



Development Lifecycle » Debug Tips » General

ASM, VC6, VC7, VC7.1Win2K, WinXP, Win2003, Visual Studio, Dev, QA

Debug Tutorial Part 5: Handle Leaks

By [Toby Opferman](#)

Posted: **9 May 2004**

Views: **109,796**

Bookmarked: **109 times**

Learn how to debug handle leaks in Windows.

29 votes for this article.

Popularity: **6.84** Rating: **4.68** out of 5

Introduction

Welcome to the fifth installment of the debug tutorial series. In this article, we will go over handles in Windows, what they are and how to debug leaks. I hope that you are already familiar with the previous 4 articles, before reading this one. The intent of these articles is not to regurgitate the WinDbg/NTSD help files but rather introduce real problems, what they are and how to solve them using the debuggers.

What is a handle?

To an application, a "handle" is some instance of a device, file, or some other system object or resource. The application will create an instance of the resource by calling a function such as `CreateFile` or `RegOpenKey`, and use the handle in subsequent calls to these functions to perform operations on the resource.

In general, you never really care what this handle value is as it really is of no value to your application aside from passing it into functions that perform operations on it. So, what exactly is the handle and what does it represent?

In order to get a clearer picture, let's go over a simple `CreateFile` call.

If you debug `CreateFile`, you will notice eventually it gets to `NtCreateFile`.

```
kernel32!CreateFileW+0x34a:
77e7b24c ff150810e677 call dword ptr [kernel32!_imp__NtCreateFile (77e61008)]
0:000> kb
ChildEBP RetAddr  Args to Child
0012f728 77e7b4a3 000007c4 80000000 00000000 kernel32!CreateFileW+0x40e
0012f74c 00401ba9 00406760 80000000 00000000 kernel32!CreateFileA+0x2e
```

`NtCreateFile` results into a kernel call. How the call is actually performed varies on the system, it could be a `sysenter` instruction or a software generated interrupt `int 2eh`. Either case, a call is made into the kernel and the system dispatches the call to the correct driver. When this occurs, an object is created in the kernel which represents the requested resource, in this case a file. If you look at the parameters to `NtCreateFile`, posted [here](#), you will notice that the first parameter to `NtCreateFile` is where the handle value will be returned. Let's take a look at what was the first parameter.

```
eax=0012f730 ebx=00000000 ecx=80100080 edx=00200000 esi=77f58a3e edi=00000000
eip=77e7b24b esp=0012f69c ebp=0012f728 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000282
kernel32!CreateFileW+0x349:
77e7b24b 50                push     eax
0:000>
```

As we can see, the first parameter was an address, most likely to some local variable in the function. Upon return, we can now check the value it points to. According to the documentation, it's a file handle.

```
0:000> dd 0012f730
0012f730  000007c4
```

The returned value is 7c4h, which is NOT a pointer to any memory in your application. It is also not even a pointer in kernel memory. To further investigate, let's find what information we can gather from the handle. There is a debugger command that displays handle information. This command is called **!handle**. Let's use this command on the value and see what we get.

```
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
```

So, we are told that it is indeed a file type object, we are told the attributes and even granted access to the handle. What we aren't told in this information is the file that this handle points to. To me, that would seem to be the most valuable information that you could gather. There is a reason why we do not see this information, but before I tell you that, we must continue to figure out how the system even found what it did in the first place.

What does the value of a handle represent?

To show you this, I need to hook up the kernel debugger. I have started to debug another process on a Windows 2000 professional machine since I do not have it setup for my Windows XP box. I started Notepad and attempted to get handles in *notepad.exe*.

