C++, ASMWin2K, WinXP, Win2003, Visual Studio, Dev, QA

Debug Tutorial Part 1: Beginning Debugging Using CDB and NTSD

By Toby Opferman

Posted: **20 Mar 2004**
Views: **198,168**
Bookmarked: **256 times**

Learn how to debug problems in software.

62 votes for this article.
Popularity: 8.69 Rating: **4.85** out of 5
1 2 3 4 5

Introduction

Debugging is one of the most valuable skill sets when it comes to software development and maintenance. This is a skill that is used at every stage of a product's life cycle. The developer first creating the project will obviously run into bugs. These bugs can be anywhere from logic bugs to syntax bugs and compiler errors. The quality assurance being conducted on the software may run into problems as more advanced scenarios are tested and the software interacts with other environments. Finally, after release of the product, it must be supported. The debugging does not end when the customer gets the software, bugs are generally escalated back to the company who will now again need to debug.

What is the goal of this tutorial?

This tutorial is merely an introduction to debugging. This would be considered "tutorial #1" and I will write more add-ons if the feedback is good. There are a lot of complex debugging techniques and issues that it's hard to know where to start. This tutorial attempts to start at the beginning and get you acquainted with debugging. I hope to expose novices and intermediate level programmers to the world of advanced debugging. "Advanced" debugging, basically without recompiling, without doing "message box or `printf` debugging".

Debuggers and Operating Systems

To download the latest debuggers from Microsoft, visit here.

CDB, NTSD and Windbg

This article will generally talk about Windows 2000 and higher Operating Systems. The three debuggers that we will talk about here are CDB, NTSD and WinDbg. Windows 2000 and higher systems generally have NTSD already installed on the system! This is a big bonus as you do not need to install any extra software for quick debugging.

So what's the difference? The documentation says "NTSD does not need a console window and CDB does". That is true. NTSD does not need a console window in order to run, while CDB does. However, I have found that there are a lot more differences. The first is that older NTSDs do not support PDB symbol files, they only support DBG! I also found that NTSD does not support the symbol server, while CDB does. Older NTSDs could not create a memory dump and I've also found other problems such as NTSD only supports up to 2 breakpoint commands. There is one advantage that NTSD has now that CDB does not. The ability to not have a console window.

The ability to not have a console window is vital when you are debugging a user-mode service or process before anyone has logged onto the system. If no one has logged onto the system, you cannot create a console window. There is a command line option, -d, which specifies for NTSD to communicate with the attached kernel debugger (CDB has the same option). This can be used on processes during startup to debug them through the kernel debugger. While you can debug a process using the kernel debugger already, this gives you the flexibility to debug the process using the user-mode debugger. This is outside the scope of this current introduction article, just digest

the concept for now.

WinDbg and CDB are basically the same with some few exceptions. The first is that WinDbg is a GUI and CDB is a console application. WinDbg also supports kernel debugging and source level debugging.

Visual C++ Debugger

I do not use this debugger and I would not recommend using it. The reasons are that this debugger is firstly a resource hog. It's slow loading and contains more than just debugging tools which makes it cumbersome. The second reason is generally, you need to reboot after you install this debugger. I generally work off the principal that the machine running or testing the software may not already have a debugger installed. VC++ is also a large, time consuming installation.

Windows 9x/ME

What can we do on Windows 9x/ME? Well, you can actually use WinDbg. The debug APIs are the same for all systems, so it has been long known to me that WinDbg should just "work" on Windows 9x/ME. My only concerns were if WinDbg attempted to detect it was on Windows 9x and not allow debugging. I recently found this to be untrue. The only problem is that the latest WinDbg installs are MSI packages that do not natively install on Windows 9x. This can be solved simply by installing them on an NT based machine and sharing the directory or even putting it on a CD. This obviously has other side effects though, such as do not think you can use all the !xxx commands as NT and 9x place their data in different memory locations. Do symbols work? Yes, PDBs work. I did find stepping through code after setting a *ba r1 xxxxx* was very slow though. This article does not cover Windows 9x/ME.

Setting Up Your Environment

This is a very important step before you start debugging or successfully set up your debug environment. The system needs to be configured to your liking and contain all the tools you need.

Symbols and the Symbol Server

Symbols are an important part of any debug operation. Microsoft contains a location where you can download all the symbols for any particular Operating System (Windows XP, etc.). The problem is, you need to have a lot of hard disk space and if you debug many Operating Systems on one machine (from crash dumps, etc.), then this is cumbersome.

