WinDbg. From A to Z!

By Robert Kuster

December 2007. All rights reserved.
www.windbg.info

Why WinDbg?

Because WinDbg is:

- used by the Microsoft Windows product team to develop Windows
- much more powerful than the well-known Visual Studio debugger
- extensible through extension DLLs
- its debug engine is part of the Windows OS

Up from Windows XP dbgeng.dll and dbghelp.dll are installed in "C:\Windows\System32".
Why “WinDbg. From A to Z”? 

- WinDbg's documentation is sub-optimal for people new to the topic  
- Without good documentation and examples the learning curve for WinDbg is very steep  
  In fact many people give up soon after the installation. 
- “WinDbg. From A to Z!” is a quick start and introduction to WinDbg. After reading it you will have a good feeling about what WinDbg is and what it can do for you. 

While many parts of “WinDbg. From A to Z!” are based on user-mode examples, you will benefit from it even if you are doing kernel-mode development. Note that the same debugging engine is running behind the scenes, no matter if you debug user-mode or kernel-mode code. Essentially the only visible difference for kernel-mode debugging is that you will end up using another set of extension commands.

Table of Contents - Roadmap

→ Behind the Scenes  
• Using WinDbg  
• Global Flags  
• Application Verifier  
• Process Dumps
Debugging Tools for Windows XP

- WinDbg.exe
- ntds.exe
- cdb.exe
- kd.exe
- dbgsrv.exe
- userdump.exe
- drwtsn32.exe
- livekd.exe
- OlyDbg.exe
- ProcessExplorer.exe
- ...

\[ \text{\{}} \]

\[ \text{\{}} \]

\[ \text{dbgeng.dll} \]

\[ \text{dbghelp.dll} \]

Debug Help Library: dbghelp.dll

- Documented in MSDN
- Included in the operating system, starting with Windows 2000
- Contains support routines for:
  a) Process Dumping (MiniDumpWriteDump, DbgHelpCreateUSerDump, ..)
  b) Obtaining Stack Traces (StackWalk64, ..)
  c) Symbol Handling (SymFromAddr, Sym* ..)
  d) Obtaining info about executable images (ImageNtHeader, FindDebugInfoFile, ..)

Many c) and d) functions are duplicates (same declaration) also found and exported from imagehlp.dll. While many imaghlp functions are simply forwarded to dbghelp functions, a disassembly of some functions reveals that they are obviously build from the same sources (see disassembly on next slide). While some MS Tools prefer the usage of DbgHelp.dll, some tools like Visual Studio or Dependency Walker rely on imagehlp.dll or use both libraries.
**Debugger Engine API: dbgeng.dll**

- Documented in WinDbg's documentation
  To get the header and lib files for dbgeng.dll: Chose "Custom Installation" and select "SDK" components in addition to the standard items.

- Included in the operating system, starting with Windows XP

- Accessible through interfaces:
  \textit{IDebugAdvanced, IDebugControl, IDebugSystemObjects, ...}

- Everything that can be performed by a debugger is exposed by an interface

Fact 1: WinDbg is really just a shell on top of a debugging engine.

Fact 2: You can write new standalone tools on top of this engine.

---

**DbgEng Dependencies**

![Dependency Walker](dependency_walker.png)
Debug Symbols

- Executables are just sequences of raw bytes
- Symbols help the debugger to:
  - map raw addresses in the executable to source-code lines
  - analyze internal layout and data of applications

Program Database → PDB Files

- The newest Microsoft debug information format
  COFF and CodeView are considered deprecated.
- PDB’s are stored in a file separately from the executable
- PDB format is not documented
- There are special APIs to work with it: DbgHelp.dll and MsDiaXY.dll

Kinds of Debug Information

<table>
<thead>
<tr>
<th>Kind of information</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public functions and variables</td>
<td>Functions and variables visible across several compilation units (source files)</td>
</tr>
<tr>
<td>FPO information</td>
<td>Additional information needed for retrieving stack-frames when compiling with FPO optimization (frame pointer omission)</td>
</tr>
<tr>
<td>Private functions and variables</td>
<td>All functions and variables including local variables, function parameters, ..</td>
</tr>
<tr>
<td>Source file and line information</td>
<td>Source file and line information</td>
</tr>
<tr>
<td>Type information</td>
<td>Additional information for functions and variables. Variables: type (int, string, ..) Functions: number and type of parameters, calling convention, return value</td>
</tr>
</tbody>
</table>

Public Symbols for MS modules (kernel32.dll, user32.dll, ..) are always stripped.
Generating Debug Information

- The build process consists of two steps
  1) compiler: generates machine instructions which are stored into .OBJ files
  2) linker: combines all available .OBJ and .LIB files into the final executable

- For Debug Information we also need two steps:
  1) compiler: generates debug information for every source file
  2) linker: combines available debug information into the final set of debug information for the executable

- Compiler options: /Z7, /Zi, /ZI
- Linker options: /debug, /pdb, /pdbstripped

Point of interest for Static libraries: Use /Z7 to store the debug information in the resulting .LIB file.

Matching Debug Information

- Signature stored into executable and PDB file during build
  For PDB 2.0 files: Time Stamp
  For PDB 7.0 files: GUID generated during build

- For a debugger match this signature must be the same

- Algorithm to search PDB files:
  1. Try module (EXE or DLL) folder
  2. Try name and path specified in the PE file (the NB10 or RSDS debug header)
  3. Try environment variables: _NT_SYMBOL_PATH and _NT_ALT_SYMBOL_PATH
Invasive vs. Noninvasive Debugging and Attaching

- **Invasive attach:**
  - DebugActiveProcess is called
  - break-in thread is created
  - prior to Windows XP: target application is killed on debugger exit or detach
  - there can be only one invasive debugger attached to a process at any time

- **Noninvasive attach:**
  - OpenProcess is called
  - no break-in thread is created
  - we don’t attach to the process as a debugger
  - all threads of the target application are frozen
  - we can change and examine memory
  - we cannot set breakpoints
  - we cannot step through the application
  - we can exit or detach the debugger without killing the target application
  - we can attach several noninvasive debuggers to a process (+ one invasive debugger)
  - useful if:
    - the target application is being debugged by Visual Studio (or any other invasive debugger), we can still attach WinDBG as a noninvasive debugger in order to get additional information
    - the target application is completely frozen and cannot launch the break-in thread necessary for a true attach
Exceptions

- A system mechanism that isn’t language specific.

- Exceptions are made accessible through language extensions.
  Example: the `__try & __except` construct in C++.

- Don’t use try-catch-except for condition checking in time critical parts of your application.
  For every exception the system creates an exception record, searches for frame based exception handlers (catch-except) through all stack frames in reverse order, and finally continues with program execution. This can result in performance degradation due to the execution of hundreds of instructions.

Exception Dispatching

1) The system first attempts to notify the process's debugger, if any
2) If the process is not being debugged, or if the associated debugger does not handle the exception (WinDbg -> `gN` == Go with Exception Not Handled), the system attempts to locate a frame-based exception handler
3) If no frame-based handler can be found, or no frame-based handler handles the exception, the UnhandledExceptionFilter makes a second attempt to notify the process's debugger. This is known as second-chance or last-chance notification.
4) If the process is not being debugged, or if the associated debugger does not handle the exception, the postmortem debugger specified in AeDebug will be started.
Exception Dispatching and SetUnhandledExceptionFilter

AeDebug? Postmortem Debugging!