Tip: *In Windows XP/2003, more information is available to the system as well as displayed in the debugger. For example, !handle on a thread will display the thread entry function. Windows 2000 will not do this. This can be very useful if you want to find the matching thread to a handle, especially if the thread handle was leaked and the thread is no longer around. There is also the command dt which was shown earlier in these tutorials. Windows 2000 may have problems displaying structures that are always displayed on Windows 2003/XP (Referring to system structures like NT's _EPROCESS). This is either due to the way DBGs and PDBs are handled when debugging Windows 2000, or the information is simply missing. These and many other benefits make debugging on Windows XP/2003 more desirable than Windows 2000.*

So, I selected a file in Notepad and set break points on `CreateFile`. After I selected a file, the breakpoint hit and I simply set another breakpoint on the return address. Now, I dump `EAX` to get the handle information of the file I selected.

```
0:000> !handle eax ff
Handle 120
Type           File
Attributes     0
GrantedAccess  0x120089:
                ReadControl,Synch
                Read/List,ReadEA,ReadAttr
HandleCount    2
PointerCount   3
```

This doesn't really tell me much aside from really, it's a file. Now, I'm going to let you in on the first secret. These handles are process specific. This value, "120h", is only known from within this process space. If this handle value was sent to another process, it would either not exist or most likely be a handle to a different object. These handles are not like window handles, their scope is within a process.

Handles are also representations of kernel mode objects. This means that every process has a handle table located in kernel mode and each entry points to a kernel memory location. This is secret number two. This is why I hooked up the kernel debugger. Now, let's break into the kernel

and we will attempt to find this handle.

The first thing I do in the kernel debugger is "!process 0 0" to list all processes. Once all processes are listed, I will then use the **!handle** command. The syntax is a little different though. I will need to specify the process in order to list the correct handle.

```
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}
```

The first **bold** number is the process object. It's basically a structure in memory that contains information and locations of data specific to that process. Since handles are process specific, we need to tell the **!handle** command what process to look in. The "ff" I use simply sets all the bit fields to give me all the information it possibly can. The help will tell you what each bit represents if you want to be picky about the displayed information. I always just use "ff" so I don't need to bother with setting exact flags and I know I should get all available information.

The second bold address is the handle table for that process. This memory location specifies all entries to the handles used by that process. Notice that it currently has 74 handles open. The last bold is the object's memory location itself. This is where all the data is being pulled from. Notice that the "Name" property gives us the actual location and file name. How come we couldn't see this in the user mode debugger?

Displaying Handle Information In User Mode

Obviously, this table is located in the kernel so it cannot be seen directly from user mode. NT does provide APIs that can be used to query these handles from the kernel. If you have read my [article #3](#), the QuickView: System Explorer utility I wrote actually does this and will display the handle information for processes that it can access. The API that does this is called **NtQueryObject**.

The downfall of this API is that certain objects may hang when you attempt to query them. This is because those objects, for example, are opened with SYNC access and some of them really mean business. There are pipes opened on the system that if you attempt to query them will hang indefinitely. In order to prevent this, the debugger as well as the application I wrote attempt to prevent this by not querying objects that have the potential to cause deadlock. In my application though, I not only check for the **SYNC** flag to be set, I found some common access masks that would not hang and allow those to be queried.

There is another utility called **HANDLE.EXE** that can be downloaded from [SysInternals](#) that DOES display all information. So, how does it do that without deadlocking? They have their own kernel driver, and since all memory in system space is accessible by any driver also running in system space, it's quite simple. As shown, the kernel object's memory location can be found and simply read directly without having to acquire a system lock which deadlocks the **NtQueryObject** call. I have been thinking to add a driver to future versions of QuickView to get around the deadlocking issue and display more detailed information.

Multiple Handles

You have seen the above example on my Windows 2000 machine. I have opened **\TripItinerary.txt** file. So, what happens if I open it again with another instance of Notepad? Let's try this and see what happens.

Inside the user mode debugger:

```
0:000> !handle eax ff
Handle 58
  Type           File
  Attributes      0
  GrantedAccess  0x120089:
                 ReadControl,Synch
                 Read/List,ReadEA,ReadAttr
  HandleCount    2
  PointerCount   3
```

In the kernel debugger, I found the process and list the handle:

```
kd> !handle 58 ff fcd8ace0
processor number 0
PROCESS fcd8ace0 SessionId: 0 Cid: 0258 Peb: 7ffdf000 ParentCid: 0198
  DirBase: 04b19000 ObjectTable: fccc9648 TableSize: 22.
  Image: notepad.exe

Handle Table at ele89000 with 22 Entries in use
0058: Object: fcce7028 GrantedAccess: 00120089
Object: fcce7028 Type: (fced7c40) File
  ObjectHeader: fcce7010
    HandleCount: 1 PointerCount: 1
    Directory Object: 00000000 Name: \TripItinerary.txt {HarddiskVolume1}
```

I can also list the handle in the original Notepad process:

```
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 ele5f000 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}
```

In this case, not only did they get their own handle, but they got their own memory location in the kernel. This isn't always the case with all objects. Sometimes the kernel object can be shared across processes. If the same process opens the file, generally they get two handles but one kernel object as well. You can experiment by using the utility I wrote or SysInternals handle application to display handle information and sort them by kernel object or handle #. This can help to get you acquainted with handles.