To accommodate this need to debug many Operating Systems, Microsoft supports a "symbol server". This will help you to get the correct symbols onto your system. The symbol server is located here. If you set your symbol path to this location, your debugger will automatically download the system symbols that you need. The symbols that you need for your application are up to you.

Image File Execution Options

There is a location in the registry that will automatically attach a debugger to an application when it starts to run. This registry location is the following:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows
     NT\CurrentVersion\Image File Execution Options
```

Under this registry key, you simply create a new registry key with the name of the process you want to debug, such as "myapplication.exe". If you have not used this before, there is probably a default key already created called "Your Application Here" or something similar. You can rename that key and use it if you like.

One of the values on this key is "Debugger". This should point to the debugger you want to start when this application is run. The default for "Your Application Here" is "ntsd -d". You cannot use this unless you have a kernel debugger attached so I would remove the "-d" part.

Note: Keeping "-d" and not having a kernel debugger attached could result in locking up of your system every time that application is run! Be careful. If you have a kernel debugger setup, you can

unlock the system by hitting "g".

There is another value that may be there called "GlobalFlags". This is another tool that can be used for debugging, however it is outside the scope of this article. For more information on that, look up "gflags.exe".

Kernel Debugging Equipment

In order to kernel debug, you first need to boot the Operating System in debug mode. Although there is a GUI under *system properties* to do this, I generally edit the *boot.ini* directly. Locate the *boot.ini* on the root of your *C:* drive. It is most likely a hidden system file. I would attrib *-r -s -h boot.ini* and then open it for edit.

Caution: Editing this file incorrectly can prevent you from ever booting again!

The boot file may look like this:

```
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS.0=
    "Microsoft Windows XP Professional" /fastdetect
```

I would duplicate the first line under "Operating Systems":

```
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS.0=
    "Microsoft Windows XP Professional" /fastdetect
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS.0=
   "Microsoft Windows XP Professional"
   /fastdetect /debug /debugport=COM1 /baudrate=115200
```

The duplicated line can then contain your setup. */debug*, then */debugport=port* and finally */baudrate=baudrate*. The debug port to use is the port of that machine where you would hook up your SERIAL NULL MODEM CABLE. This is a piece of hardware that you need. You will also need another machine. Aside from using the COM ports, you can use firewire which is a lot faster.

Next time you boot, just select the "Debugger Enabled" selection in order to boot in debug mode.

Environment Variables

I would generally setup `_NT_SYMBOL_PATH` to point to the Microsoft Symbol server and the local directory that contains your symbol information. To set this environment path, go to System Properties -> Advanced -> Environment Variables.

Default Debugger

This is the default debugger that will be used when any crash happens on the system. By default, it's generally set to "Doctor Watson". That program is not worth mentioning here. The registry key is this location:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug
```

I would set "Auto" to 1 and "Debugger" to your debugger.

Assembly

I highly recommend that you learn assembly programming. These tutorials will not show source level debugging as I never do it and I don't even know how! The problems with source level debugging is that the source is not always available as well as sometimes the problem is not seen

when just looking at the source, but rather in the generated code. It also makes walking the system much easier. If you understand how the environment was setup, you can easily reverse the system to finding out the information you need to know and it may not always be available using Source Level debugging.

The other thing I hate about source level debugging is that if the source does not match the symbols, the source debugger will not show you the correct information. This means that if you create multiple builds of your program or change your program after you've built, you better be able to find the source that matches the build you're debugging!

Let's Get Started

This tutorial is basically Part One and if it's liked, I will write more, each getting more and more advanced. This first tutorial will walk through a couple of simple scenarios of user-mode programming problems.

Symbols For Release Executables

First, how do you create symbols for "release" binaries? That's simple. You create a *make* file that properly rebases the binaries.

