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

Debug Tutorial Part 4: Writing WINDBG Extensions

By Toby Opferman

Posted: **25 Mar 2004**
Views: **115,839**
Bookmarked: **113 times**

This tutorial we will attempt to write a simple debug extension.

30 votes for this article.
Popularity: 7.03 Rating: **4.76** out of 5
1 2 3 4 5

Download source files - 3.94 Kb
Download demo project - 1.55 Kb

Introduction

Welcome to the fourth installment of this debugging series. In this installment, we will be getting a bit away from actual debugging for a bit and dive into creating helpful debugging aids. I definitely would never condone writing debugging aids just to write them. I would find a reason to write something, perhaps something you do all the time that gets tedious. Once you have found something you would love to automate, I would then see how you could automate it.

This is what I have done. During my debugging escapades, I **always** search the stack and other locations for strings. Why do I do this? People are not computers, we understand language rather than numbers. This being the case, a lot of applications and even drivers are written based upon strings. I would not say everything is a string, but there's usually a string somewhere. If you think about it, you really don't need strings at all. We could all just use numbers and never use another string again. Some may think well, when you expose a UI of course, you're going to eventually run into a string somewhere... Well, doesn't have to be that way, now does it? I mean, I'm not even talking about just User Interfaces, I'm talking about the guts of programs that aren't even exposed to the user at all. Programmers are still human and like to talk in some language beyond binary. This being the case, even the internals of applications sometimes use strings to represent things. You can find strings just about anywhere, even in drivers.

So There're Strings, So What?

Well, they provide an added level of readability to an application. This is the first rule of thumb, perhaps if you could find a string somewhere on the stack, you could better track down what the application is doing. These strings could be environment strings, file names, device names (COM1, *Device**xxxx*, etc.), names of other objects, user names, GUIDs, etc. This information can better help track down what the application is doing and where it could be in your program.

Another interesting concept is, I don't know this for a fact, but it's my belief that the most common buffer overruns are due to invalid string manipulations. Whether it be forgetting the `NULL` character, or misjudging allocation since an API returns # of characters and not # of bytes. If a string overwrote memory in your program and you can find the string, it's a lot easier to track down who created it.

Where do we start?

I start on the stack. I have a trap, the first thing I do after "KB" and "DDS ESP" would then be to use "DC ESP". This command dumps the `DWORD`s on one side and the printable characters on the other. Let's see an example of this:

```
0:000> dc esp
0006febc  77d43a09 77d43c7d 0006fefc 00000000  .:.w}<.w........
0006fecc  00000000 00000000 00000000 0006ff1c  ................
0006fedc  010028e4 0006fefc 00000000 00000000  .(..............
0006feec  00000000 00000000 77e7ad86 00091ee7  ...........w....
0006fefc  001a03e4 00000118 0000ffff bf8a75ed  .............u..
0006ff0c  0768a2ca 00000229 00000251 00000000  ..h.)...Q.......
0006ff1c  0006ffc0 01006c54 01000000 00000000  ....Tl..........
0006ff2c  00091ee7 0000000a 00000000 00000000  ................
0:000> dc
0006ff3c  7ffdf000 80543940 f544fc5c 00000044  ....@9T.\.D.D...
0006ff4c  00092b28 00092b48 00092b70 00000000  (+..H+..p+......
0006ff5c  00000000 00000000 00000000 00000000  ................
0006ff6c  00000000 00000000 00000000 00000000  ................
0006ff7c  00000000 ffffffff ffffffff ffffffff  ................
0006ff8c  00091ee7 00000000 00000001 00272620  ............ &'.
0006ff9c  00272d00 00000000 00000000 0006ff34  .-'.........4...
0006ffac  e24296d0 0006ffe0 01006d14 01001888  ..B......m......
0:000>
0006ffbc  00000000 0006fff0 77e814c7 00000000  ...........w....
0006ffcc  00000000 7ffdf000 f544fcf0 0006ffc8  ..........D.....
0006ffdc  80534504 ffffffff 77e94809 77e91210  .ES......H.w...w
0006ffec  00000000 00000000 00000000 01006ae0  .............j..
0006fffc  00000000 78746341 00000020 00000001  ....Actx .......
0007000c  00000654 0000007c 00000000 00000020  T...|....... ...
0007001c  00000000 00000014 00000001 00000003  ................
0007002c  00000034 000000ac 00000001 00000000  4...............
0:000>
0007003c  00000000 00000000 00000000 00000000  ................
0007004c  00000002 00000000 00000000 00000000  ................
0007005c  00000168 00000190 00000000 2d59495b  h...........[IY-
0007006c  000002f8 00000032 0000032c 000002b8  ....2...,.......
0007007c  00000010 00000002 0000008c 00000002  ................
0007008c  00000001 000000ac 00000538 00000001  ........8.......
0007009c  00000002 000005e4 00000070 00000001  ........p.......
000700ac  64487353 0000002c 00000001 00000001  SsHd,...........
0:000>
000700bc  00000003 00000002 0000008c 00000001  ................
000700cc  00000000 0000002c 0000005e 0000005e  ....,...^...^...
000700dc  00000000 00000000 00000000 00000000  ................
000700ec  00000000 00000000 00000000 00000000  ................
000700fc  00000000 00000002 00000028 00000034  ........(...4...
0007010c  003a0043 0057005c 004e0049 004f0044  C.:.\.W.I.N.D.O.
0007011c  00530057 0030002e 0057005c 006e0069  W.S...0.\.W.i.n.
0007012c  00780053 005c0073 00000000 00000000  S.x.s.\.........
```