- Set/Change postmortem debugger:
  - Windows - I
  - drwtsn32 - i

- Postmortem settings:
  HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug

Whatever program is specified in AeDebug is run.
No validation is made that the program is actually a debugger!
Table of Contents - Roadmap

✓ Behind the Scenes
→ Using WinDbg
  • Global Flags
  • Application Verifier
  • Process Dumps

WinDbg Commands

• Regular commands
  – are used to debug processes
  – Examples: k, lm, g

• Meta or Dot-Commands
  – usually control the behavior of the debugger
  – Examples: .sympath, .cls, .lastevent, .detach, .if

• Extension Commands
  – implemented as exported functions in extension DLLs
  – are a large part of what makes WinDbg such a powerful debugger
  – there is a set of preinstalled extension DLLs: exts.dll, ntsdexts.dll, uext.dll, wow64exts.dll, kdexts.dll, ..
  – we can write our own extension DLLs
  – Examples: !analyze, !address, !handle, !peb
Main Extensions

- !exts.help → General Extensions
- !Uext.help → User-Mode Extensions (non-OS specific)
- !Ntsdexts.help → User-Mode Extensions (OS specific)
- !Kdexts.help → Kernel-Mode Extensions
- !logexts.help → Logger Extensions
- !clr10\sos.help → Debugging Managed Code
- !wow64exts.help → Wow64 Debugger Extensions
- ..

Symbols in WinDbg

- _NT_SYMBOL_PATH environment variable must be set
  
  Example for MS symbols:

  ```
  _NT_SYMBOL_PATH=srv*C:\Symbols\MsSymbols*;http://msdl.microsoft.com/download/symbols;
  ```

  With this setting WinDbg will automatically download all needed symbols for MS components (i.e. kernel32) from the MS server.

- In WinDbg's GUI you can access symbol settings from:
  - (Menu) File → Symbol File Path ... (Ctrl+S)

- Useful Commands:
  - .sympath → get/set path for symbol search
  - .sympath +XY → append XY directory to the searched symbol path
  - !sym noisy → instructs the debugger to display information about its search for symbols
  - ld kernel32 → load symbols for kernel32.dll
  - ld * → load symbols for all modules
  - .reload → reloads symbol information
  - x kernel32!* → examine and list all symbols in kernel32
  - x kernel32!*"LoadLibrary" → list all symbols in kernel32 which contain "LoadLibrary"
  - dt ntdll!* → display all variables in ntdll
Sources in WinDbg

- _NT_SOURCE_PATH_ environment variable must be set
  
  Example: _NT_SOURCE_PATH=C:\Sources

- In WinDbg’s GUI you can access source settings from:
  - (Menu) File → Source File Path … (Ctrl+P)

- Useful Commands:
  - .srcpath → get/set path for source-file search
  - .srcpath+ XY → append XY directory to the searched source path

Important: Be sure to set up the symbols and sources for WinDbg correctly. This is the first and most important step where people new to WinDbg often fail. Note that without symbols for MS components (kernel32.dll, ntdll.dll,...) many commands in the following sections will not work.

Processes and Threads on Windows NT

- Every Windows process is represented by an executive process block (EPROCESS) in kernel-mode
- EPROCESS points to a number of related data structures; for example, each process has one or more threads represented by executive thread blocks (ETHREAD)
- EPROCESS points to a process environment block (PEB) in process address space
- ETHREAD points to a thread environment block (TEB) in process address space
PEB and TEB

- PEB = Process Environment Block
  - basic image information (base address, version numbers, module list)
  - process heap information
  - environment variables
  - command-line parameter
  - DLL search path
  - Display it: !peb, dt nt\_PEB

- TEB = Thread Environment block
  - stack information (stack-base and stack-limit)
  - TLS (Thread Local Storage) array
  - Display it: !teb, dt nt\_TEB

FACT: Many WinDbg commands (lm, !dlls, !imgreloc, !tls, !gle) rely on the data retrieved from PEB and TEB.

Example - PEB "dump"

```plaintext
0:001> dt nt\_PEB +r @$peb // @$peb = address of our process's PEB (see pseudo-register syntax)
0x00000000 InheritedAddressSpace : 0 ''
0x00000001 ReadImageFileExecOptions : 0 ''
0x00000002 BeingDebugged : 0x1 ''
 0x00000008 ImageBaseAddress : 0x00400000
 0x0000000c Ldr : 0x7d6a01e0 _PEB\_LDR\_DATA
    0x00000000 Length : 0x28
    0x00000004 Initialized : 0x1 ''
    0x00000008 SubHandle : (null)
    0x0000000c InLoadOrderModuleList : _LIST\_ENTRY [ 0x2d1eb0 - 0x2da998 ]
    0x00000010 Flink : 0x002d1eb0 _LIST\_ENTRY [ 0x2d1f08 - 0x7d6a01ec ]
    0x00000014 Blink : 0x002da998 _LIST\_ENTRY [ 0x7d6a01ec - 0x2d9f38 ]
    0x0000001c InMemoryOrderModuleList : _LIST\_ENTRY [ 0x2d1eb8 - 0x2da9a0 ]
    0x00000020 EntryInProgress : (null)
    0x00000024 ProcessParameters : 0x001c0000 _RTL\_USER\_PROCESS\_PARAMETERS
      0x00000000 MaximumLength : 0x102c
      0x00000004 Length : 0x102c
      0x00000008 Flags : 0x4001
      0x0000000c DebugFlags : 0
      0x00000010 CurrentDirectory : _CURDIR
      0x00000014 DosPath : _UNICODE\_STRING "D:\Development\Utils\"
      0x00000018 Handle : 0x00000024
      0x0000001c DllPath : _UNICODE\_STRING "C:\WINDOWS\system32;C:\WINDOWS\system;C:\WINDOWS;...
```
### WinDbg Commands for Retrieving Process and Module Information

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!peb</td>
<td>displays a formatted view of the information in the process environment block (PEB)</td>
</tr>
<tr>
<td>dt ntl!_PEB Addr</td>
<td>full PEB dump</td>
</tr>
<tr>
<td>Im</td>
<td>list loaded and unloaded modules</td>
</tr>
<tr>
<td>ImD</td>
<td>-</td>
</tr>
<tr>
<td>Im vm kernel32</td>
<td>verbose output (including image and symbol information) for kernel32</td>
</tr>
<tr>
<td>!lmi kernel32</td>
<td>similar implementation as an extension</td>
</tr>
<tr>
<td>!dlls</td>
<td>display list of loaded modules with loader specific information (entry point, load count)</td>
</tr>
<tr>
<td>!dlls -c kernel32</td>
<td>same as before for kernel32 only</td>
</tr>
<tr>
<td>!lmreloc</td>
<td>display relocation information</td>
</tr>
<tr>
<td>!ldh kernel32</td>
<td>display the headers for kernel32</td>
</tr>
</tbody>
</table>

### Example - Module Information

```plaintext
0:001> !dlls -c msvcrt
Dump dll containing 0x77ba0000:
0x002d40c0: C:\WINDOWS\system32\msvcrt.dll
  Base   0x77ba0000  EntryPoint  0x77baf78b  Size        0x0005a000
  Flags  0x80084006  LoadCount   0x00000007  TlsIndex    0x00000000
LDRP_STATIC_LINK
LDRP_IMAGE_DLL
LDRP_ENTRY_PROCESSED
LDRP_PROCESS_ATTACH_CALLED

0:001> lm vm msvcrt
start   end        module name
77ba0000 77bfa000   msvcrt     (deferred)
Image path: C:\WINDOWS\system32\msvcrt.dll
Image name: msvcrt.dll
Timestamp:        Fri Mar 25 03:33:02 2005 (4243785E)
CheckSum:         0006288A
ImageSize:        0005A000
FileVersion:      7.0.3790.1830
ProductVersion:   6.1.8638.1830
...               
CompanyName:      Microsoft Corporation
ProductName:      Microsoft® Windows® Operating System
InternalName:     msvcrt.dll
OriginalFilename: msvcrt.dll
ProductVersion:   7.0.3790.1830
FileVersion:      7.0.3790.1830 (srv03_sp1_rtm.050324-1447)
FileVersion:      7.0.3790.1830 (srv03_mpl_xtm.050324-1447)
FileDescription:  Windows NT CRT DLL
LegalCopyright:   © Microsoft Corporation. All rights reserved.
```
WinDbg Commands for Retrieving Thread Information