The options I generally would use for *cl.exe* would be:

```
/nologo /MD /W3 /Oxs /Zi /I "..\..\inc" /D "WIN32" /D "_WINDOWS"
/Fr$(OBJDIR)\\ /Fo$(OBJDIR)\\ /Fd$(OBJDIR)\\ /c
```

The options I generally would use for *link.exe* would be:

```
/nologo /subsystem:console
  /out:$(TARGETDIR)\$(TARGET)/pdb:<YourProjectName>.pdb
  /debug /debugtype:both
/LIBPATH:"..\..\..\bin\lib"
```

This will create the *.PDB* for your project. Of course, with the introduction of VC++ 7, they have gotten rid of *.DBG*s (so */debugtype:both* may error on this compiler). *.DBG* is a smaller version of the *.PDB* and it does not contain source information, strictly symbol look ups. It does not even contain the parameters or anything. If you're using a compiler that can still generate them, here's what you do:

```
rebase -b 0x00100000 -x $(TARGETDIR) -a $(TARGETDIR)\$(TARGET)
```

The *-b* is the new memory location to rebase the executable to. However, this will strip the debug symbols from the release executable making it smaller in size. If you build an executable the default Visual Studio method, it may be a tiny bit smaller than this executable. However, you do not have symbols. The generated code is the same and just as optimized using the optimization flags you specify. The difference is that these binaries are now more useful, as no matter where they go or who uses them where, you can still get symbols!

Remember, the best debugging always occurs if you do not have to rebuild the executable. Once you have to rebuild the executable, you must also know that you've now changed the memory foot print of the executable. You may also have changed the speed of the executable. This is critical since you now have to reproduce the problem using this binary! What if it took 4 days to cause this problem? It would be best to be able to debug it as much as possible on the spot.

Simple Access Violation Trap

Let's walk through a simple problem. Your program crashes with "Access Violation", this is not uncommon! This is probably the most frequent problem that occurs when running an executable. There are three steps to help solving this problem.

1. Who is attempting to do the access? What module?
2. *What is it attempting to access? Where did the memory come from?*
3. Why is it attempting to access it? What does it want to do with it?

These are general guidelines to solving this problem. I put #2 in italics as it is probably the most important of the three. However, solving 1 and 3 can also help determine #2 if it is not immediately apparent.

I have created a very simple program that crashes. I have setup my default debugger to be CDB and I have now just run the program. I have also created symbols for this executable as well as set the `_NT_SYMBOL_PATH` to the Microsoft symbol server.

As we can see, this is what happens when we run the program:

```
C:\programs\DirectX\Games\src\Games\temp\bin>temp

Microsoft (R) Windows Debugger  Version 6.3.0005.1
Copyright (c) Microsoft Corporation. All rights reserved.

*** wait with pending attach
Symbol search path is:
  SRV*c:\symbols*http://msdl.microsoft.com/download/symbols

Executable search path is:
ModLoad: 00400000 00404000   C:\programs\DirectX\Games\src\Games\temp\bin\temp.e
xe
ModLoad: 77f50000 77ff7000   C:\WINDOWS.0\System32\ntdll.dll
ModLoad: 77e60000 77f46000   C:\WINDOWS.0\system32\kernel32.dll
ModLoad: 77c10000 77c63000   C:\WINDOWS.0\system32\MSVCRT.dll
ModLoad: 77dd0000 77e5d000   C:\WINDOWS.0\system32\ADVAPI32.DLL
ModLoad: 78000000 78086000   C:\WINDOWS.0\system32\RPCRT4.dll
(ee8.c38): Access violation - code c0000005 (!!! second chance !!!)
eax=00000000 ebx=7ffdf000 ecx=00001000 edx=00320608 esi=77c5aca0 edi=77f944a8
eip=77c3f10b esp=0012fb0c ebp=0012fd60 iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
MSVCRT!_output+0x18:
77c3f10b 8a18             mov     bl,[eax]                ds:0023:00000000=??
0:000>
```

What is the first thing we notice? This trap occurred in *MSVCRT.DLL*. This is apparent because the debugger generally displays this information using *<module>!<nearest symbol>+offset*. This means the closest symbol in *MSVCRT.DLL* is `_output` and we are +18h bytes into it. Given that this is such a small offset and providing that the symbols are correct (even symbols can be incorrect, but that's a later tutorial), we can assume that we are in `_output()` function of MSVCRT.

```
(ee8.c38): Access violation - code c0000005 (!!! second chance !!!)
eax=00000000 ebx=7ffdf000 ecx=00001000 edx=00320608 esi=77c5aca0 edi=77f944a8
eip=77c3f10b esp=0012fb0c ebp=0012fd60 iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
MSVCRT!_output+0x18:
77c3f10b 8a18             mov     bl,[eax]                ds:0023:00000000=??
0:000>
```

If we wanted to prove this, what could we do?

```
<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)
```

This command will give us a list of all the modules in the process with their beginning and ending memory locations. Our trap is at 77c3f10b, which is 77c10000 <= 77c3f10b <= 77c63000, so we are definitely trapped in MSVCRT. The next thing to do is find out where this memory came from.