I started *notepad.exe* and just broke into it, then dumped the stack of the primary (and only) thread. The strings on the stack are "local arrays", such as declaring `char x[10];` in your function. These aren't the only strings in a program though. There are others and these others are stored in pointers that are declared as local variables and even passed to functions such as `CreateFile`. The first parameter of `CreateFile` takes a string.

So, what do I usually do? I then search the stack for memory locations which could be strings and I then do "DC" again on them or "DA" for Dump ANSI string or "DU" for Dump Unicode String. The problem with this is that it's slow and tedious. I could not find any debugger command to do this for me (if there is one, let me know), so I ended up writing my own.

Writing Your Own?

WINDBG supports DLLs created by anyone as long as they export functions and behave in a manner defined by Microsoft. This means you can write PLUGINS! It used to be that people would write plug-ins to !<mydatastructure> <address> which basically dumped the members of their data structure along with the names. However, WINDBG supports the "dt" command that if you have a PDB (symbol file) it can do this for you without writing any code! Let's see an example of this.

Using the "dt <yourdll>!<your data structure>" will dump the structure's content along with their names. Let's look at a quick example.

```
0:000> dt ntdll!_PEB
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged    : UChar
   +0x003 SpareBool        : UChar
   +0x004 Mutant           : Ptr32 Void
   +0x008 ImageBaseAddress : Ptr32 Void
   +0x00c Ldr              : Ptr32 _PEB_LDR_DATA
   +0x010 ProcessParameters : Ptr32 _RTL_USER_PROCESS_PARAMETERS
   +0x014 SubSystemData    : Ptr32 Void
   +0x018 ProcessHeap      : Ptr32 Void
   +0x01c FastPebLock      : Ptr32 _RTL_CRITICAL_SECTION
   +0x020 FastPebLockRoutine : Ptr32 Void
   +0x024 FastPebUnlockRoutine : Ptr32 Void
   +0x028 EnvironmentUpdateCount : Uint4B
   +0x02c KernelCallbackTable : Ptr32 Void
   +0x030 SystemReserved   : [1] Uint4B
   +0x034 ExecuteOptions   : Pos 0, 2 Bits
   +0x034 SpareBits        : Pos 2, 30 Bits
   +0x038 FreeList         : Ptr32 _PEB_FREE_BLOCK
   +0x03c TlsExpansionCounter : Uint4B
   +0x040 TlsBitmap        : Ptr32 Void
   +0x044 TlsBitmapBits    : [2] Uint4B
   +0x04c ReadOnlySharedMemoryBase : Ptr32 Void
   +0x050 ReadOnlySharedMemoryHeap : Ptr32 Void
   +0x054 ReadOnlyStaticServerData : Ptr32 Ptr32 Void
   +0x058 AnsiCodePageData : Ptr32 Void
   +0x05c OemCodePageData  : Ptr32 Void
   +0x060 UnicodeCaseTableData : Ptr32 Void
   +0x064 NumberOfProcessors : Uint4B
   +0x068 NtGlobalFlag     : Uint4B
   +0x070 CriticalSectionTimeout : _LARGE_INTEGER
   +0x078 HeapSegmentReserve : Uint4B
   +0x07c HeapSegmentCommit : Uint4B
   +0x080 HeapDeCommitTotalFreeThreshold : Uint4B
   +0x084 HeapDeCommitFreeBlockThreshold : Uint4B
   +0x088 NumberOfHeaps    : Uint4B
   +0x08c MaximumNumberOfHeaps : Uint4B
   +0x090 ProcessHeaps     : Ptr32 Ptr32 Void
   +0x094 GdiSharedHandleTable : Ptr32 Void
   +0x098 ProcessStarterHelper : Ptr32 Void
   +0x09c GdiDCAttributeList : Uint4B
   +0x0a0 LoaderLock       : Ptr32 Void
   +0x0a4 OSMajorVersion   : Uint4B
   +0x0a8 OSMinorVersion   : Uint4B
   +0x0ac OSBuildNumber    : Uint2B
   +0x0ae OSCSDVersion     : Uint2B
   +0x0b0 OSPlatformId     : Uint4B
   +0x0b4 ImageSubsystem   : Uint4B
   +0x0b8 ImageSubsystemMajorVersion : Uint4B
   +0x0bc ImageSubsystemMinorVersion : Uint4B
   +0x0c0 ImageProcessAffinityMask : Uint4B
   +0x0c4 GdiHandleBuffer  : [34] Uint4B
   +0x14c PostProcessInitRoutine : Ptr32
   +0x150 TlsExpansionBitmap : Ptr32 Void
   +0x154 TlsExpansionBitmapBits : [32] Uint4B
   +0x1d4 SessionId        : Uint4B
   +0x1d8 AppCompatFlags   : _ULARGE_INTEGER
   +0x1e0 AppCompatFlagsUser : _ULARGE_INTEGER
   +0x1e8 pShimData        : Ptr32 Void
   +0x1ec AppCompatInfo    : Ptr32 Void
   +0x1f0 CSDVersion       : _UNICODE_STRING
   +0x1f8 ActivationContextData : Ptr32 Void
   +0x1fc ProcessAssemblyStorageMap : Ptr32 Void
   +0x200 SystemDefaultActivationContextData : Ptr32 Void
   +0x204 SystemAssemblyStorageMap : Ptr32 Void
   +0x208 MinimumStackCommit : Uint4B
0:000>
```

