Introduction to Debugging with Windbg
Module Overview

Introduction to Debugging

- Callstacks and Symbols
- Windbg for .NET Debugging
- Son of Strike (SOS)
- Review
Callstacks and Symbols
Anatomy Of A Call Stack
(unmanaged)

<table>
<thead>
<tr>
<th>ChildEBP</th>
<th>RetAddr</th>
<th>Args to Child</th>
<th>Module &amp; Function name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0006cfac</td>
<td>77d51ca9</td>
<td>0006cfd0</td>
<td>KERNEL32!CreateFileW</td>
</tr>
<tr>
<td>0006d010</td>
<td>77d51ae4</td>
<td>00088770</td>
<td>RPCRT4 !NMP_Open+0x159</td>
</tr>
<tr>
<td>0006d1a0</td>
<td>77d456c7</td>
<td>0006d200</td>
<td>RPCRT4 !I_RpcGetBuffer+0xb</td>
</tr>
<tr>
<td>0006d1ac</td>
<td>77da1d9d</td>
<td>0006d24c</td>
<td>RPCRT4 !NdrGetBuffer+0x26</td>
</tr>
<tr>
<td>0006d3f8</td>
<td>77dbd1db</td>
<td>77dbbe9c</td>
<td>RPCRT4 !NdrClientCall2+0x3f0</td>
</tr>
<tr>
<td>0006d408</td>
<td>77dbd147</td>
<td>0006d408</td>
<td>ADVAPI32 !LsarOpenPolicy2+0x14</td>
</tr>
<tr>
<td>0006d45c</td>
<td>77dbeb64</td>
<td>0006d480</td>
<td>ADVAPI32 !LsaOpenPolicy+0xaf</td>
</tr>
<tr>
<td>0006d4ec</td>
<td>77a8aaef</td>
<td>0006d510</td>
<td>ADVAPI32 !LookupAccountSidW+0x1f</td>
</tr>
<tr>
<td>0006d628</td>
<td>77a8b08a</td>
<td>0006d684</td>
<td>ole32 !LookupPrincName+0x166</td>
</tr>
<tr>
<td>0006d664</td>
<td>77a8b779</td>
<td>00092c30</td>
<td>ole32 !CoInitializeSecurity+0x2db</td>
</tr>
<tr>
<td>0006d690</td>
<td>77b29544</td>
<td>00083430</td>
<td>ole32 !InitializeSecurity+0x27</td>
</tr>
<tr>
<td>0006d69c</td>
<td>77a5e3b1</td>
<td>77a52060</td>
<td>ole32 !ChannelProcessInitialize+0xeb</td>
</tr>
</tbody>
</table>
Calling Convention
(32Bit unmanaged)

The function’s return **address is at EBP+4**
mostly StdCall:

- function arguments start at (Child) EBP+8, EBP+0c etc.
  - When EBP is **not** present function arguments start at
    ESP+4, then increment by 4H (ESP+8, ESP+C, ESP+10 etc.)

- Local variables are negative offsets of EBP (EBP-4, EBP-8, etc.)

- Most functions will store their **return value** in EAX -> pseudo
  register: @$retreg
Calling Convention (X64 unmanaged)

FASTCALL Only
Return value gets stored within rax
First four integer arguments go into registers
- Integer: rcx, rdx, r8, r9
- Floating point: XMM0 – XMM3
Parameters smaller than 64 are not zero extended - the upper bits are garbage

Stack Pointer is rsp
The Stack is growing downwards (like x86)
Calling Convention (X64)

```c
int _tmain(int argc, _TCHAR* argv[])
{
    SomeFunction(1,2,3,4,5);

    00000001400010A6 mov dword ptr [rsp+20h], 5
    00000001400010AE mov r9d, 4
    00000001400010B4 mov r8d, 3
    00000001400010BA mov edx, 2
    00000001400010BF mov ecx, 1
    00000001400010C4 call SomeFunction (14000100Ah)
    return 0;

    ...
```
C++ fastcall - X64

//Debug Version for better callstacks:
int _stdcall SomeFunction (int a, int b, int c, int d, int e)
{
    mov    DWORD PTR [f], r9d  //Data is realigned on the stack
    mov    DWORD PTR [rsp+18h], r8d
    mov    DWORD PTR [rsp+10h], edx
    mov    DWORD PTR [rsp+8], ecx
    push   rdi
    sub    rsp, 30h
    mov    rdi, rsp
    mov    ecx, 0Ch