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>thread status for all threads</td>
</tr>
<tr>
<td>~0</td>
<td>thread status for thread 0</td>
</tr>
<tr>
<td>~.</td>
<td>thread status for currently active thread</td>
</tr>
<tr>
<td>~*</td>
<td>thread status for all threads with some extra info (priority, StartAdres)</td>
</tr>
<tr>
<td>~* k</td>
<td>call stacks for all threads ~ !uniqstack</td>
</tr>
<tr>
<td>~&lt;thread&gt;s</td>
<td>set current thread</td>
</tr>
<tr>
<td>!gle</td>
<td>Get last error</td>
</tr>
<tr>
<td>!runaway</td>
<td>displays information about time consumed by each thread \quick way to find out which threads are spinning out of control or consuming too much CPU time</td>
</tr>
<tr>
<td>!teb</td>
<td>displays a formatted view of the information in the thread environment block (TEB)</td>
</tr>
<tr>
<td>dt nt !_TEB Addr</td>
<td>full TEB dump</td>
</tr>
</tbody>
</table>

Example - Threads

```
0:001> !runaway 7
User Mode Time
Thread     Time
0:d28     0 days 0:00:00.015
1:2b0     0 days 0:00:00.000

Kernel Mode Time
Thread     Time
0:d28     0 days 0:00:00.093
1:2b0     0 days 0:00:00.000

Elapsed Time
Thread     Time
0:d28     0 days 0:04:04.156
1:2b0     0 days 0:03:53.328
```

```
0:000> ~*
  0 Id: dac.d28 Suspend: 1 Teb: 7efdd000 Unfrozen
    Start: TestApp!ILT+1415(_wWinMainCRTStartup) (0041158c) Priority: 0 Priority class: 32 Affinity: 3

  1 Id: dac.2b0 Suspend: 1 Teb: 7efda000 Unfrozen
    Start: 00000001 Priority: 0 Priority class: 32 Affinity: 3
```

```
0:000> !gle
LastErrorValue: (Win32) 0 (0) - The operation completed successfully. LastStatusValue: (NTSTATUS) 0 - STATUS_WAIT_0
```
Windows and Menus in WinDbg

WinDbg’s windows can be docked or floating.

1) Docked Windows = the preferred way of using windows
   • Shrink and grow with the WinDbg frame
   • Are positioned and sized relatively to each other as the frame changes
   • Can be tabbed. Tabbed windows are overlaid
   • WinDbg supports multiple docks (handy for a multi-monitor system)
   • Ctrl-Tab iterates through all windows in all docks

2) Undocked or floating windows
   • Are always on top of the WinDbg window

Each window in WinDbg has its own menu.
   • Menus can be accessed by:
     • left-click on the menu button (next to the close button)
     • right-click on the title bar of a window
     • right-click on the tab of a tabbed window
   • Be sure to check these menus. They are often hiding interesting features.

Example of a Running Instance of WinDbg
Debugger Markup Language (DML)

- DML allows debugger output to include directives and extra non-display information in the form of tags
- Debugger user interfaces parse out the extra information to provide new behaviors
- DML is primarily intended to address the following issues:
  - Linking of related information
  - Discoverability of debugger and extension functionality
  - Enhancing output from the debugger and extensions
- DML was introduced with version 6.6.0.7 of Debugging Tools

<table>
<thead>
<tr>
<th>DML Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.dml_start</td>
<td>Kick off to other DML commands</td>
</tr>
<tr>
<td>.prefer_dml 1</td>
<td>Global setting: all DML-enhanced commands will produce DML output</td>
</tr>
<tr>
<td>.help /D a*</td>
<td>.help has a new DML mode where a top bar of links is given</td>
</tr>
<tr>
<td>.chain /D</td>
<td>.chain has a new DML mode where extensions are linked to a .extmatch</td>
</tr>
<tr>
<td></td>
<td>Check &quot;..\Debugging Tools for Windows\dml.doc&quot; for more commands.</td>
</tr>
</tbody>
</table>

DML in WinDbg

- Note that you can click on any “link”
- If you right-click on it, you can even start the command in a new window
Memory: Stack Details

- From MSDN:
  - Each new thread receives its own stack space, consisting of both committed and reserved memory.
  - By default, each thread uses 1 Mb of reserved memory, and one page of committed memory.
  - The system will commit one page block from the reserved stack memory as needed. (see MSDN CreateThread > dwStackSize > "Thread Stack Size").

Example - Stack Size for a Thread

```
0:000> !tcb
TEB at 7fffd000
  ExceptionList: 0012F784
  StackBase: 00130000
  StackLimit: 0012C000
...
0:000> dt ntdll!_TEB DeallocationStack 7fffd000
  +0xe0c DeallocationStack : 0x00030000
0:000> !address esp
  AllocBase : SubRegionBase - SubRegionSize
  00030000 : 0012c000 - 00004000
    Type : 00000000 MEM_PRIVATE
    Protect : 00000004 PAGE_READWRITE
    State : 00000000 MEM_COMMIT
    Usage : RegionUsageStack
    Pid.Tid : e34.e78
0:000> ? 00130000 - 0012c000
  Evaluate expression: 16384 = 00004000
0:000> ? 00130000 - 00030000
  Evaluate expression: 1048576 = 00100000

0x004000 ➔ Our thread has 4 pages or 16KB of committed memory.
0x100000 ➔ Our thread has 256 pages or 1MB of reserved memory.
```
Memory: Stack Growth

- The ESP register points to the current stack location of a thread.
- If a program attempts to access an address within a guard page, the system raises a STATUS_GUARD_PAGE_VIOLATION (0x80000001) exception. A guard page provides a one-shot alarm for memory page access.
- If a stack grows until the end of reserved memory, a STATUS_STACK_OVERFLOW is raised.

Example - Stack Growth

```
0:000> !teb
TEB at 7ffdf000
ExceptionList: 0012f784
StackBase: 00130000
StackLimit: 0012c000
...

0:000> dt ntdll!_TEB DeallocationStack 7ffdf000
+0x0c DeallocationStack : 0x00030000

0:000> ? 00130000 - 0012c000
Evaluate expression: 16384 = 0x004000
0x004000 ➔ Our thread has 4 pages or 16KB of committed memory.
0x100000 ➔ Our thread has 256 pages or 1MB of reserved memory.

-------------------------------------------------------------------

0:000> !teb
TEB at 7ffdf000
ExceptionList: 0012f784
StackBase: 00130000
StackLimit: 0012c000
...

0:000> ? 00130000 - 00033000
Evaluate expression: 1036288 = 0x0fd000
0x0fd000 ➔ Now our thread has 253 pages of committed memory.
The system will throw a stack-overflow exception if another page will be requested.
```
WinDbg Commands for Retrieving Call-Stack Information

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!uniqstack</td>
<td>displays call-stacks for all of the threads in the current process</td>
</tr>
<tr>
<td>!findstack MySymbol</td>
<td>locates all call-stacks that contain MySymbol</td>
</tr>
<tr>
<td>k</td>
<td>display call stack for current thread</td>
</tr>
<tr>
<td>kP</td>
<td>P == full parameters for each function called</td>
</tr>
<tr>
<td>kf</td>
<td>f == distance between adjacent frames to be displayed (useful to check stack consumption of each frame)</td>
</tr>
<tr>
<td>kv</td>
<td>v == display FPO information + calling convention</td>
</tr>
<tr>
<td>kb</td>
<td>b == display the first three parameters passed to each function</td>
</tr>
<tr>
<td>kM</td>
<td>Output in DML format; frame numbers link to a .frame/dv command which displays locals for the frame</td>
</tr>
</tbody>
</table>