Why did the user mode debugger show a HandleCount of 2 and a PointerCount of 3?

If you noticed, there was a "HandleCount" 2 and "PointerCount" 3 when we used the user mode debugger to show handle information. This is because it did not adjust itself when displaying the information. In order for the debugger to get the handle information, it must duplicate the handle using `DuplicateHandle`. The duplication increases the HandleCount by 1 and the PointerCount by 2. If the debugger would decrement by 1 and by 2, it could display the correct information but it chooses not to. Let's check this out and see.

What I have first done is looked at the handle `ObjectHeader: "fcd32430"` in the kernel.

```

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}

kd> dd fcd32430
fcd32430 00000001 00000001 fced7c40 00000800
fcd32440 fce55fc8 00000000 00700005 fceccbd0
fcd32450 fcecc248 e1e730d8 e1e73250 fcc790b4
fcd32460 00000000 00000000 00000000 00010000
fcd32470 00010100 00040042 00380024 e1bf2568
fcd32480 00000000 00000000 00000000 00000000
fcd32490 00000000 00040001 00000000 fcd3249c
fcd324a0 fcd3249c 00040000 00000001 fcd324ac
kd> ba r1 fcd32430
kd> ba r1 fcd32434
kd> g

```

As we can see, the handle count is 1 and the pointer count is 1. I then set "ba r1" break points on the handle count address and the pointer count address. "BA" means Break if address is accessed. "r" means if it's accessed through read or write. The 1 on "r1" simply means 1 byte.

Once I have done this and hit "g" to let the kernel continue, I will now simply type "!handle 120 ff" in the user mode debugger. This will cause the user mode debugger to access this object to find the information. Let's see when these pointers are incremented.

```

kd> kb
ChildEBP RetAddr Args to Child
fb66ebf0 8049ff0d 00000120 00000000 00000000 nt!ObReferenceObjectByHandle+0x1af
fb66ed40 80461691 00000074 00000120 ffffffff nt!NtDuplicateObject+0x12d
fb66ed40 77f83f85 00000074 00000120 ffffffff nt!KiSystemService+0xc4
0006ef28 00000000 00000000 00000000 00000000 ntdll!NtDuplicateObject+0xb

```

As we can see, our first call calls `ObReferenceObjectByHandle`. This adds a reference to the pointer. Let's see what our handle count says now.

```

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: 2
    Directory Object: 00000000 Name: \TripItinerary.txt {HarddiskVolume1}

```

As we can see, our pointer count has incremented to 2. Let's see what happens next.

```

kd> kb
ChildEBP RetAddr  Args to Child
fb66ebec 8049fffb 00000002 fcc77800 fcd32448 nt!ObpIncrementHandleCount+0x236
fb66ed40 80461691 00000074 00000120 ffffffff nt!NtDuplicateObject+0x3c3
fb66ed40 77f83f85 00000074 00000120 ffffffff nt!KiSystemService+0xc4
0006eed0 77e846ed 00000074 00000120 ffffffff ntdll!NtDuplicateObject+0xb
0006ef28 69b22188 00000074 00000120 ffffffff KERNEL32!DuplicateHandle+0xd4
0006f454 69b2252e 00000074 00000120 000000ff ntsdexts!GetHandleInfo+0x29
0006f4d0 0100d562 00000074 00000050 01001dec ntsdexts!handle+0x10f
0006f52c 0100e497 00233848 0006f54c 0006f553 ntsd!CallExtension+0x77
0006f65c 0100c2bb 01089c81 010241d8 01089138 ntsd!fnBangCmd+0x377
0006f868 0100be0f 80000003 00000001 0006fc2c ntsd!ProcessCommands+0x2ac
0006fab4 01008406 00000000 00000000 ffffffff ntsd!ProcessStateChange+0x687
0006fc2c 01008a14 0006fc4c 00000000 00000000 ntsd!DebugEventHandler+0x6a5
0006fca8 010076dc 00000000 00000000 7ffdf000 ntsd!NtsdExecution+0x13b
0006ff70 010226bf 00000002 002337b0 00232978 ntsd!main+0x3d7
0006ffc0 77e87903 00000000 00000000 7ffdf000 ntsd!mainCRTStartup+0xff
0006fff0 00000000 010225c0 00000000 000000c8 KERNEL32!BaseProcessStart+0x3d
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 ele5f000 with 74 Entries in use
0120: Object: fcd32448 GrantedAccess: 00120089
Object: fcd32448 Type: (fced7c40) File
  ObjectHeader: fcd32430
    HandleCount: 2 PointerCount: 2
    Directory Object: 00000000 Name: \TripItinerary.txt {HarddiskVolume1}