Reference:
The history of calling conventions, part 5: amd64
And for Debugging: Challenges of Debugging Optimized x64 Code
C++ fastcall - X64

#pragma optimize("g",off) //for better callstacks!
int _stdcall SomeFunction (int a, int b, int c, int d, int e)
{
    00007FF7B95832B0 mov    dword ptr [f],r9d //Data is realigned on the stack
    00007FF7B95832B5 mov    dword ptr [rsp+18h],r8d
    00007FF7B95832BA mov    dword ptr [rsp+10h],edx
    00007FF7B95832BE mov    dword ptr [rsp+8],ecx
    00007FF7B95832C2 push   rdi
    00007FF7B95832C3 sub    rsp,30h
    00007FF7B95832C7 mov    rdi,rsp
    00007FF7B95832CA mov    ecx,0Ch
Symbols (unmanaged)

You *need* good symbols

- Always ensure that the version of source you are using matches the module image
  - Is the call stack consistent with the source?
  - Are blank lines being executed?
  - Optimisations may invalidate source
  - Use Microsoft public symbol server:

```
set _NT_SYMBOL_PATH=srv*DownstreamStore*http://msdl.microsoft.com/download/symbols
```

Source Level Debugging

- Only works if you have “Private” symbols
  - Private symbols do not guarantee source level debugging
    - It may lack line number information
    - Make sure Debug / Source Mode is checked
Understanding Mismatched Symbols

Mismatched symbols occur when you use symbols for a binary that were not built at the same time as that binary.

The debugger will normally fail to load mismatched symbols or will warn you that they are wrong:

Use `!sym noisy` to understand why your symbols are not loading.

```
ERROR: Symbol file could not be found. Defaulted to export symbols for ntdll.dll

The debugger will normally fail to load mismatched symbols or will warn you that they are wrong.
Use `!sym noisy` to understand why your symbols are not loading.
```

```
01bffd70 77f7f49f SharedUserData!SystemCallStub+0x4
WARNING: Stack unwind information not available. Following frames may be wrong.
01bfffe10 77d46db9 ntdll!ZwWaitForMultipleObjects+0xc
01bfffe6c 77d46e5b USER32!UserLpkPSMTextOut+0x15c
01bfffe88 75f8a5f3 USER32!MsgWaitForMultipleObjects+0x1d
```
Managed Symbols?

**Managed .NET Debugging:**

- possible without
- Better with :-)
  - Complete callstack
  - More infos

**Instruct CLR not to optimize the code (during jit) without recompiling the dll:**

- Use an ini file (and symbols)
  - MyDll.ini:
    
    ```ini
    [\$.NET Framework Debugging Control]
    AllowOptimize=0
    ```

**Instruct CLR to ignore (optimized) Ngen Image**

- Use Environment variable: `set COMPLUS_ZapDisable=1`
Where to store my symbols?

Companies are storing their symbols in symbol servers.

Maintain all versions which you might need to troubleshoot at some point.

Microsoft public symbol server: http://msdl.microsoft.com/download/symbols

Microsoft private symbols server: http://symweb

Customers should set up and maintain their own store for this.
Symbol Server Structure
How to set up a symbol server

Set up a file share
Give access to the user account doing build operation
Use Symstore.exe to store symbols on the share
To store public symbols use the Binplace.exe which generates stripped symbols from the private ones
Windbg Debugging
WinDBG outlook
Execution Control

**Break**
- Ctrl+Break, or Menu: Debug -> Break

**Go**
- g or F5 : continue execution
- gn : go not handled
- gh : go handled

**Step**
- p or F10 : Step Over
- t or F11 : Step Into

**Detach**
- q : Quit the debug session – it will terminate your application
- qd : Quit the debug session with detach – the process won’t terminate
Thread Symbols and Commands

~. : Current Thread
~# : Display thread caused exception
~5 : Display Thread 5 can be 0...Thread Count-1
~5s : Set Thread 5 to be the current one
~*: All Threads

Example: Display the call stack for all threads ~*kb
Navigating on a Thread

**Change frames**
.frame <frame number>

**View Local Variables**
d : display names of locals
dv –v : display used registers
dt : displays information about variables

0:000> dv
hInstanceId = 01000000
hPrevInstanceId = 00000000
lpAnsiCmdLine = 00091eeb ""
cmdShow = 0xa
msg = tagMSG
lpfnRegisterPenApp = 00000000
Call Stacks

**CDB/Windbg**

*k* command with several options

Display of calls and arguments

*kb*: include first three parameters

*kn*: include frame numbers

*kv*: include frame type info, including FPO info

*kd*: display raw stack data

*kf*: display the distance between frames

*kp*: gives detailed symbol information about parameters

*kL*: display without source lines
Call Stacks – examples

K Command

Return address
poi(ESP)

OutputListBox has a variable number of arguments: (..)
Call Stacks – examples

what are the arguments?