Example - UniqStack

```
0:000> !uniqstack
Processing 2 threads, please wait.
.
. 0 Id: dac.154c Suspend: 1 Tab: 7e6dd000 Unfrozen
   Start: TestApp\ILYv14\[\wWinMainCRTStartup] (0041158c)
   Priority: 0 Priority class: 32 Affinity: 3
ChildEBP RetAddr
002df44c 00411eeb ntdll!DbgBreakPoint
002df52c 783c2100 TestApp\CMyDialog::OnBtnClicked_ExecuteBreakPoint+0x2b [d:\TestApp\MyDialog.cpp @ 72]
002df570 783c2442 NTPC80UD!_KfxDispatchCmdMsg+0x2b0
002df5d4 783d67f1 NTPC80UD!CCmdTarget::OnCmdMsg+0x2e2
002df610 7836142d NTPC80UD!CDialog::OnCmdMsg+0x21
.
. 1 Id: dac.127c Suspend: 1 Tab: 7e6da000 Unfrozen
   Start: 00000001
   Priority: 0 Priority class: 32 Affinity: 3
ChildEBP RetAddr
0242f550 7d62633f ntdll!NtQueryAttributesFile+0x299 [f:\\ap\\vtools\\crt\bld\self_x86\\crt\arc\crtxtex.c @ 589]
0242f0c0 7d62955a TestApp\WWinMainCRTStartup+0x2d [f:\\ap\\vtools\\crt\bld\self_x86\\crt\arc\crtxtex.c @ 414]
0242fff0 00000000 ntdll!DbgUiRemoteBreakin+0x41
.
Total threads: 2
```
WinDbg Commands for Memory Handling

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>d, dd, da, du, ..</td>
<td>Display memory</td>
</tr>
<tr>
<td></td>
<td>dd == double word values</td>
</tr>
<tr>
<td></td>
<td>da == display ASCII characters</td>
</tr>
<tr>
<td></td>
<td>du == display Unicode characters</td>
</tr>
<tr>
<td>f</td>
<td>fill memory</td>
</tr>
<tr>
<td>!vprot MyAddr</td>
<td>Displays virtual memory protection information for MyAddr</td>
</tr>
<tr>
<td>!address MyAddr</td>
<td>Display information (type, protection, usage, ..) about the memory specified by MyAddr</td>
</tr>
<tr>
<td>!address -RegionUsageStack</td>
<td>Display stack regions for all threads in the process</td>
</tr>
<tr>
<td>dds</td>
<td>Display Words and Symbols</td>
</tr>
<tr>
<td>ddp</td>
<td>Display Referenced Memory. If a match to a known symbol is found, this symbol is displayed as well.</td>
</tr>
</tbody>
</table>

Example – Process’s Memory Information

```
0:000> !address
00000000 : 00000000 - 00010000
  Type  00000000
  Protect 00000001 PAGE_NOACCESS
  State  00010000 MEM_FREE
  Usage RegionUsageFree
00010000 : 00010000 - 00001000
  Type  00020000 MEM_PRIVATE
  Protect 00000004 PAGE_READWRITE
  State  00001000 MEM_COMMIT
  Usage RegionUsageImage

-------------------- Usage SUMMARY --------------------------
 TotSize (      KB)   Pct(Tots) Pct(Busy)   Usage
7b9b1000 ( 2025156) : 96.57%    00.00%    : RegionUsageFree
12e2000 (   19336) : 00.92%  26.88%    : RegionUsageImage
1950000 (   25920) : 01.24%  36.03%    : RegionUsageIsVAD
1745000 (   23828) : 01.14%  31.81%    : RegionUsageHeap
1658000 (   22880) : 01.09%  31.81%    : RegionUsagePageHeap
1000 (  4) : 00.00%  00.00%    : RegionUsageTeb
1000 (  4) : 00.00%  00.00%    : RegionUsagePageHeap
1000 (  4) : 00.00%  00.00%    : RegionUsagePageHeap
1000 (  4) : 00.00%  00.00%    : RegionUsageHeap
1000 (  4) : 00.00%  00.00%    : RegionUsageHeap
1000 (  4) : 00.00%  00.00%    : RegionUsageHeap
1000 (  4) : 00.00%  00.00%    : RegionUsageHeap
1000 (  4) : 00.00%  00.00%    : RegionUsageHeap
1000 (  4) : 00.00%  00.00%    : RegionUsageHeap
1000 (  4) : 00.00%  00.00%    : RegionUsageHeap
1000 (  4) : 00.00%  00.00%    : RegionUsageHeap
-------------------- State SUMMARY --------------------------
 TotSize (      KB)   Pct(Tots)  Usage
25a000 (   48104) : 02.29%    : MEM_COMMIT
7b9b1000 ( 2025156) : 96.57%    : MEM_FREE
1745000 (   23828) : 01.14%    : MEM_RESERVE
```

---

www.windbg.info
WinDbg Commands for Retrieving Heap Information

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!heap -?</td>
<td>Brief help</td>
</tr>
<tr>
<td>!heap -h</td>
<td>List heaps with index and range (= startAddr, endAddr)</td>
</tr>
<tr>
<td>!heap -s 0</td>
<td>Summary for all heaps = reserved and committed memory, ..</td>
</tr>
<tr>
<td>!heap -flt s Size</td>
<td>Dump info for allocations matching Size</td>
</tr>
<tr>
<td>!heap -stat</td>
<td>Dump HeapHandle list HeapHandle = value returned by HeapCreate or GetProcessHeap</td>
</tr>
<tr>
<td>!heap -stat -h 0</td>
<td>Dump usage statistic for every AllocSize = AllocSize, #blocks, and TotalMem for each AllocSize</td>
</tr>
<tr>
<td>!heap -p</td>
<td>GFlags settings, HeapHandle list</td>
</tr>
<tr>
<td>!heap -p -all</td>
<td>Details of all allocations in all heaps in the process = all HeapAlloc calls listed</td>
</tr>
<tr>
<td>!heap -p -a UserAddr</td>
<td>Details of heap allocation containing UserAddr (i.e. the address returned by HeapAlloc). Prints back traces when available.</td>
</tr>
</tbody>
</table>

More Heap Structs

If page heap is disabled for your application, then the following structs apply.
Note that page heap is disabled by default.

- **_HEAP struct**
  - Defined in ntdll.dll: dt ntdll::_HEAP
  - For every HeapCreate there is a unique _HEAP
  - You can use "!heap -p -all" to get addresses for all _HEAP structs in your process

- **_HEAP_ENTRY struct**
  - Defined in ntdll.dll: dt ntdll::_HEAP_ENTRY
  - For every HeapAlloc there is a unique _HEAP_ENTRY
  - You can use "!heap -p -all" to get addresses for all heap entries in your process
If page heap is **enabled** for your application, then the following structs apply. You can enable page heap with Global Flags (`gflags.exe`).

- **_DPH_HEAP_ROOT struct**
  - Defined in `ntdll.dll`: `dt ntdll!_DPH_HEAP_ROOT`
  - For every `HeapCreate` there is a unique `_DPH_HEAP_ROOT`
  - You can use `!heap -p -all` to get addresses for all heap roots in your process
    - Usually address of a `_DPH_HEAP_ROOT` = value of HeapHandle + 0x1000

- **_DPH_HEAP_BLOCK struct**
  - Defined in `ntdll.dll`: `dt ntdll!_DPH_HEAP_BLOCK`
  - For every `HeapAlloc` there is a unique `_DPH_HEAP_BLOCK`
  - You can use `!heap -p -all` to get addresses for all heap blocks in your process
Who called HeapAlloc?