```

As we can see, this now calls `ObpIncrementHandleCount` and our handle count is now also 2. Notice that both calls are within `NtDuplicateObject` which was called from the usermode API `DuplicateHandle`. So, duplicate object will increment the reference pointer count to this kernel memory and increment the number of handles that are pointing to this kernel object. This is an instance where two processes are referencing the same kernel object. So, where does the third pointer reference come in?

```

kd> kb
ChildEBP RetAddr  Args to Child
fb66ec68 804baed8 00000100 00000000 00000000 nt!ObReferenceObjectByHandle+0x1af
fb66ed48 80461691 00000100 00000002 0006ef4c nt!NtQueryObject+0xc1
fb66ed48 77f8c4e1 00000100 00000002 0006ef4c nt!KiSystemService+0xc4
0006ef24 69b22203 00000100 00000002 0006ef4c ntdll!NtQueryObject+0xb
0006f454 69b2252e 00000000 00000120 000000ff ntsdexts!GetHandleInfo+0xa4
0006f4d0 0100d562 00000074 00000050 01001dec ntsdexts!handle+0x10f
0006f52c 0100e497 00233848 0006f54c 0006f553 ntsd!CallExtension+0x77
0006f65c 0100c2bb 01089c81 010241d8 01089138 ntsd!fnBangCmd+0x377
0006f868 0100be0f 80000003 00000001 0006fc2c ntsd!ProcessCommands+0x2ac
0006fab4 01008406 00000000 00000000 ffffffff ntsd!ProcessStateChange+0x687
0006fc2c 01008a14 0006fc4c 00000000 00000000 ntsd!DebugEventHandler+0x6a5
0006fca8 010076dc 00000000 00000000 7ffdf000 ntsd!NtsdExecution+0x13b
0006ff70 010226bf 00000002 002337b0 00232978 ntsd!main+0x3d7
0006ffc0 77e87903 00000000 00000000 7ffdf000 ntsd!mainCRTStartup+0xff
0006fff0 00000000 010225c0 00000000 000000c8 KERNEL32!BaseProcessStart+0x3d
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 ele5f000 with 74 Entries in use
0120: Object: fcd32448 GrantedAccess: 00120089
Object: fcd32448 Type: (fced7c40) File
  ObjectHeader: fcd32430
    HandleCount: 2 PointerCount: 3
    Directory Object: 00000000 Name: \TripItinerary.txt {HarddiskVolume1}

```

There it is, as I mentioned before, the debugger will need to call `NtQueryObject` in order to ask the system to tell it information about an object. The `NtQueryObject` API before attempting to read the memory will reference it by again calling `ObReferenceObjectByHandle`. This causes the

pointer count to increase to 3. However, `NtQueryObject` does not need to create another handle instance or duplicate any handles. Once it's able to get a pointer reference to the object, it simply queries the information and releases the reference count.

Another process could possibly release its handle or pointer count during this operation and could even display only the debugger's references to the object. The returned information to the debugger will be 1 extra handle reference (the one it created), and 2 extra pointer references (the one it created when duplicating the handle and the one created by `NtQueryObject` to read the object information). The debugger, as mentioned before, could always subtract this to display only relevant information, since after the `NtQueryObject` call, one pointer reference goes away and it will also close its duplicated handle. That will let the object dereference to its original state (or new state if another process also now references it).