We just dumped out the structure information for the `_PEB` structure defined by Windows. We will now attempt to find our PEB and dump this address.

```
0:000> !teb
TEB at 7ffde000
    ExceptionList:        0006ffb0
    StackBase:            00070000
    StackLimit:           0005f000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7ffde000
    EnvironmentPointer:   00000000
    ClientId:             00000b80 . 00000f40
    RpcHandle:            00000000
    Tls Storage:          00000000
    PEB Address:          7ffdf000
    LastErrorValue:       0
    LastStatusValue:      c0000034
    Count Owned Locks:    0
    HardErrorMode:        0
0:000> dt ntdll!_PEB 7ffdf000
   +0x000 InheritedAddressSpace : 0 ''
   +0x001 ReadImageFileExecOptions : 0 ''
   +0x002 BeingDebugged    : 0x1 ''
   +0x003 SpareBool        : 0 ''
   +0x004 Mutant           : 0xffffffff
   +0x008 ImageBaseAddress : 0x01000000
   +0x00c Ldr              : 0x00191ea0
   +0x010 ProcessParameters : 0x00020000
   +0x014 SubSystemData    : (null)
   +0x018 ProcessHeap      : 0x00090000
   +0x01c FastPebLock      : 0x77fc49e0
   +0x020 FastPebLockRoutine : 0x77f5b2a0
   +0x024 FastPebUnlockRoutine : 0x77f5b380
   +0x028 EnvironmentUpdateCount : 1
   +0x02c KernelCallbackTable : 0x77d42a38
   +0x030 SystemReserved   : [1] 0
   +0x034 ExecuteOptions   : 0y00
   +0x034 SpareBits        : 0y000000000000000000000000000000 (0)
   +0x038 FreeList         : (null)
   +0x03c TlsExpansionCounter : 0
   +0x040 TlsBitmap        : 0x77fc4680
   +0x044 TlsBitmapBits    : [2] 0x7ff
   +0x04c ReadOnlySharedMemoryBase : 0x7f6f0000
   +0x050 ReadOnlySharedMemoryHeap : 0x7f6f0000
   +0x054 ReadOnlyStaticServerData : 0x7f6f0688  -> (null)
   +0x058 AnsiCodePageData : 0x7ffb0000
   +0x05c OemCodePageData  : 0x7ffc1000
   +0x060 UnicodeCaseTableData : 0x7ffd2000
   +0x064 NumberOfProcessors : 1
   +0x068 NtGlobalFlag     : 0x70
   +0x070 CriticalSectionTimeout : _LARGE_INTEGER 0xffffe86d`079b8000
   +0x078 HeapSegmentReserve : 0x100000
   +0x07c HeapSegmentCommit : 0x2000
   +0x080 HeapDeCommitTotalFreeThreshold : 0x10000
   +0x084 HeapDeCommitFreeBlockThreshold : 0x1000
   +0x088 NumberOfHeaps    : 5
   +0x08c MaximumNumberOfHeaps : 0x10
   +0x090 ProcessHeaps     : 0x77fc5a80  -> 0x00090000
   +0x094 GdiSharedHandleTable : 0x00360000
   +0x098 ProcessStarterHelper : (null)
   +0x09c GdiDCAttributeList : 0x14
   +0x0a0 LoaderLock       : 0x77fc1774
   +0x0a4 OSMajorVersion   : 5
   +0x0a8 OSMinorVersion   : 1
   +0x0ac OSBuildNumber    : 0xa28
   +0x0ae OSCSDVersion     : 0x100
   +0x0b0 OSPlatformId     : 2
   +0x0b4 ImageSubsystem   : 2
   +0x0b8 ImageSubsystemMajorVersion : 4
   +0x0bc ImageSubsystemMinorVersion : 0
   +0x0c0 ImageProcessAffinityMask : 0
   +0x0c4 GdiHandleBuffer  : [34] 0
   +0x14c PostProcessInitRoutine : (null)
   +0x150 TlsExpansionBitmap : 0x77fc4660
   +0x154 TlsExpansionBitmapBits : [32] 0
   +0x1d4 SessionId        : 0
   +0x1d8 AppCompatFlags   : _ULARGE_INTEGER 0x0
```