- Enable stack traces and page heap for your application
  - Start GFlags, select “Create user mode stack trace database” and “Enable page heap” for your image
  - Or from the command line: gflags.exe /i <IMAGE.EXE> +ust +hpa
- Restart your application and attach WinDbg

From WinDbg’s command line:

- `!heap -p -a <UserAddr>`
  - `<UserAddr>` = address of our allocation (returned by HeapAlloc, new, ...)
  - Will dump the call-stack but without source information
- `dt ntdll!_DPH_HEAP_BLOCK StackTrace <MyHeapBlockAddr>`
  - `<MyHeapBlockAddr>` = DPH_HEAP_BLOCK address retrieved in previous step
  - StackTrace = member of DPH_HEAP_BLOCK which stores the call stack for our HeapAlloc
- `dds <StackTrace>`
  - `<StackTrace>` = value retrieved in previous step
  - dds will dump the call-stack with source information included

Example - Who called HeapAlloc?

```
// HeapAlloc: 0x00150000, dwBytes -0x0A000000 |= 0x025F1000
0:000> !heap -p -a 0x025F1000
address 0x025F1000 found in
  _DPH_HEAP_ROOT @ 151000
in busy allocation (  DPH_HEAP_BLOCK:  UserAddr  UserSize  VirtAddr  VirtSize)
  15449c:  25f1000  a00000  25f0000  a02000
    7c911b29 ntdll!RtlAllocateHeap+0x00000e64
    0045b8b1 TestApp!CMyDlg::OnBnClicked_HeapAlloc+0x00000051
    004016e0 TestApp!_AfxDispatchCmdMsg+0x043
    004018ed TestApp!CCmdTarget::OnCmdMsg+0x0118
    00408f7f TestApp!CDialog::OnCmdMsg+0x00000118
    ...  
0:000> dt ntdll!_DPH_HEAP_BLOCK StackTrace 15449c
  *0x02a4 StackTrace : 0x0238e328 _RTL_TRACE_BLOCK
0:000> dds 0x0238e328
0238e328: abcdefgh
0238e334 00000001
0238e338 00a00000
0238e33c 00151000
0238e340 01b17b1c
0238e344 0238e348
0238e348 0238e34c 7c91b298 ntdll!RtlAllocateHeap+0xe64
0238e34c 0045b8b1 TestApp!CMyDlg::OnBnClicked_HeapAlloc+0x51 [d:\development\sources\TestApp\MyDlg.cpp @ 366]
0238e350 004016e0 TestApp!_AfxDispatchCmdMsg+0x43 [f:\sp\vctools\vc7\libs\ship\atlmfc\src\mfc\cmdtarg.cpp @ 82]
0238e354 004018ed TestApp!CCmdTarget::OnCmdMsg+0x118 [f:\sp\vctools\vc7\libs\ship\atlmfc\src\mfc\cmdtarg.cpp @ 381]
0238e358 00408f7f TestApp!CDialog::OnCmdMsg+0x1b [f:\sp\vctools\vc7\libs\ship\atlmfsrc\src\mfc\dlgcore.cpp @ 85]
    ...  
```
Who called HeapCreate?

- Enable stack traces and page heap for your application
  - Start GFlags, select "Create user mode stack trace database" and "Enable page heap" for your image
  - Or from the command line: gflags.exe /i <IMAGE.EXE> +ust +hpa
- Restart your application and attach WinDbg

From WinDbg's command line:
- !heap -p -h <HeapHandle>
  - <HeapHandle> = value returned by HeapCreate
  - You can do a "heap -stat" or "heap -p" to get a list of heaps for your process and their handles
- dt ntdll!_DPH_HEAP_ROOT CreateStackTrace <MyHeapRootAddr>
  - <MyHeapRootAddr> = DPH_HEAP_ROOT address retrieved in previous step
  - CreateStackTrace = member of DPH_HEAP_ROOT which stores the call stack for our HeapCreate call
- dds <CreateStackTrace>
  - <CreateStackTrace> = value retrieved in previous step
  - dds will dump the call-stack with source information included

Example - Who called HeapCreate?

```c
// HeapCreate( 0x0000000A, 0, 0 ) --> 0x03000000;
0:000> !heap -p -h 0x03000000
_DPH_HEAP_ROOT @ 3001000
Freed and decommitted blocks
_DPH_HEAP_BLOCK : VirtAddr VirtSize
Busy allocations
_DPH_HEAP_BLOCK : UserAddr UserSize - VirtAddr VirtSize
...
0:000> dt ntdll!_DPH_HEAP_ROOT CreateStackTrace 3001000
+0x08c CreateStackTrace : 0x0238e328 _RTL_TRACE_BLOCK
0:000> dds 0x0238e328
```

```
0238e328 aabcdaaa
0238e32c 00000001
0238e330 00000010
0238e334 00000000
0238e338 00000000
0238e33c 00000000
0238e340 00000000
0238e344 0238e348
0238e348 7c93a874 ntdll!RtlCreateHeap+0x41
0238e34c 7c812bff kernel32!HeapCreate+0x55
0238e350 00460b41 TestApp!OnClicked_HostHeapCreate+0x31 [d:\development\sources\TestApp\MyDlg.cpp @ 345]
0238e354 0040b122 TestApp!AfxDispatchCmMgMsg+0x33 [f:\ap\vtools\vc7\lib\ship\atlmfc\src\mfc\cmdmrg.cpp @ 82]
0238e358 0040b32f TestApp!CCmdTarget::OnCmdMsg+0x118 [f:\ap\vtools\vc7\lib\ship\atlmfc\src\mfc\cmdtarg.cpp @ 381]
0238e35c 00408838 TestApp!CDialog::OnCmdMsg+0x1b [f:\ap\vtools\vc7\lib\ship\atlmfc\src\mfc\dlgcore.cpp @ 85]
```
Finding Memory Leaks on the Heap

- **!address –summary**
  - Summary about memory usage for your process. If `RegionUsageHeap` or `RegionUsagePageHeap` is growing constantly, then you might have a memory leak on the heap. Proceed with the following steps.

- Enable stack traces and page heap for your application
- Restart your application and attach WinDbg

From WinDbg’s command line:
- **!heap –stat –h 0**
  - Will list down handle specific allocation statistics for every AllocSize. For every AllocSize the following is listed: AllocSize, #blocks, and TotalMem.
  - Take the AllocSize with maximum TotalMem.
- **!heap –flt –s <size>**
  - `<size>` = size being allocated by HeapAlloc. Value retrieved in previous step.
- **!heap -p -a <UserAddr>**
  - `<UserAddr>` = address of our allocation (returned by HeapAlloc, new ..)
  - Will dump the call-stack but without source information. Check the “Who called HeapAlloc?” slide for how to proceed to get a call-stack with source information included.

---

**Example - Finding Memory Leaks on the Heap**

```
0:001> !heap –stat –h 0
Allocations statistics for heap @ 00150000
group-by: TOTSIZE max-display: 20
<table>
<thead>
<tr>
<th>size</th>
<th>#blocks</th>
<th>total</th>
<th>( %) (percent of total busy bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>101</td>
<td>-1010000</td>
<td>(99.99)</td>
</tr>
<tr>
<td>928</td>
<td>2</td>
<td>-1250</td>
<td>(0.00)</td>
</tr>
<tr>
<td>64</td>
<td>24</td>
<td>-e10</td>
<td>(0.00)</td>
</tr>
</tbody>
</table>
...    |         |         |                                   |
0:001> !heap –flt s 100000
| _DPH_HEAP_ROOT @ 151000 |
| Freed and decommitted blocks |
| DPH_HEAP_BLOCK ; VirtAddr VirtSize |
| Busy allocations |
| _DPH_HEAP_BLOCK ; UserAddr UserSize - VirtAddr VirtSize |
| 024F0698 | 13831000 | 00100000 - 13830000 00102000 |
| 024F0620 | 13721000 | 00100000 - 13720000 00102000 |
...    |         |         |                                   |
0:001> !heap -p -a 13831000
address 13831000 found in _DPH_HEAP_ROOT @ 151000
in busy allocation ( _DPH_HEAP_BLOCK; UserAddr UserSize - VirtAddr VirtSize)
| 24F0698 | 13831000 100000 - 13830000 102000 |
| 7c91b298 ntdll!RtlAllocateHeap+0x00000e64 |
| 0045b74e TestApp!CMyDlg ::OnBnClicked_DoMemoryLeak+0x0000003e |
| 0040b122 TestApp!_AfxDispatchCmdMsg+0x0000043 |
| 0040b32f TestApp!CCmdTarget ::OnCmdMsg+0x00000118 |
| 00408838 TestApp!CDialog ::OnCmdMsg+0x000001b |
```