BTW, the `DuplicateHandle` API takes a process handle from which to duplicate the handle. So, you must be able to call `OpenProcess` with duplicate privileges in order to duplicate handles in other processes, just in case you were wondering. [DuplicateHandle](#).

Debugging Handle Leaks

So, now that we have been acquainted with handles, how do we find leaks? Aside from the usual applications such as "Bounds Checker", we can find them ourselves. The first thing to do is check the application at a good time to get a baseline for handles. Task Manager has an option to display a "handles" column. This is a good place to start. Next, you should then perform operations with the application and of course, see if the handles grow or shrink. When the handles grow, you should not immediately think there's a handle leak. You must know to determine expected behavior from unexpected.

For example, if you had a server application and you were checking network connections. You notice that as you connect a lot of clients to the application, the number of network connections on the server grew. This would be expected, obviously so it would need to be determined if it's more than expected. Also, when all the clients disconnected, it would then be expected that all the network connections were cleaned up. So, if you know your handle count grows on purpose but selecting another option should then decrease the handle count, that should be tested.

Step One: Determine A Leak

If you've been monitoring the application, determine if the handle count is expected or not. See if it's growing steadily over time, fast or slow. Once you have determined this is truly a leak or are unsure at this point, it may be time to move onto the next step which would be to identify the handles being leaked. If possible, it would also be great to attempt to determine the pattern for the leak. An example would be every time a certain menu item is selected or a file is opened. This would help to narrow down reproduction steps and limit the area of the source that appears to be the location causing the problem.

Step Two: Determine The Type And Object Information

The best ways to do this would be to use a tool such as "Handle" from SysInternals and/or the debugger. I created a simple program that leaked handles very fast. I ran the application and looked in Task Manager. I found that over 65,000 handles were leaked. So, the first thing I want to do is determine what type of handle leaked. This way, I can limit down what APIs I want to look at in my application to find the leak.

```
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
```

As we can see in this exaggerated example, the handle that appears to be leaking is a "Key", which is a registry handle. Some place in this application is opening the registry. The first thing now to do is attempt to identify the location and if there are more opens on a particular key than others. If we can get the key information, we would then be able to sort out which key was opened the most if there is one, and then we can limit our search in the application.

```
!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
```

With the "Key", we get lucky. The debugger shows us the key being opened and *HKCU\Software* appears to be being opened the most. We have identified it is a key object, which means points of creation would be [RegOpenKey](#) and [RegOpenKeyEx](#). We have also now identified the actual key being opened, *HKCU\Software*. Step 3:

Step Three: Browse Source and Debug Application

We now have the type, the APIs, and luckily enough even the exact resource being opened. So now, we can do things like look through the source for locations opening this key or even setting break points in the application on [RegOpenKey](#)/[RegOpenKeyEx](#) to get stack traces. We could then keep track of the allocated handles and determine which ones were getting closed. Bounds checker could also be used as a tool in aiding this process.

Another technique which can be done but may be overkill in most cases would be to use the memory leak trick stated in the "Heap" tutorial. Wrapper functions can be written around the APIs creating the handles and you could add the handle to a linked list. Upon free, you could search the linked list and remove the handle. This operation would allow the handle itself to still be returned so you do not have to have a wrapper for all functions. The only downfall is that it would be slower since you would need to traverse the list on a close.

```
//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;
}
```

Also, remember that the above are examples, almost pseudo code. Critical sections would be used when needed and implementations can vary. The purpose of a global list is so a debugger extension (tutorial part 4) could be written to traverse the linked list and debug symbols could be used to find the location of this global to walk the list. This may be overkill for most problems and the implementation would probably only benefit if it was pre-built into all the builds without necessary implementation all the time (such as `#define MyOpenKey RegOpenKey` for retail builds).

Smaller Leaks

Not all handle leaks will appear like the above. For example, you may notice an operation is causing one or two handle leaks every time it's called. You could then break into the application, get a handle count for all values, then perform the leak and get the next snapshot. Once that occurs, you could then cause the problem again, but this time set break points on locations that would create the objects that you have seen increasing.