If you remember correctly, the !teb gives you the "thread environment block". A part of this block will show you the PEB for a process. Now, you see that not only does it print the information contained in the data structure, it also knows **how** to display the information based upon the type of data. Why did I show you this? This is because of what I specified above. The first thing people do or want to do is start to write debug extensions to dump all their data structures and this is not feasible anymore! You then have to keep changing your debug code every time you add or move information around. It's best to use the "dt" command and painlessly find all the data contained in any of your internal data structures.

Starting To Write The Extension

So, my proposed extension is !dumpstrings. This will go through a memory location and dump all the pointers. I can then do !dumpstrings esp and dump the strings to all pointers on the stack.

First, let's start with how you write the extension. WINDBG requires you at least export two functions. These functions in my source are posted below:

```
/***********************************************************
 * ExtensionApiVersion
 *
 * Purpose: WINDBG will call this function to get the version
 *          of the API
 *
 *  Parameters:
 *     Void
 *
 *  Return Values:
 *     Pointer to a EXT_API_VERSION structure.
 *
 ***********************************************************/
LPEXT_API_VERSION WDBGAPI ExtensionApiVersion (void)
{
    return &g_ExtApiVersion;
}


/***********************************************************
 * WinDbgExtensionDllInit
 *
 * Purpose: WINDBG will call this function to initialize
 *          the API
 *
 *  Parameters:
 *     Pointer to the API functions, Major Version, Minor Version
 *
 *  Return Values:
 *     Nothing
 *
 ***********************************************************/
VOID WDBGAPI WinDbgExtensionDllInit (PWINDBG_EXTENSION_APIS
           lpExtensionApis, USHORT usMajorVersion,
           USHORT usMinorVersion)
{
     ExtensionApis = *lpExtensionApis;
}
```

The first function, `ExtensionApiVersion`, simply returns a version, and all we do is supply version numbers that WINDBG would expect to make it happy. Here is the declaration of `g_ExtApiVersion`.

```
/***********************************************************
 * Global Variable Needed For Versioning
 ***********************************************************/
EXT_API_VERSION g_ExtApiVersion = {
         5 ,
         5 ,
         EXT_API_VERSION_NUMBER ,
         0
     } ;
```

The `EXT_API_VERSION_NUMBER` is declared in *wdbgexts.h*. Please note that there are other variations of debugger extension DLLs, such as using *ntsdexts.h*. I am going over just one simple example that works with the current CDB and WinDbg debuggers that you can download off of Microsoft's web site. I am also using *windbgexts.h*, not *ntsdexts.h*. If you look at your SDK include files, you will notice you have both headers.

