00000000 00000000 00000000 0000000c
 [  4, 0]   0  0 fe709df8 00000000 f5c16558-fe6da7f0
              \Driver\PartMgr     ftdisk!FtpRefCountCompletionRoutine
                    Args: 00000000 00000000 00000000 0000000c
 [  4, 0]   0  0 fe6da738 00000000 f5bc0638-fe6d6a80
              \Driver\Ftdisk      volsnap!VspRefCountCompletionRoutine
                    Args: 00000000 00000000 00000000 0000000c
 [  4, 0]   0  0 fe6d69c8 00000000 f5d0af28-00000000
              \Driver\VolSnap     vsp
                    Args: 00000000 00000000 00000000 0000000c
 [  4, 0]   0  0 fe6d66c0 00000000 f5a9351c-fde38cd0
              \Driver\VSP   Ntfs!NtfsSingleAsyncCompletionRoutine
                    Args: 00000000 00000000 00000000 0000000c
 [  4, 0]   0  0 fe00d718 00000000 00000000-00000000
              \FileSystem\Ntfs
                    Args: 00000000 00000000 00000000 00000000
```

What is apparent from looking at this IRP is that it is flagged as completed. Moreover, the information in the IoStatus fields looks to be relevant as well. But after studying the situation a little deeper, it appeared that the completion routine had never been fired. We can find out more about the completion routine by dumping out the Tail.Apc portion of the IRP as shown below:

```
0: kd> dt nt!_IRP fcc20788 Tail.Apc.
   +0x040 Tail      :
      +0x000 Apc       :
         +0x000 Type      : 0x12 ''
         +0x001 SpareByte0 : 0 ''
         +0x002 Size      : 0x30 '0'
         +0x003 SpareByte1 : 0 ''
         +0x004 SpareLong0 : 0
         +0x008 Thread    : 0xfe77f1e0 _KTHREAD
         +0x00c ApcListEntry : _LIST_ENTRY [ 0xfcde8ce4 - 0xfd089e6c ]
         +0x014 KernelRoutine : 0xe101cb86           void  nt!IopCompletePageWrite+0
         +0x018 RundownRoutine : (null)
         +0x01c NormalRoutine : (null)
         +0x020 NormalContext : (null)
         +0x024 SystemArgument1 : (null)
         +0x028 SystemArgument2 : (null)
         +0x02c ApcStateIndex : 0 ''
         +0x02d ApcMode   : 0 ''
         +0x02e Inserted  : 0x1 ''
```

Recall from the rules of IRP processing on Windows that **IRPS like these have their completion routines called within the thread context that initiated the I/O**. As you can see above, that thread is the same thread that is running nt!MiMappedPageWriter. Moreover, the Inserted flag is set above, which means that the APC has been placed in the APC queue for the thread. Deductive reasoning would imply that if the completion routine has not run, then the APC has not been delivered. And the APC will not be delivered if normal kernel mode APCs are disabled at the moment. So, let's check on that by looking in the

nt!KTHREAD structure:

```
0: kd> dt nt!_KTHREAD 0xfe77f1e0 KernelApcDisable
   +0x070 KernelApcDisable : -2
```

Sure enough, kernel APCs are disabled for this thread at the moment. How can that be? Well, there are several ways to disable normal kernel APC delivery and it often involves either directly or indirectly entering a **critical or guarded region**. Critical regions are entered directly via **KeEnterCriticalRegion** and guarded regions are entered via **KeEnterGuardedRegion**. However, there are several means of indirectly entering critical regions including **FsRtlEnterFileSystem**. Additionally, **holding a mutex object automatically places the holder in a critical region**.
Therefore, the root cause in this case was that the file system drivers appear to have put the thread into a state where it cannot receive APCs and, therefore, I/O initiated on that thread could not be completed. The APCs build up in the queue so that they can be delivered when kernel APC delivery is re-enabled. Incidentally, the documentation for FsRtlEnterFileSystem states that file system filter drivers should never disable normal kernel APCs across calls to IoCallDriver.

Conclusion

Many times, when it looks like you are hitting up against a brick wall in determining what went wrong in a dump, you can get a lot further than you initially expect with a little bit of intuition and the right tools. Of course, this intuition will grow as you become more and more familiar with the Windows operating system internals, or whatever platform you work on. Using **!search** to search physical memory in the dump file can help find references (a.k.a pointers) to objects in hard to find places. Additionally, pool tag information along with the helpful text in the pooltag.txt file displayed by **!pool** goes a long way when it comes to figuring out what a particular pool block is used for. Armed with all of these tools, you can always get farther than one may initially expect.
Happy debugging everyone!

Comments

zhzhtst
13 Apr 2009 10:26 PM

Great! A very good post. Thank you.

Michael G
14 Apr 2009 8:28 PM

> Eagle-eye readers may have noticed that the debugger states every frame in the above thread uses frame pointer optimization (FPO). This is a bug in version 6.11.0001.402 of the debugger.

Is that bug fixed in .404? The release notes don't mention the difference between .404 and .402.

DIV class=commentowner>[It doesn't appear to be yet. Thanks, Trey.]</DIV>

gOODiDEA.NET
14 Apr 2009 8:40 PM

Web 15 Essential Checks Before Launching Your Website High Performance Web Pages – Real World Examples

gOODiDEA
15 Apr 2009 2:18 AM

Web 15EssentialChecksBeforeLaunchingYourWebsite

HighPerformanceWebPages

Gabe
8 May 2009 2:48 PM

!fileobj seems to be available if I manually load kdexts.dll I thought it would be a very useful extension after seeing this post. However, when I run it seems that I am missing nt symbols in order to use it.

I do have pdb symbols loaded for almost everything (i.e. doing an 'lm' shows (pdb symbols) for nt and almost everything else) so I don't think its an issue of not having the public symbols loaded.

Am I missing something else?

Gabe
8 May 2009 2:56 PM

Nevermind my last comment. I am debugging a 2K target and noticed that that extension in unavailable for 2k