Other Tips

Here are some other tips when dealing with handles.

Invalid thread wait for exit looping

I have noticed many applications use a `Sleep()/GetExitCodeThread()` loop combination to wait for a thread to exit. For an example:

```
do {
    Sleep(10);
    bReturn = GetExitCodeThread(hThread, &ExitCode);
} while (bReturn && ExitCode == STILL_ACTIVE) ;
```

MSDN states, the problem with this is that the thread could return `STILL_ACTIVE` (value 259) as its return value. While that may be true, that's not the reason I would worry about this loop. A problem that I found was that some applications were using MFC's `CThread` libraries and attempting to do this loop. The problem is that `AfxBeginThread()` sends the thread handle to the

created thread. The created thread uses `AfxEndThread()` to exit, which then closes the handle. If you have not duplicated this handle, that handle is now invalid.

In general, this may not be a problem as the loop would still exit when it got the failure to use the handle. The problem occurs when another object was now created using the old thread handle! Remember, these handles are re-used by the system once they are freed. That means that another object could be assigned to the handle in the time it takes to call the function again. This object could be anything and the call may fail with invalid object. It could also be another, newly created thread which would now not let this loop exit!

So the first rules are, even outside of the `Afx*` functions, if you're going to close the handle elsewhere, be sure to duplicate it. Secondly, you don't need to loop since the thread handle becomes signaled when the thread exits.

```
WaitForSingleObject(hThread, INFINITE);
GetExitCodeThread(hThread, &ExitCode);
CloseHandle(hThread);
```

So, the above could have been better solved by the series of code modeled above.

.DMP files missing handle information?

If you have a `.DMP` file and you use `!handle` sometimes, you may receive an error. This is because if you have a `.DMP` file, you can't call `NtQueryObject` to get the handle information anymore. In that case, the debugger needed to have queried all objects and saved the information into the `.DMP` while creating it. Some `.dump` flags do not do this. What I have found is that almost all NTSD/CDB/WinDbg versions never save handle information when using `.dump /f x.dmp` (full dump). They have a separate option, `/mh` though that does. `.dump /mh x2.dmp` but it's not a full dump, it's a "mini dump". What I have done in the past is create two dumps using both options.

This is not necessary if you have a newer debugger downloaded from Microsoft's site though. There is a new option, `/ma` that creates a full dump with handle information. `.dump /ma x3.dmp` and that is all you need. This is the flag I would recommend to create all user mode dumps since you never know if you may need to look at handle information.

Conclusion

Handles are yet another part of Windows which we must learn to work with and "handle" them correctly. When checking applications you write for memory leaks and other problems, always be sure to check the handle count! Hopefully, this article helped you learn what handles are and how to investigate them.

License

This article has no explicit license attached to it but may contain usage terms in the article text or the download files themselves. If in doubt please contact the author via the discussion board below.

A list of licenses authors might use can be found [here](#)

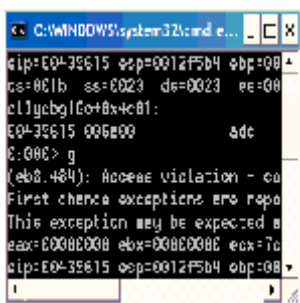
About the Author

Toby Opferman

Toby Opferman has worked in just about all aspects of Windows development including applications, services and drivers.

He has also played a variety of roles professionally on a wide range of projects. This has included pure researching roles, architect roles and developer roles. He also was also solely responsible for debugging traps and blue screens for a number of years.

Previously of Citrix Systems he is very experienced in the area of Terminal Services. He currently works on Operating Systems and low level architecture at Intel.




Member

Occupation: Engineer

Company: Intel

Location:  United States

Discussions and Feedback

 **17 messages** have been posted for this article. Visit <http://www.codeproject.com/KB/debug/cdbntsd5.aspx> to post and view comments on this article, or click [here](#) to get a print view with messages.

[PermaLink](#) | [Privacy](#) | [Terms of Use](#)

Last Updated: 9 May 2004

Editor: [Smitha Vijayan](#)

Copyright 2004 by Toby Opferman
Everything else Copyright © [CodeProject](#), 1999-2009
Web20 | [Advertise on the Code Project](#)