So, how is `EXT_API_VESRION_NUMBER` declared? On my system, it is declared as:

```
#define EXT_API_VERSION_NUMBER   5

typedef struct EXT_API_VERSION {
    USHORT  MajorVersion;
    USHORT  MinorVersion;
    USHORT  Revision;
    USHORT  Reserved;
} EXT_API_VERSION, *LPEXT_API_VERSION;
```

Where did you come up with 5, 5? I just debugged some of the other extensions that come with WINDBG to find those numbers. I also found that older WINDBGs use 3, 5. This is really a moot point though, we just need a basic frame work written, after which we can simply write all the commands we want! I'm not big on making things like this big issues or digging deep into their meaning when, it's really irrelevant to me as long as my extension loads.

The `WinDbgExtensionDllInit` API simply gives your application a virtual function table. This table ***must*** be named a certain name. Well, it doesn't **have** to be but it's easier when it is. The reason is that the *windbgexts.h* contains macros to make function calls to functions stored in this structure. If you don't use the same name, you'll have to create the calls yourself. This is my global in the code:

```
/***********************************************************
 * Global Variable Needed For Functions
 ***********************************************************/
WINDBG_EXTENSION_APIS ExtensionApis = {0};
```

It's really no big deal, the macros just make it a little easier, that's all. You can do whatever you like though. This is the dump of *WINDBGEXTS.H* macros:

```
#define dprintf          (ExtensionApis.lpOutputRoutine)
#define GetExpression    (ExtensionApis.lpGetExpressionRoutine)
#define GetSymbol        (ExtensionApis.lpGetSymbolRoutine)
#define Disassm          (ExtensionApis.lpDisasmRoutine)
#define CheckControlC    (ExtensionApis.lpCheckControlCRoutine)
#define ReadMemory       (ExtensionApis.lpReadProcessMemoryRoutine)
#define WriteMemory      (ExtensionApis.lpWriteProcessMemoryRoutine)
#define GetContext       (ExtensionApis.lpGetThreadContextRoutine)
#define SetContext       (ExtensionApis.lpSetThreadContextRoutine)
#define Ioctl            (ExtensionApis.lpIoctlRoutine)
#define StackTrace       (ExtensionApis.lpStackTraceRoutine)
```

What's next? Aside from these two APIs, you can have a `CheckVersion()` function to force your extension to only use commands on certain versions of WINDBG. I found this completely useless and didn't implement it as it's not required. So, let's just write a function!

The first function will be simple. We'll do "!help" which will dump the help.

```
/*********************************************************
 * !help
 *
 * Purpose: WINDBG will call this API when the user types !help
 *
 *
 *  Parameters:
 *     N/A
 *
 *  Return Values:
 *     N/A
 *
 *********************************************************/
DECLARE_API (help)
{
    dprintf("Toby's Debug Extensions\n\n");
    dprintf("!dumpstrings <ADDRESS register> - Dumps 20 strings in"\
       "ANSI/UNICODE using this address as a pointer to strings (useful for" \
       "dumping strings on the stack) \n");
       /* String Split so it is readable in this article. */
}
```

`dprintf();` is basically, "debug `printf`" and it's exactly like `printf()`! It will print output to the debugger. The `DECLARE_API(<command>)` is a simple way to declare the API name. Remember, the name you give the function is the name you use in the debugger. I called this help, so to use it, it's !help or !<dllname>.help. This is a simple function that will simply display a help message to the user.

The next thing we want to do is implement our string function. This function will take a parameter which will be a memory address. I also want it to work like current commands, if you do *dc <address>* then *dc* again, it will continue where it left off dumping the memory. So, let's see what I've come up with.

```
/***********************************************************
 * !dumpstrings
 *
 * Purpose: WINDBG will call this API when the user types !dumpstrings
 *
 *
 *  Parameters:
 *     !dumpstrings or !dumpstrings <ADDRESS register>
 *
 *  Return Values:
 *     N/A
 *
 ***********************************************************/
DECLARE_API (dumpstrings)
{
    static ULONG Address = 0;
    ULONG  GetAddress, StringAddress, Index = 0, Bytes;
    WCHAR MyString[51] = {0};


    GetAddress = GetExpression(args);

    if(GetAddress != 0)
    {
        Address = GetAddress;
    }

    dprintf("STACK   ADDR   STRING \n");

    for(Index = 0; Index < 4*20; Index+=4)
    {
        memset(MyString, 0, sizeof(MyString));

        Bytes = 0;

        ReadMemory(Address + Index, &StringAddress,
                           sizeof(StringAddress), &Bytes);

        if(Bytes)
        {
           Bytes = 0;

           ReadMemory(StringAddress, MyString,
                 sizeof(MyString) - 2, &Bytes);

           if(Bytes)
           {
              dprintf("%08x : %08x = (UNICODE) \"%ws\"\n",
                       Address + Index, StringAddress, MyString);
              dprintf("%08x : %08x = (ANSI)    \"%s\"\n",
                       Address + Index, StringAddress, MyString);
           }
           else
           {
              dprintf("%08x : %08x =  Address Not Valid\n",
                             Address + Index, StringAddress);
           }
        }
        else
        {
           dprintf("%08x : Address Not Valid\n", Address + Index);
        }
    }

    Address += Index;
}
```