There are a few methods of doing this, we could un-assemble the code and attempt to find out where the memory came from. We could also get a stack trace and figure out who's on the stack.

Let's first attempt to disassemble the `_output` function to see where the memory came from.

```
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
77c3f107 8b450c          mov     eax,[ebp+0xc]
0:000> u
MSVCRT!_output+0x17:
77c3f10a 53              push    ebx
77c3f10b 8a18            mov     bl,[eax]
```

I have highlighted all the important instructions to look at. Even if you do not know assembly, you will want to hear this out for what it is. First, we notice that the memory is coming from `EAX`. It's a register in the CPU, but we can just consider it a variable. The `[]`s around `EAX` is the same as doing `*MyPointer` in C. This means we are referencing the memory pointed to by `EAX`. Where did `EAX` come from? `EAX` came from [`EBP + 0Ch`], which you could think of as "`DWORD *EBP EAX = EBP[3];`". This is because in assembly, there are no types. `EAX` is a 32 bit (`DWORD`) register. Dereferencing the `DWORD` at `EBP` + 12 is the same in C as adding 3 to a `DWORD` pointer (or 12 to a byte pointer then typecasting to a `DWORD`).

The next thing to look at is `MOV EBP, ESP`. `ESP` is the STACK POINTER. As you should know, parameters (pending calling convention and optimizations) are pushed on the stack, return addresses are pushed on the stack and local variables are on the stack. `ESP` points to the stack! In memory, a function call would look like this for the C calling convention:

```
[Parameter n]
...
[Parameter 2]
[Parameter 1]
[Return Address]
```

Now, we see `PUSH EBP`. `PUSH` means put something on the stack. So, we are saving `EBP`'s previous value on the stack. So, our stack looks like this now:

```
[Parameter n]
...
[Parameter 2]
[Parameter 1]
[Return Address]
[Previous EBP]
```

Now that we have set `EBP` to `ESP`, we can treat it just like a pointer and the stack is just an array of `DWORD` values! So, here's the offsets of `EBP` and where they point:

```
[Parameter n]      ==   [EBP + n*4 + 4] (The formula)
...
[Parameter 2]      ==   [EBP + 12]
[Parameter 1]      ==   [EBP + 8]
[Return Address]   ==   [EBP + 4]
[Previous EBP]     ==   [EBP + 0]
```

This being the case, we know that our variable came from the second parameter of `_output`. So, now what? Let's un-assemble the calling function! We know that `EBP` + 4 points to the return address, or we could try to just get a stack trace.

```
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" is one of the commands to do this. Now, we may not always get a full stack trace, however, this too is for a more advanced tutorial. In this simple tutorial, we will assume we got the full stack trace. We notice, this is a `printf` function call or it looks that way. As we notice, `printf` called `_output`. Let's un-assemble `printf`. Please note that we may not always want to disassemble the entire function and we may use discession. Sometimes, we can find out the trap just from doing a stack trace (I will go over this in this simple context at the end). These are small functions though and we may be able to trace them simply.

```
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
77c3f107 8b450c           mov     eax,[ebp+0xc]
0:000> u
MSVCRT!_output+0x17:
77c3f10a 53               push    ebx
77c3f10b 8a18             mov     bl,[eax]
77c3f10d 33c9             xor     ecx,ecx
77c3f10f 84db             test    bl,bl
77c3f111 0f8445070000     je      MSVCRT!_output+0x769 (77c3
77c3f117 56               push    esi
77c3f118 57               push    edi
77c3f119 8bf8             mov     edi,eax
0:000> u MSVCRT!printf
MSVCRT!printf:
77c3e658 6a10             push    0x10
77c3e65a 68e046c177       push    0x77c146e0
77c3e65f e8606effff       call    MSVCRT!_SEH_prolog (77c354
77c3e664 bea0acc577       mov     esi,0x77c5aca0
77c3e669 56               push    esi
77c3e66a 6a01             push    0x1
77c3e66c e8bdadffff       call    MSVCRT!_lock_file2 (77c394
77c3e671 59               pop     ecx
0:000> u
MSVCRT!printf+0x1a:
77c3e672 59               pop     ecx
77c3e673 8365fc00         and     dword ptr [ebp-0x4],0x0
77c3e677 56               push    esi
77c3e678 e8c7140000       call    MSVCRT!_stbuf (77c3fb44)
77c3e67d 8945e4           mov     [ebp-0x1c],eax
77c3e680 8d450c           lea     eax,[ebp+0xc]
77c3e683 50               push    eax
77c3e684 ff7508           push    dword ptr [ebp+0x8]
0:000> u
MSVCRT!printf+0x2f:
77c3e687 56               push    esi
77c3e688 e8660a0000       call    MSVCRT!_output (77c3f0f3)
```

This is simple. We notice that the second parameter to `_output` is [EBP + 8]. We now notice that `PUSH EBP` and `MOV EBP, ESP` are there and thus the stack is setup the same way I mentioned previously. This is ***not always*** the case, but we are starting out slowly here.

Thus, we can determine that the first parameter to `printf()` is where the memory came from. And, as luck would have it, `printf()` was called from our program! From the trap information, we know that `EAX` was 0, so we were trying to dereference a `NULL` pointer.