KB Command – displays the first 3 Arguments -> DWORDs on the stack
Call Stacks – examples

KN adds the frame number to the display

options can be chained together to get the desired results.
View Registers and Assembly

View Registers R command – CDB/Windbg

Disassembly

U <address>
Uf <function>

> u eip-10
MSVCRTD!_output+0x59c:
1021673C  898A8FDFFFFFFF  mov  dword ptr [i],eax
10216742  85D2  test edx,edx
10216744  741E  je $L17842+0FEh
10216746  88DB0FDFFFFFFF  mov  ecx,dword ptr [p]
1021674C  0FBE11  movsx edx,byte ptr [ecx]
1021674F  85D2  test edx,edx
10216751  7411  je $L17842+0FEh
10216753  8B85B0FDFFFFFFF  mov  eax,dword ptr [p]
Examine/Modify Memory

d commands – display address or address range

dp : display pointer (64Bit on 64Bit target)
dd : display double words (DWORDS)
dc : display double words and ASCII values
da : display ASCII value
du : display Unicode value
ed : edit memory (ex. ed 0x23478924 10)

? : expression evaluator

0:000> ?10
Evaluated expression: 16 = 00000010
CDB and Windbg

**X** command, used to examine modules, types,…

**X *! :** list all the modules

**X Kernel32!Op* :** list all symbols starting with “Op” in Kernel32.dll

```
0:001> x kernel32!Op*
76d60964 kernel32!OpenProfileUserMapping
76cd1225 kernel32!OpenThread
..```

**LM :** List loaded modules with source path

**LN <address> :** find closest symbol to the given address

**LMVM <module name> :** List verbose module information
Breakpoints

**Static (Fixed) Breakpoints**

bp : set breakpoint

ba : break on access

**Conditional Breakpoints**

Break at location if condition is true

\[ \text{bp MyFunction+0xb "j (poi(MyVar) > 0x20)"} \]

Break if location changes value

\[ \text{ba w &MyVar} \quad \text{- breaks on a write to the address of MyVar} \]

Break at location if location == value

\[ \text{bp MyMod!myFunction "j MyMod!g_myGlobal == 1"} \]

Break at location after count is reached

\[ \text{bp MyFunction+0xb 7} \]
Breakpoints (continued)

**Execute debug statements when breakpoint is reached**