So, the first function I use is `GetExpression()`. On the newer WINDBGs, it works like this. `ADDRESS GetExpression(SYMBOLIC STRING)`. You pass in a symbolic string, such as the arguments to the command and it attempts to resolve it to an address. The arguments to the command are stored in `args`, so I pass in `args`. This will resolve symbols, addresses or even registers to numbers as would be with the case of passing in `ESP`.

I now have a static variable. If `GetExpression()` returned 0, it's possible there are no arguments. If this is the case, I use `Address`, my static variable. This works if someone does !dumpstrings. It will continue where it left off. At the end of the function, I always save `Address` as being the next location to dump.

The next function I use is `dprintf()` which works just like using `printf`. I've explained this one already, so onto the next tidbit. Addresses are 4 bytes long, so I will increment this address by 4 each time around in the loop. I want to dump 20 addresses, so I loop from 0 to 4*20. Very simple indeed.

Now, you cannot simply reference the memory returned as you are not in the process space of the application. So, WINDBG provides functions to do this such as `ReadMemory()`. (The *windbgexts.h* provides prototypes for all the APIs, so you can experiment if you cannot find the API online.) The `ReadMemory()` function takes 4 arguments:

```
ReadMemory(Address In Process To Read,
           Local Variable to store the memory read,
           size of the local variable,
           pointer to a DWORD that returns the number
           of bytes read from the memory location);
```

So, we pass in our pointer to the memory in the application, a pointer to place the data on return, then receive the number of bytes. If no bytes are returned, we print an invalid memory address; if the bytes are returned, we then reference this memory location to read up to 49 bytes (we have 51 and put two `NULL`s at the end for Unicode support). If I was able to read anything, I then attempt to display it using the `dprintf()` function in ANSI then in UNICODE. If the memory returns 0 bytes, I print an error specifying it's not a valid address. This is all very simple.

The next thing we need to do is create our *.DEF* file. This file will simply export our functions.

```
LIBRARY "TDBG.DLL"

EXPORTS
    WinDbgExtensionDllInit
    ExtensionApiVersion
    dumpstrings
    help
```

Now, we need to build it. I like using *make* files and I use Visual Slickedit as my editor. I don't use the VC++ IDE at all. So, in the project, I've created a *make* file. This is how you would set it up. First, run *VCVARS32.BAT* which is located in the *BIN* directory of your VC++ installation. I moved mine to *C:* for easier use. The next thing to do is simply type "nmake" in the directory that the source code was extracted to.

```
C:\Programming\Programs\debugext\src\debug>\vcvars32
Setting environment for using Microsoft Visual C++ tools.
C:\Programming\Programs\debugext\src\debug>nmake

Microsoft (R) Program Maintenance Utility   Version 6.00.8168.0
Copyright (C) Microsoft Corp 1988-1998. All rights reserved.

        cl /nologo /MD /W3 /Oxs /Zi  /I ".\inc"  /D "WIN32" /DLL /D "_WINDOWS"
/Fr.\obj\i386\\ /Fo.\obj\i386\\ /Fd.\obj\i386\\ /c .\tdbg.c
tdbg.c
C:\PROGRA~1\MICROS~2\VC98\INCLUDE\wdbgexts.h(526) : warning C4101: 'li' : unrefe
renced local variable
        link.exe /DLL /nologo /def:tdbg.def /out:..\..\bin\tdbg.dll  /pdb:tdbg.p
db /debug /debugtype:both USER32.LIB  KERNEL32.LIB .\obj\i386\tdbg.obj
   Creating library ..\..\bin\tdbg.lib and object ..\..\bin\tdbg.exp
        rebase.exe -b 0x00100000 -x ..\..\bin -a ..\..\bin\tdbg.dll

REBASE: Total Size of mapping 0x00010000
REBASE: Range 0x00100000 -0x00110000

C:\Programming\Programs\debugext\src\debug>nmake clean

Microsoft (R) Program Maintenance Utility   Version 6.00.8168.0
Copyright (C) Microsoft Corp 1988-1998. All rights reserved.

Deleted file - C:\Programming\Programs\debugext\src\debug\obj\i386\tdbg.obj
Deleted file - C:\Programming\Programs\debugext\src\debug\obj\i386\tdbg.sbr
Deleted file - C:\Programming\Programs\debugext\src\debug\obj\i386\vc60.pdb

C:\Programming\Programs\debugext\src\debug>
```