```
77c3f10b 8a18     mov bl,[eax]    ds:0023:00000000=??
```

This was the code used:

```
int main(int argc, char *argv[])
{
 char *TheLastParameter[100];

 sprintf(*TheLastParameter, "The last parameter is %s", argv[argc]);
 printf(*TheLastParameter);

 return 0;
}
```

You can notice a lot of problems with it! However, the `printf` is what trapped since it was `NULL`. `*TheLastParameter` is `NULL`. Surprisingly it didn't trap on `sprintf()`. So, how would we have solved this just with *KB*? Look at this trace:

```
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>
```

We had symbols and we had the stack trace. The italics is the first parameter. It's 0. We also know that we called it. This is a very simple scenario though and I tried to portray some of the techniques that could be used to back trace to the location of a problem. Learn the stack. Knowing how the stack is setup and what memory is on the stack can be vital to finding and tracing where data came from. You will not always be that lucky to find where all information can be found with just doing "kb".

Program Not Working As Expected

This is a popular error. You run the program and you don't see the correct output or the program keeps giving you an error message. The file you want to create is not being created, etc. This is a very common problem that can be easy to complex to solve. What are some of the first steps you would take to debug this?

1. What is not working?
2. What APIs or modules would this revolve around?
3. What would cause those APIs to not function properly?

These are some steps, though they are not general. Let's say you have a program that attempts to create a file in Windows. The file is not created though. Let's look at some code:

```
HANDLE hFile;
DWORD dwWritten;

hFile = CreateFile("c:\MyFile.txt", GENERIC_READ,
                     0, NULL, OPEN_EXISTING, 0, NULL);

if(hFile != INVALID_HANDLE_VALUE)
{
 WriteFile(hFile, "Test", strlen("Test"), &dwWritten, NULL);
 CloseHandle(hFile);
}
```

This is your code. Generally, you would want to recompile with perhaps `GetLastError()` and print it out. However, you do not have to do that. Although in this case it may be simple to, if you're stepping through code and a function fails, wouldn't you want to know what happened on the spot? Let's try to debug this. First, we'll start the debugger and break on our function. Since we have symbols, this is easy. If we didn't, we could just break on `CreateFile` as it is an exported symbol

and would always be available.

```
C:\programs\DirectX\Games\src\Games\temp\bin>cdb temp

Microsoft (R) Windows Debugger  Version 6.3.0005.1
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: temp
Symbol search path is:
    SRV*c:\symbols*http://msdl.microsoft.com/download/symbols

Executable search path is:
ModLoad: 00400000 00404000   temp.exe
ModLoad: 77f50000 77ff7000   ntdll.dll
ModLoad: 77e60000 77f46000   C:\WINDOWS.0\system32\kernel32.dll
ModLoad: 77c10000 77c63000   C:\WINDOWS.0\system32\MSVCRT.dll
(2a0.94): Break instruction exception - code 80000003 (first chance)
eax=00241eb4 ebx=7ffdf000 ecx=00000004 edx=77f51310 esi=00241eb4 edi=00241f48
eip=77f75a58 esp=0012fb38 ebp=0012fc2c iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
77f75a58 cc               int     3
0:000> bp temp!main
0:000> g
```

We set a break point on our `main()` function and hit "go". We get the break point and we use "p" to step instruction by instruction to our `CreateFile` function.