---

www.windbg.info
Critical Section Related Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!locks</td>
<td>displays a list of locked critical sections for the process</td>
</tr>
<tr>
<td>!locks -v</td>
<td>display all critical sections for the process</td>
</tr>
<tr>
<td>!cs [Opt] [CsAddr]</td>
<td>Displays one or more critical sections, or the entire critical section tree.</td>
</tr>
<tr>
<td>!avrf -cs</td>
<td>Display a list of deleted critical sections (DeleteCriticalSection API)</td>
</tr>
</tbody>
</table>

Options:
- -l == display only locked sections
- -s == causes each CS's initialization stack to be displayed
- -o == causes the owner's stack to be displayed
- -t == display critical section tree → EnterCnt, WaitCnt, ...

Example – Critical Section

```
0:000> !cs -s -o 0x0012fe08
-----------------------------------------
Critical section   = 0x0012fe08 (+0x12FE08)
DebugInfo          = 0x031c4fe0
LOCKED
LockCount          = 0x0
OwningThread       = 0x00000c8c

OwningThread Stack =
ChildEBP RetAddr Args to Child
0012f488 004badd9 0012f810 02854f10 00000000 ntdll!DbgBreakPoint
0012f568 0054f62c 00000000 004bb621 00685a70 TestApp!MyDialog::OnBnClicked_EnterCs+0x39
0012f598 00550365 0012fd78 0000001c 00000000 TestApp!_AfxDispatchCmdMsg+0x9c
0012f5f0 005517f1 0000001c 00000000 00000000 TestApp!CCmdTarget::OnCmdMsg+0x285
...

Stack trace for DebugInfo (Initialization Stack) = 0x031c4fe0:
0x7c911a93: ntdll!RtlInitializeCriticalSectionAndSpinCount+0xC9
0x004c101d: TestApp!CCriticalSection::Init+0x3D
0x004c10a0: TestApp!CCriticalSection::CCriticalSection+0x40
...
```

```
0:000> !cs -t
Verifier package version >= 3.00
Tree root 02fd8fd0
Level Node CS Debug InitThr EnterThr WaitThr TryEnThr LeaveThr EnterCnt WaitCnt
0 02f88f40 0012fe08 031c4fe0 c8c c8c 0 0 0 1 0
1 02f88f40 06607f4 03148fe0 c8c 0 0 0 0 0
2 02f88f40 06607f4 02855fe0 c8c c8c 0 0 c8c 4848 0
...```
Other useful WinDbg Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dt</td>
<td>Display information about a local variable, function parameter, global variable or data type</td>
</tr>
<tr>
<td>dt ntdll!&quot;peb&quot;*</td>
<td>List all ntdll.dll variables which contain the word peb</td>
</tr>
<tr>
<td>dt ntdll!_PEB</td>
<td>Display type for PEB</td>
</tr>
<tr>
<td>dt ntdll!_PEB 7efde000</td>
<td>Dump PEB at address 7efde000</td>
</tr>
<tr>
<td>dv</td>
<td>Display local variables</td>
</tr>
</tbody>
</table>
| dv /t /i /V | Display local variables  
/t == classify them into categories (parameters or locals) 
/i == show addresses and offsets for the relevant base frame register (usually EBP) 
/V == display type information |

Example – dt & dv

```
0:000> dt TestApp!CMyDialog
+0x000 __VPN_table : Ptr32
  +0x000000 classCObject : CRuntimeClass
  +0x000000 classCmdTarget : CRuntimeClass
  +0x000000 _commandEntries : [0] AFX_OLECMDMAP_ENTRY
  +0x000000 commandMap : AFX_OLECMDMAP
  +0x000000 _dispatchEntries : [0] AFX_DISPMAP_ENTRY

+0x004 m_dwRef : Int4B
+0x008 m_pOuterUnknown : Ptr32 IUnknown
+0x00c m_pInnerUnknown : Uint4B
+0x010 m_pDispatch : CCmdTarget::XDispatch
+0x014 m_pResultExpected : Int4B
+0x018 m_pConnPtContainer : CCmdTarget::XConnPtContainer
+0x01c m_pModuleState : Ptr32 AFX_MODULE_STATE
+0x020 m_hWnd : Ptr32 HWND__
+0x024 __ipDispatchInit : Ptr32 Void
+0x028 __pParentWnd : Ptr32 CWnd
+0x02c __hWndTop : Ptr32 HWND__
+0x070 pGdiCallInfo : Ptr32 _AFX_OLECALLINFO
+0x074 hIcon : HICON__
+0x078 nn : Int4B

0:000> dv /t /i /V
prev local 002df440 @ebp-0x08 class CMyDialog * this = 0x002dfe24
prev param 002df450 @ebp-0x08 int nn = 1
```
Pseudo-Registers in WinDbg

- Virtual registers provided by the debugger
- Begin with a dollar sign ($)

1) Automatic pseudo-registers
   - are set by the debugger to certain useful values
   - examples: $ra, $peb, $teb, ..

2) User-defined pseudo-registers
   - there are twenty user-defined registers: $t0, $t1, $t2, .., $t19
   - integer variables that can be used to store intermediate data
   - can additionally hold type-information
   - $r? assigns a typed result to an lvalue
     - $r? $t0 = @peb->ProcessParameter
       - Assigns a typed value to $t0
       - $t0’s type is remembered so it can be used in further expressions
     - ?? @$t0->CommandLine

Automatic Pseudo-Registers

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ra</td>
<td>Return address currently on the stack. Useful in execution commands, i.e.: “g $ra”</td>
</tr>
<tr>
<td>$ip</td>
<td>The instruction pointer</td>
</tr>
<tr>
<td></td>
<td>x86 = EIP, Itanium = IIP, x64 = RIP</td>
</tr>
<tr>
<td>$exentry</td>
<td>Entry point of the first executable of the current process</td>
</tr>
<tr>
<td>$retreg</td>
<td>Primary return value register</td>
</tr>
<tr>
<td></td>
<td>X86 = EAX, Itanium = ret0, x64 = rax</td>
</tr>
<tr>
<td>$csp</td>
<td>Call stack pointer</td>
</tr>
<tr>
<td></td>
<td>X86 = ESP, Itanium = BSP, x64 = RSP</td>
</tr>
<tr>
<td>$peb</td>
<td>Address of the process environment block (PEB)</td>
</tr>
<tr>
<td>$tpeb</td>
<td>Address of the thread environment block (TEB) of current thread</td>
</tr>
<tr>
<td>$tpid</td>
<td>Process ID (PID)</td>
</tr>
<tr>
<td>$tid</td>
<td>Thread ID (tID)</td>
</tr>
<tr>
<td>$ptrsize</td>
<td>Size of a pointer</td>
</tr>
<tr>
<td>$pagesize</td>
<td>Number of bytes in one page of memory</td>
</tr>
<tr>
<td>...</td>
<td>See “Pseudo-Registry Syntax” in WinDbg’s help.</td>
</tr>
</tbody>
</table>
Expressions in WinDbg

1) MASM expressions
   - evaluated by the ? command
   - each symbol is treated as an addresses (the numerical value of a symbol is the memory address of that symbol → to get its value you must dereference it with poi)
   - source line expressions can be used (`myfile.c:43`)
   - the at sign for register values is optional (eax or @eax are both fine)
   - used in almost all examples in WinDbg’s help
   - the only expression syntax used prior to WinDbg version 4.0 of Debugging Tools

2) C++ expressions
   - evaluated by the ?? command
   - symbols are understood as appropriate data types
   - source line expressions cannot be used
   - the at sign for register values is required (eax will not work)

MASM operations are always byte based. C++ operations follow C++ type rules (including the scaling of pointer arithmetic). In both cases numerals are treated internally as ULON64 values.

More About Expressions

- **MASM:**
  - The numerical value of any symbol is its memory address
  - Any operator can be used with any number
  - Numerals: are interpreted according to the current radix: n [8 | 10 | 16] Can be overridden by a prefix: 0x (hex), 0n (decimal), 0t (octal), 0y (binary)

- **C++:**
  - The numerical value of a variable is its actual value
  - Operators can be used only with corresponding data types
  - A symbol that does not correspond to a C++ data type will result in a syntax error
  - Data structures are treated as actual structures and must be used accordingly. They do not have numerical values.
  - The value of a function name or any other entry point is the memory address, treated as a function pointer
  - Numerals: the default is always decimal Can be overridden by a prefix: 0x (hex), 0 (zero- octal)
Example – Value of a variable