```
bp MyFunction+0xb "kb;.frame 2;dv;g"
```

**Others**

- **bu**: Breakpoint unresolved
  - can be set on modules which are not loaded right now
  - gets resolved when module loads
  - stores Breakpoint in the Workspace

- **bl**: list breakpoints
- **bd**: disable breakpoint
- **be**: enable breakpoint
- **bc**: clear breakpoint
- **bd ***: disables all breakpoints
- **bd 3**: disables Breakpoint 3
Set Exceptions

Controls debugger actions when an exception occurs

- sxe : enable
- sxd : disable
- sxi : ignore
- sxn : notify

Types of events to handle

- sxe ld : break when a module loads
- sxn av : notify (don’t break) on Access Violations
Debugger Log File Commands

CDB/Windbg

Logfile Open/Close/Append

.logopen [filename] // open a new log file
.logappend [filename] // appends to an existing log file
.logclose // close current log file

Add comments

* [comment] // Used to add a comment
.echo [comment] // Will echo back what you type

Screen

.cls : Clears the screen
WinDbg and .NET

No .NET support
Need sos to work with managed code.

Sos 1.0 / 1.1

- Ships with the Framework SDK but a better and newer version is included with WinDbg
  - .load clr10\sos.dll

.NET 2.0 ships his “own” sos.dll

- .loadby sos mscorwks
to load out of the framework directory use
- psscor2
- Needs mscordacwks.dll of the framework you are debugging (.NET 2.0, .NET 2.0 Sp2,...)
- For troubleshooting use
  - .cordll –ve –u –l if
Windbg

.NET 4.0, 4.5 ships his “own” sos.dll

- .loadby sos clr
- psscor4 for .NET 4.5 not available
- Needs mscordacwks.dll of the framework you are debugging
- For troubleshooting use
  - .cordll –ve –u –l if

Silverlight comes with its „own“ sos

- .load C:\Program Files (x86)\Microsoft Silverlight\4.0.60310.0\sos.dll
Intro to .NET Debugging
“Extensions” for .NET

!SosEx
  !dlk  - DeadLocks
  !mbm  - Sets a Breakpoint
  !mbe/mbc/mbd – Sets/Clear Breakpoints
  !mx   - !x for managed
  !mdt  - dt for managed
  !mlocks - displays locks
  and more…

CLRMD
  API for automating
  Dump Analysis and writing debugging extensions
• data mining .NET extension
  !wdo - Display ad-hoc objects or arrays from GAC or Stack
  !wfrom - Perform SQL-like analysis of Heap objects enabling comparison, expression evaluation and indexed filtering.
  !whash - Display HashTable objects
  !wservice - List WCF service Objects,....

  !wfrom -type *.BasicHttpBinding select $addr(), $typename(), name;

  !wfrom -type *.SqlCommand where ( $contains(_commandText, "SELECT") && (!$contains(_commandText,"SELECT TOP")) ) select _commandText;

http://netext.codeplex.com/
Dumps from different System

To Debug a Dump from a different System:

• Use Microsoft public symbol server

or

• Copy the mscordacwks.dll where your dump comes from onto your Vista Machine
• add the directory of the file to the symbol path in WinDBG.

For troubleshooting use

• .cordll –ve –u –l if
• If sos does not give correct results use the one from the “dump system”
Remote Debugging with Windbg

Three Forms:

- With a Debugging Server - 2 debuggers
- With a Process Server
- With a Shared Command Line
Remote Debugging with Debugging Server (2*Windbg)

Two debuggers one acting as server and the other as client

**Advantages:**
- Easy to set up:
  - .server tcp:port=1234
  - windbg -remote tcp:server=MyComp,port=1234
- Fast and efficient

**Disadvantages:**
- Symbols need to be in the target
- Not suitable if the target cannot handle the load
Remote Debugging with Process Server

**DbgSrv Process Server** runs on the target system
**Debugger** runs on the client system

**Advantages:**
- Advanced capabilities on the host used by the target
- No symbols required on the target
- Minimal load on the target

**Disadvantages:**
- Sensitive to network issues, outages, and so on
- Less efficient than using debugging server
- Cannot remote debug a dump file
Remote Debugging with a Shared Command Line

Debugger and Target Process on the same machine with the debugger’s command line is shared

REMOTE.EXE (named-pipes)

Advantages:
- Better tolerance of network issues, except the shared command line
- Setup fairly easy with CDB
- CDB with WSREMOTE.EXE can use Internet for debug

Disadvantages:
- Only console debugging
Debugger Command Programs

Convenient for immediate and smaller tasks.

Uses debugger commands combined with simple control flow tokens:

- `if`, `else`, `elsif`
- `for`, `foreach`
- `do`, `while`
- `break`, `continue`
- `catch`, `leave`
- `printf`
- `block`

!for_each_module, !for_each_frame, and !for_each_local

-c (command-line option to run a command on startup)
Windbg helpers

.prefer_dml 1

- Enables Debugger Markup Language per default

.cmdtree init.txt

- Released in Debugging Tools for Windows (6.6.7.5) and largely undocumented
- Save a file with commonly used commands and load at runtime
- windbg.exe -c ".cmdtree init.txt" to load during startup

```
windbg ANSI Command Tree 1.0
  title {"Common Commands"}
  body
  {"Common Commands"}
    {"Information"}
    {"Time of dump"} {".time"}
    {"Process being debugged"} {"|"}
    {"Dump Location"} {"||"}
    {"Create server on port 9999"} {".server tcp:port=9999"}
    {"Process Environment Block"} {"!peb"}
  {"Logging"}
    {"Open Log"} {".logopen /t /u /d"}
    {"Close Log"} {".logclose"}
  {"Modules"}
    {"All Modules"} {"lm D sm"}
    {"Loaded Modules"} {"lmo D sm"}
    {"Loaded Modules (verbose)"} {"lmvo D sm"}
    {"Modules w/o symbols"} {"lme D sm"}
```
Reference

http://windbg.info
Lab

L01_Intro to dotNET Debugging