```
Breakpoint 0 hit
eax=77c5c9e4 ebx=7ffdf000 ecx=00322cf8 edx=00322cf8 esi=00000000 edi=00000000
eip=00401000 esp=0012ff50 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:
00401000 51               push    ecx
0:000> p
eax=77c5c9e4 ebx=7ffdf000 ecx=00322cf8 edx=00322cf8 esi=00000000 edi=00000000
eip=00401001 esp=0012ff4c 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+0x1:
00401001 56               push    esi
0:000>
eax=77c5c9e4 ebx=7ffdf000 ecx=00322cf8 edx=00322cf8 esi=00000000 edi=00000000
eip=00401002 esp=0012ff48 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+0x2:
00401002 57               push    edi
0:000>
eax=77c5c9e4 ebx=7ffdf000 ecx=00322cf8 edx=00322cf8 esi=00000000 edi=00000000
eip=00401003 esp=0012ff44 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+0x3:
00401003 33ff             xor     edi,edi
0:000>
eax=77c5c9e4 ebx=7ffdf000 ecx=00322cf8 edx=00322cf8 esi=00000000 edi=00000000
eip=00401005 esp=0012ff44 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+0x5:
00401005 57               push    edi
0:000>
eax=77c5c9e4 ebx=7ffdf000 ecx=00322cf8 edx=00322cf8 esi=00000000 edi=00000000
eip=00401006 esp=0012ff40 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+0x6:
00401006 57               push    edi
0:000>
eax=77c5c9e4 ebx=7ffdf000 ecx=00322cf8 edx=00322cf8 esi=00000000 edi=00000000
eip=00401007 esp=0012ff3c 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+0x7:
00401007 6a03             push    0x3
0:000>
eax=77c5c9e4 ebx=7ffdf000 ecx=00322cf8 edx=00322cf8 esi=00000000 edi=00000000
eip=00401009 esp=0012ff38 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+0x9:
00401009 57               push    edi
0:000>
eax=77c5c9e4 ebx=7ffdf000 ecx=00322cf8 edx=00322cf8 esi=00000000 edi=00000000
eip=0040100a esp=0012ff34 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+0xa:
0040100a 57               push    edi
0:000>
eax=77c5c9e4 ebx=7ffdf000 ecx=00322cf8 edx=00322cf8 esi=00000000 edi=00000000
eip=0040100b esp=0012ff30 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+0xb:
0040100b 6800000080       push    0x80000000
0:000>
eax=77c5c9e4 ebx=7ffdf000 ecx=00322cf8 edx=00322cf8 esi=00000000 edi=00000000
eip=00401010 esp=0012ff2c 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+0x10:
00401010 6810304000       push    0x403010
0:000>
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
```

After we call CreateFile, EAX will have the return value. We notice it's ffffffff or "Invalid Handle Value". We want to know the GetLastError. This is stored at fs:34. FS is the TEB selector, so we can dump it.

```
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 also has a quicker way to do it, !gle:

```
0:000> !gle
LastErrorValue: (Win32) 0x2 (2) - The system cannot find the file specified.
LastStatusValue: (NTSTATUS) 0xc0000034 - Object Name not found.
0:000>
```

So, the file cannot be found. But, the file is there! So what's the problem? We need to debug this further. One thing we could look at is what parameter was passed into CreateFile.

```
eax=77c5c9e4 ebx=7ffdf000 ecx=00322cf8 edx=00322cf8 esi=00000000 edi=00000000
eip=00401010 esp=0012ff2c 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+0x10:
00401010 6810304000       push    0x403010
0:000>
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)]{kernel32!CreateFileA (77e7b476)} ds:0023:00402004=77e7b476
```

Luckily, it's a constant so the memory will still be around. It would still be around even if it wasn't though since we didn't step too far away from the return of CreateFile.

We can then use "da", "dc" or "du". "da" is dump ANSI string, "du" is dump Unicode string and "dc" is similar to "dd" except it dumps all characters, even unprintable ones. Since we know it's an ANSI string, just use *da*.

```
0:000> da 403010
00403010  "c:MyFile.txt"
0:000>
```

That's wrong! We need to use *C:\\MyFile.txt* to get it to work with the *C:*!

So, we fix this. But wait, it still won't write! We need to debug this further.