```c
void MyFunction() {
    int nLocalVar = 7;
    ..
}
```

```c
0:000> dd nLocal1 L1
0012f830 00000007
```

// MASM syntax

```c
0:000> ? nLocalVar // get address (memory location) of nLocalVar
Evaluate expression: 1243184 = 0012f830
```

```c
0:000> ? dwo(nLocalVar) // get value of nLocalVar - dereference it
Evaluate expression: 7 = 00000007 // (dwo = double-word, poi = pointer sized data)
```

```c
0:000> ? poi(nLocalVar)
Evaluate expression: 7 = 00000007
```

// C++ syntax

```c
0:000> ?? nLocalVar // get value of nLocalVar
int 7
```

```c
0:000> ?? & nLocalVar // get address (memory location) of nLocalVar
int * 0x0012f830
```

Example – MASM vs. C++ Expressions

```c
// -------------------------------------------------------------
// The following examples will return:
//   eax == ebx ? 0
//   eax >  ebx ? 1
//   eax <  ebx ? -1
// -------------------------------------------------------------
```

```c
0:000> r eax = 4, ebx = 3
0:000> ? 0*(eax = ebx) + 1*(eax > ebx) + -1*(eax < ebx)
Evaluate expression: 1 = 00000001
```

```c
0:000> ?? 0*(int)(@eax == @ebx) + 1*(int)(@eax > @ebx) + -1*(int)(@eax < @ebx)
int 1
```

```c
0:000> r eax = 3, ebx = 4
0:000> ? 0*(eax = ebx) + 1*(eax > ebx) + -1*(eax < ebx)
Evaluate expression: -1 = ffffffff
```

```c
0:000> ?? 0*(int)(@eax == @ebx) + 1*(int)(@eax > @ebx) + -1*(int)(@eax < @ebx)
int -1
```
Common Numeric MASM Operators

Unary operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dwo, qwo, poi</td>
<td>dwo = dword from specified address; qwo = qword from specified address; poi = pointer-size data from specified address</td>
</tr>
<tr>
<td>wo, by</td>
<td>wo = low-order word from specified address by = low-order byte from specified address</td>
</tr>
</tbody>
</table>

Binary operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>= (or ==), !=</td>
<td>Equal to, not equal to</td>
</tr>
<tr>
<td>&lt;, &gt;, &lt;=, &gt;=</td>
<td>Less than, greater than, less than or equal to, greater or equal to</td>
</tr>
<tr>
<td>and (or &amp;), xor (or ^), or (or</td>
<td>)</td>
</tr>
<tr>
<td>+, -, *, /</td>
<td>Addition, subtraction, multiplication, division</td>
</tr>
<tr>
<td>&lt;&lt;, &gt;&gt;, &gt;&gt;&gt;</td>
<td>Left shift, right shift, arithmetic right shift</td>
</tr>
</tbody>
</table>

Some Non-Numeric Operators in MASM

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$iment( Address)</td>
<td>Returns the image entry point. Address = image base address</td>
</tr>
<tr>
<td>$scmp( &quot;String1&quot;, &quot;String2&quot;)</td>
<td>Evaluates to -1, 0, or 1. See strcmp.</td>
</tr>
<tr>
<td>$sicmp( &quot;String1&quot;, &quot;String2&quot;)</td>
<td>Evaluates to -1, 0, or 1. See stricmp.</td>
</tr>
<tr>
<td>$spat ( &quot;String&quot;, &quot;Pattern&quot;)</td>
<td>TRUE → String matches Pattern; FALSE → String doesn’t match Pattern; Pattern = can be an alias or string constant but not a memory pointer (i.e. you cannot use a “poi (address)” directly with $spat. You must save the result into an alias first). Pattern may contain a variety of wildcard specifiers.</td>
</tr>
<tr>
<td>$vvalid(Address, Length)</td>
<td>1 → memory in the given range is valid 0 → memory is invalid</td>
</tr>
</tbody>
</table>
Optimizations

To avoid unnecessary symbol lookup time:

- **MASM:**
  - The usage of @ for registers is recommended. Otherwise they may be interpreted as symbols.

- **C++:**
  - Prefix for local symbols: $!MySymbol
  - Prefix for global symbols: <moduleName>!MySymbol

Example – Structs in C++ Syntax

```plaintext
// For better performance:
// $!symName             … for local symbols
// <ModuleName>!symName  … for global symbols

0:000> ?? dlg.m_nn
int 0
0:000> ?? $!dlg.m_nn
int 0
0:000> ?? sizeof($!dlg.m_nn)
unsigned int 0xac
0:000> ?? ((MyModule!CMyDlg*) 0x12f878)->m_nn
int 0

0:000> ?? ((ntdll!_TEB*) 0x7ffdf000)->ClientId
struct _CLIENT_ID                           // The C++ expression evaluator casts
+0x000 UniqueProcess    : 0x000017d8     // UniqueProcess    : 0x000017d8
+0x004 UniqueThread     : 0x00000ea8     // UniqueThread     : 0x00000ea8

0:001> r? $t0 = @$peb->ProcessParameters   // Note that type information is preserved
struct _UNICODE_STRING
+0x000 Length           : 0x6a
+0x002 MaximumLength    : 0x6c
+0x004 Buffer           : 0x00020724

// Or

0:001> ?? @$teb->ClientId
struct _CLIENT_ID                           // pseudo-registers to their appropriate types
+0x000 UniqueProcess    : 0x000017d8     // UniqueProcess    : 0x000017d8
+0x004 UniqueThread     : 0x00000ea8     // UniqueThread     : 0x00000ea8
```

```plaintext
// "D:\Development\Sources\CrashMe\release\CrashMe.exe"
+0x000 Length : 0x6a
+0x002 MaximumLength : 0x6c
+0x004 Buffer : 0x00020724 "D:\Development\Sources\CrashMe\release\CrashMe.exe"
```
Example – Pointer Arithmetic

```
// int myInt[2] = { 1,2 };
// Note that MASM operations are always byte based,
// whereas pointer arithmetic is used for c++ operations.