If you want to build again, you'd need to either change the source so the date is updated or type "nmake clean" to rebuild. You should now have a binary.

```
 Volume in drive C has no label.
 Volume Serial Number is 2CF8-F7B5

 Directory of C:\Programming\Programs\debugext\bin

03/25/2004  08:56 PM    <DIR>          .
03/25/2004  08:56 PM    <DIR>          ..
03/25/2004  09:53 PM             2,412 tdbg.dbg
03/25/2004  09:53 PM            20,752 tdbg.dll
03/25/2004  09:53 PM             1,044 tdbg.exp
03/25/2004  09:53 PM             2,538 tdbg.lib
03/25/2004  09:53 PM            82,944 tdbg.pdb
               5 File(s)        109,690 bytes
               2 Dir(s)  12,229,009,408 bytes free
```

Simply copy this binary to a location that can be found by your WinDbg, such as your *windows* directory. You can specify the path to load these extensions using !load and !unload (to unload the extension if you built a new version/ want to force the debugger to unload it so you can reload the next one). You can use !<dll name>.<function name> or just !<function name>. The mentioning of the binary name will force WINDBG to look for it to load it. It can also help to distinguish between which DLL to use for a command if multiple DLL extensions export it.

Let's Try It Out

Now that we have created our debug extension, I move it to a location where WINDBG can load it (such as my *windows* directory). I then use "!tdbg.dumpstrings esp" to dump all the strings on the stack. This is the same application (restarted so addresses may change) restarted (I verified with *dc esp*, I get the same strings on the stack as before). I now want to dump all the pointers to strings on the stack. Let's try this and see what happens!

```
0:000> !tdbg.dumpstrings esp
STACK    ADDR    STRING
0006febc : 77d43a09 = (UNICODE) ""
0006febc : 77d43a09 = (ANSI)    "&#9516;&#9658;"
0006fec0 : 77d43c7d = (UNICODE) ""
0006fec0 : 77d43c7d = (ANSI)    "�N&#9830;�&#8729;&#9787;&#9786;"
0006fec4 : 0006fefc = (UNICODE) ""
0006fec4 : 0006fefc = (ANSI)    "&#9616;&#9829;&#8596;"
0006fec8 : 00000000 =  Address Not Valid
0006fecc : 00000000 =  Address Not Valid
0006fed0 : 00000000 =  Address Not Valid
0006fed4 : 00000000 =  Address Not Valid
0006fed8 : 0006ff1c = (UNICODE) ""
0006fed8 : 0006ff1c = (ANSI)    "&#9492; &#9824;"
0006fedc : 010028e4 = (UNICODE) ""
0006fedc : 010028e4 = (ANSI)    "�&#9492;u�&#934;&#9500;&#8745;    5�"
0006fee0 : 0006fefc = (UNICODE) ""
0006fee0 : 0006fefc = (ANSI)    "&#9616;&#9829;&#8596;"
0006fee4 : 00000000 =  Address Not Valid
0006fee8 : 00000000 =  Address Not Valid
0006feec : 00000000 =  Address Not Valid
0006fef0 : 00000000 =  Address Not Valid
0006fef4 : 77e7ad86 = (UNICODE) ""
0006fef4 : 77e7ad86 = (ANSI)    "�|$&#9830;"
0006fef8 : 00091ee8 = (UNICODE) ""
0006fef8 : 00091ee8 = (ANSI)    ""
0006fefc : 001d03de = (UNICODE) ""
0006fefc : 001d03de = (ANSI)    ""
0006ff00 : 00000118 =  Address Not Valid
0006ff04 : 0000ffff =  Address Not Valid
0006ff08 : bf8a75ed =  Address Not Valid
0:000> !tdbg.dumpstrings
STACK    ADDR    STRING