We do the same thing again.

```
C:\programs\DirectX\Games\src\Games\temp\bin>cdb temp

Microsoft (R) Windows Debugger  Version 6.3.0005.1
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: temp
Symbol search path is:
  SRV*c:\symbols*http://msdl.microsoft.com/download/symbols

Executable search path is:
ModLoad: 00400000 00404000   temp.exe
ModLoad: 77f50000 77ff7000   ntdll.dll
ModLoad: 77e60000 77f46000   C:\WINDOWS.0\system32\kernel32.dll
ModLoad: 77c10000 77c63000   C:\WINDOWS.0\system32\MSVCRT.dll
(80c.c94): Break instruction exception - code 80000003 (first chance)
eax=00241eb4 ebx=7ffdf000 ecx=00000004 edx=77f51310 esi=00241eb4 edi=00241f48
eip=77f75a58 esp=0012fb38 ebp=0012fc2c iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
77f75a58 cc               int     3
0:000> bp temp!main
0:000> g
Breakpoint 0 hit
eax=77c5c9e4 ebx=7ffdf000 ecx=00322cf8 edx=00322cf8 esi=00000000 edi=00000000
eip=00401000 esp=0012ff50 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:
00401000 51               push    ecx
0:000> p
eax=77c5c9e4 ebx=7ffdf000 ecx=00322cf8 edx=00322cf8 esi=00000000 edi=00000000
eip=00401001 esp=0012ff4c 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+0x1:
00401001 56               push    esi
0:000>
eax=77c5c9e4 ebx=7ffdf000 ecx=00322cf8 edx=00322cf8 esi=00000000 edi=00000000
eip=00401002 esp=0012ff48 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+0x2:
00401002 57               push    edi
0:000>
eax=77c5c9e4 ebx=7ffdf000 ecx=00322cf8 edx=00322cf8 esi=00000000 edi=00000000
eip=00401003 esp=0012ff44 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+0x3:
00401003 33ff             xor     edi,edi
0:000>
eax=77c5c9e4 ebx=7ffdf000 ecx=00322cf8 edx=00322cf8 esi=00000000 edi=00000000
eip=00401005 esp=0012ff44 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+0x5:
00401005 57               push    edi
0:000>
eax=77c5c9e4 ebx=7ffdf000 ecx=00322cf8 edx=00322cf8 esi=00000000 edi=00000000
eip=00401006 esp=0012ff40 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+0x6:
00401006 57               push    edi
0:000>
eax=77c5c9e4 ebx=7ffdf000 ecx=00322cf8 edx=00322cf8 esi=00000000 edi=00000000
eip=00401007 esp=0012ff3c 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+0x7:
00401007 6a03             push    0x3
0:000>
eax=77c5c9e4 ebx=7ffdf000 ecx=00322cf8 edx=00322cf8 esi=00000000 edi=00000000
eip=00401009 esp=0012ff38 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+0x9:
00401009 57               push    edi
0:000>
eax=77c5c9e4 ebx=7ffdf000 ecx=00322cf8 edx=00322cf8 esi=00000000 edi=00000000
eip=0040100a esp=0012ff34 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+0xa:
0040100a 57               push    edi
```

We get here and we notice that EAX is a valid handle and not invalid handle value! Let's continue.

```
eax=000007e8 ebx=7ffdf000 ecx=77f59037 edx=00140608 esi=000007e8 edi=00000000
eip=0040101d 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+0x1d:
0040101d 83feff           cmp     esi,0xffffffff
0:000>
eax=000007e8 ebx=7ffdf000 ecx=77f59037 edx=00140608 esi=000007e8 edi=00000000
eip=00401020 esp=0012ff44 ebp=0012ffc0 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000213
temp!main+0x20:
00401020 741b             jz      temp!main+0x3d (0040103d)
0:000>
eax=000007e8 ebx=7ffdf000 ecx=77f59037 edx=00140608 esi=000007e8 edi=00000000
eip=00401022 esp=0012ff44 ebp=0012ffc0 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000213
temp!main+0x22:
00401022 8d442408         lea     eax,[esp+0x8]     ss:0023:0012ff4c=00322cf8
0:000>
eax=0012ff4c ebx=7ffdf000 ecx=77f59037 edx=00140608 esi=000007e8 edi=00000000
eip=00401026 esp=0012ff44 ebp=0012ffc0 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000213
temp!main+0x26:
00401026 57               push    edi
0:000>
eax=0012ff4c ebx=7ffdf000 ecx=77f59037 edx=00140608 esi=000007e8 edi=00000000
eip=00401027 esp=0012ff40 ebp=0012ffc0 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000213
temp!main+0x27:
00401027 50               push    eax
0:000>
eax=0012ff4c ebx=7ffdf000 ecx=77f59037 edx=00140608 esi=000007e8 edi=00000000
eip=00401028 esp=0012ff3c ebp=0012ffc0 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000213
temp!main+0x28:
00401028 6a04             push    0x4
0:000>
eax=0012ff4c ebx=7ffdf000 ecx=77f59037 edx=00140608 esi=000007e8 edi=00000000
eip=0040102a esp=0012ff38 ebp=0012ffc0 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000213
temp!main+0x2a:
0040102a 6820304000       push    0x403020
0:000>
eax=0012ff4c ebx=7ffdf000 ecx=77f59037 edx=00140608 esi=000007e8 edi=00000000
eip=0040102f esp=0012ff34 ebp=0012ffc0 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000213
temp!main+0x2f:
0040102f 56               push    esi
0:000>
eax=0012ff4c ebx=7ffdf000 ecx=77f59037 edx=00140608 esi=000007e8 edi=00000000
eip=00401030 esp=0012ff30 ebp=0012ffc0 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000213
temp!main+0x30:
00401030 ff1500204000 call dword ptr [temp!_imp__WriteFile (00402000)]{kernel32!
WriteFile (77e7f13a)} ds:0023:00402000=77e7f13a
0:000> p
eax=00000000 ebx=7ffdf000 ecx=77e7f1c9 edx=00000015 esi=000007e8 edi=00000000
eip=00401036 esp=0012ff44 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+0x36:
00401036 56               push    esi
```

We've just called WriteFile and EAX == 0. That means false! Let's check the other variables.

The second parameter is right and its length is 4:

```
0:000> da 403020
00403020  "Test"
```

The fourth parameter in bold back there, it's a pointer to the number of bytes written. It's 0.

```
0:000> dd 012ff4c
0012ff4c  00000000 00401139 00000001 00322470
0012ff5c  00322cf8 00403000 00403004 0012ffa4
0012ff6c  0012ff94 0012ffa0 00000000 0012ff98
0012ff7c  00403008 0040300c 00000000 00000000
0012ff8c  7ffdf000 00000001 00322470 00000000
0012ff9c  8053476f 00322cf8 00000001 0012ff84
0012ffac  e1176590 0012ffe0 00401200 004020c0
0012ffbc  00000000 0012fff0 77e814c7 00000000
```

Well, let's check GetLastError.

```
0:000> !gle
LastErrorValue: (Win32) 0x5 (5) - Access is denied.
LastStatusValue: (NTSTATUS) 0xc0000022 - {Access Denied}
              A process has requested access to an object,
              but has not been granted those access rights.
0:000>
```

Access denied? What could cause that! Let's check, wait, we opened the file for READ access only! We didn't open the file for write access! So, we can easily fix this problem and move onto our next project!

```
hFile = CreateFile("c:\\MyFile.txt", GENERIC_READ,
        0, NULL, OPEN_EXISTING, 0, NULL);
```

Conclusion

In summary, this is just an introduction to some very basic debugging techniques. The examples were simple but you must take their value for the techniques they displayed. This is just the first installment of this debugging tutorial. Hopefully, if there is interest, I may add more tutorials getting more advanced.

To some, this tutorial may be simple, to others too advanced. You will not become a good debugger overnight, it takes practice. I would suggest attempting to use the debugger even on the simplest of problems, to solve them. The more you practice, the better you get. I guarantee, the more you fool around with the tools, the more you will learn.

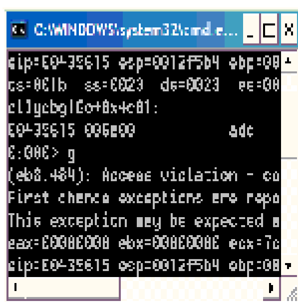
License

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



Member

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.

Occupation: Engineer

Company: Intel

Location: United States

Discussions and Feedback

33 messages have been posted for this article. Visit **http://www.codeproject.com/KB/debug/cdbntsd.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: 20 Mar 2004
Editor: Smitha Vijayan

Copyright 2004 by Toby Opferman
Everything else Copyright © CodeProject, 1999-2009
Web17 | Advertise on the Code Project