0006ff0c : 077d5cc8 =  Address Not Valid
0006ff10 : 000001f3 =  Address Not Valid
0006ff14 : 000001df =  Address Not Valid
0006ff18 : 00000000 =  Address Not Valid
0006ff1c : 0006ffc0 = (UNICODE) ""
0006ff1c : 0006ffc0 = (ANSI)    "&#8801; &#9824;"
0006ff20 : 01006c54 = (UNICODE) ""
0006ff20 : 01006c54 = (ANSI)    "�&#8801;�u�9]&#931;uV �&#9604;&#8597;"
0006ff24 : 01000000 = (UNICODE) ""
0006ff24 : 01000000 = (ANSI)    "MZ�"
0006ff28 : 00000000 =  Address Not Valid
0006ff2c : 00091ee8 = (UNICODE) ""
0006ff2c : 00091ee8 = (ANSI)    ""
0006ff30 : 0000000a =  Address Not Valid
0006ff34 : 00000000 =  Address Not Valid
0006ff38 : 00000000 =  Address Not Valid
0006ff3c : 7ffdf000 = (UNICODE) ""
0006ff3c : 7ffdf000 = (ANSI)    ""
0006ff40 : 80543940 =  Address Not Valid
0006ff44 : f4910c5c =  Address Not Valid
0006ff48 : 00000044 =  Address Not Valid
0006ff4c : 00092b30 = (UNICODE) ""
0006ff4c : 00092b30 = (ANSI)    ""
0006ff50 : 00092b50 = (UNICODE) ""
0006ff50 : 00092b50 = (ANSI)    "WinSta0\Default"
0006ff54 : 00092b78 = (UNICODE) ""
0006ff54 : 00092b78 = (ANSI)    "C:\WINDOWS.0\System32\notepad.exe"
0006ff58 : 00000000 =  Address Not Valid
0:000> !tdbg.dumpstrings
STACK    ADDR    STRING
0006ff5c : 00000000 =  Address Not Valid
0006ff60 : 00000000 =  Address Not Valid
0006ff64 : 00000000 =  Address Not Valid
0006ff68 : 00000000 =  Address Not Valid
0006ff6c : 00000000 =  Address Not Valid
0006ff70 : 00000000 =  Address Not Valid
0006ff74 : 00000000 =  Address Not Valid
0006ff78 : 00000000 =  Address Not Valid
0006ff7c : 00000000 =  Address Not Valid
0006ff80 : ffffffff =  Address Not Valid
0006ff84 : ffffffff =  Address Not Valid
0006ff88 : ffffffff =  Address Not Valid
0006ff8c : 00091ee8 = (UNICODE) ""
```

We found two strings on the stack. Not bad and all I did was start Notepad. I didn't do anything else. This is a simple example of how this API could be used. Take a memory location and dump out all the pointers to strings. You are always going to get garbage and invalid address dumped on the stack (or any memory location for that matter). It's the valid addresses that contain strings that we are looking for.

Conclusion

I hope you enjoyed learning how to write a debug extension and hopefully even find the example useful! There are other APIs listed which you can look up or perhaps another tutorial can explain them in the future. The basics of reading memory and displaying information were covered. Hopefully, in the future, we can explore more advanced debugging commands.

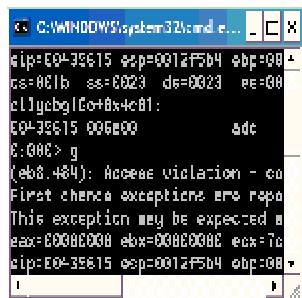
License

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

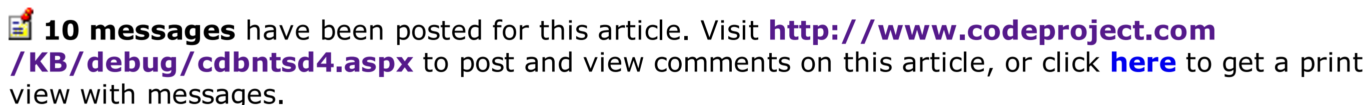
Toby Opferman has worked in just about all aspects of Windows development including applications, services and drivers.

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



10 messages have been posted for this article. Visit **http://www.codeproject.com/KB/debug/cdbntsd4.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: 25 Mar 2004
Editor: Smitha Vijayan

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