// MASM syntax

0:000> ? myInt
Evaluate expression: 1243256 = 0012f878

0:000> ? dwo(myInt)
Evaluate expression: 1 = 00000001

0:000> ? myInt+4
Evaluate expression: 1243260 = 0012f87c

0:000> ? dwo(myInt+4)
Evaluate expression: 2 = 00000002

// C++ syntax

0:000> ?? (&myInt)
int * 0x0012f878

0:000> ?? myInt
int [2] 0x0012f878 1

0:000> ?? (&myInt+1)
int * 0x0012f87c

0:000> ?? *(myInt+1)
int 2
```

Default Expression Evaluator

- The following always use the C++ expression evaluator:
  - ?? command (evaluate C++ expression)
  - the watch window
  - the locals window

- All other commands and debugging information windows use the default expression evaluator

- You can use the .expr command to change the default evaluator
  - .expr → show current evaluator
  - .expr /q → show available evaluators
  - .expr /s c++ → set c++ as the default expression evaluator
  - .expr /s masm → set masm as the default expression evaluator
Mixing Both Evaluators “on-the-fly”

- You can use both expression evaluators within one command

- For mixing both modes: @@(...)  
  - If any portion of an expression is enclosed in parentheses and prefixed by a double @@, it will be evaluated by the opposite of the current expression evaluator  
  - this way you can use two different evaluators for different parameters of a single command  
  - It is possible to nest these symbols; each appearance of this symbol switches to the other expression evaluator

- Explicitly specify an expression evaluator
  => @@c++(...)  
  => @@masm(....)

Example – Mixed Expression

```
// The following command will set the default expression evaluator to
// MASM, and then evaluate Expression1 and Expression3 as MASM
// expressions, while evaluating Expression2 as a C++ expression:
// -------------------------------------------------------------------
0:000> .expr /s masm
0:000> ? Expression1 + @@( Expression2) + Expression3
0:000> ? `myFile.cpp:118` // get address of line 118 in myFile.cpp
Evaluate expression: 4570359 = 0045bcf7
// -------------------------------------------------------------------
// source-line expressions cannot be used in C++ expressions
// let’s nest a MASM expression within a C++ expression
// => store address of line 43 of "myFile.cpp" into nLocalVar
// -------------------------------------------------------------------
0:000> ?? nLocalVar = @@("myFile.cpp:43")
int 4570359
```
## Aliases in WinDbg

- Strings that are automatically replaced with other character strings
- Consist of: alias name + alias equivalent

### 1) User-named aliases
- Set and named by the user (both are case-sensitive)
- Manipulate by: `as` or `aS` (Set Alias), `ad` (Delete Alias), `al` (List Aliases)

### 2) Fixed-name aliases
- Set by the user, named $u0, $u1, .. $u9
- Set by the `r` (register) command + . (dot) before the “u”
  
  Example: `r $.u0 = "dd esp+8; g"`

### 3) Automatic aliases
- Set and named by the debugger
- Are similar to automatic pseudo registers, except that they can be used with alias-related tokens such as `${ .. }` (pseudo-registers cannot)
- Examples: `$ntsym`, `$CurrentDumpFile`, `$CurrentDumpPath`, ...

---

## User-Named and Fixed-Name Aliases

### 1) User-named aliases
- By default a user-named alias must be separated from other characters. The first and last character of an alias name must either:
  - begin/end the line or
  - be preceded/followed by a space, semicolon, or quotation mark
- If a user-named alias is touching other text, it must be enclosed in `{ }` (Alias interpreter)
- Can be used in the definition of a fixed-name alias
  - To use a user-named alias in the definition of another user-named alias, you need to prefix the `as` or `aS` command with a semicolon (else no alias replacement will occur on that line). Explanation: Any text entered into a line that begins with `as`, `aS`, `ad`, or `al` will not receive alias replacement. If you need aliases replaced in a line that begins with these characters, prefix it with a semicolon.
  - Are easier to use than fixed-name aliases
    - Their definition syntax is simpler
    - they can be listed using the `al` (List Aliases) command

### 2) Fixed-named aliases
- Are automatically replaced if they are used adjacent to other text
- Can be used in the definition of any alias
### Commands for User-Named Aliases

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>as Name Equivalent</td>
<td>Set alias</td>
</tr>
<tr>
<td>as /ma Name Address</td>
<td>Set alias to the NULL-terminated ASCII string at Address</td>
</tr>
<tr>
<td>as /mu Name Address</td>
<td>Set alias to the NULL-terminated Unicode string at Address</td>
</tr>
<tr>
<td>ad Name</td>
<td>Delete alias with Name</td>
</tr>
<tr>
<td>ad *</td>
<td>Delete all aliases</td>
</tr>
<tr>
<td>al</td>
<td>List user-named aliases</td>
</tr>
<tr>
<td>${Alias}</td>
<td>${Alias} is replaced by the alias equivalent, even if it is touching other text. If the alias is not defined, the ${Alias} is not replaced</td>
</tr>
<tr>
<td>${/f:Alias}</td>
<td>Same as above except that ${/f:Alias} is replaced with an empty string if the alias is not defined</td>
</tr>
<tr>
<td>${/n:Alias}</td>
<td>Evaluates to the alias name</td>
</tr>
<tr>
<td>${/d:Alias}</td>
<td>Evaluates: 1 = alias defined; 0 = alias not defined</td>
</tr>
</tbody>
</table>

### Example - Aliases

```
0:001> as Short kernel32!CreateRemoteThread // user-named alias
0:001> uf Short
0:001> r $u0 = kernel32!CreateRemoteThread // fixed-name alias
0:001> uf $u0
0:001> as DoInc r eax = eax+1; r ebx = ebx+1 // alias used as a macro for commands
0:001> DoInc
0:001> DoInc
// aliases are replaced as soon as they are used
//
0:001> r $u2 = 2
0:001> r $u1 = 1+$u2
0:001> r $u2 = 6
0:001> ? $u1
Evaluate expression: 3 = 00000003
0:001> as two 2
0:001> r $u1 = 1+ two // notice the empty space before two!
0:001> as two 6
0:001> ? $u1
Evaluate expression: 3 = 00000003
// using a named alias within another named alias
//
0:001> as two 2
0:001> as xy1 two + 1 // xy1 = two + 1
0:001> as xy2 two + 1 // xy2 = 2 + 1 (you must prefix as with a semicolon for a replacement to occur)
```
Debugger Command Programs

- Consist of
  - debugger commands
  - control flow tokens (.if, .for, .while, ..)

- Variables
  - Use user-named aliases or fixed-name aliases as “local variables”
  - Use pseudo-registers ($t0, ..) for numeric or typed variables

- For comments use $$ [any text]

- A pair of braces {} is used to surround a block of statements
  - When each block is entered all aliases within a block are evaluated
  - There must be a control flow token before the opening brace
  - To create a block solely to evaluate aliases use the .block { .. }
  - Use $(Alias) (alias interpreter) for user-named aliases that touch other text

Control Flow Tokens

- Used to create execution loops and for conditional execution
- Each condition must be an expression (commands are not permitted)

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.block</td>
<td>Performs no action. It is used solely to introduce a block. Note that you cannot simply use {} to create a block.</td>
</tr>
<tr>
<td>.if, .else, .elseif</td>
<td>Like the if, else or else if keyword in C</td>
</tr>
<tr>
<td>.for, .while, .Break, .continue</td>
<td>Like the for, while, break or continue keyword in C</td>
</tr>
<tr>
<td>.foreach</td>
<td>Parses the output of debugger commands, a string or a text file. It then takes each item it finds and uses it as the input to a specified list of debugger commands.</td>
</tr>
</tbody>
</table>
Command Programs Execution

There are several possible ways to execute a program:

- Enter all statements into the debugger window as a single string (commands separated by semicolons)

- Store all statements into a script file and use $$><$$ to run the file.
  $$><$$ (Run Script File):
  - opens the specified file
  - replaces all carriage returns with semicolons
  - executes the resulting text as a single command block

Example – Debugger Command Program

```plaintext
$$ ----------------------------------------------------------
$$ From WinDbg's help: "Debugger Command Program Examples"
$$ You will find the full explanation there.
$$ ----------------------------------------------------------

$$ Get module list LIST_ENTRY in $t0.
  r? $t0 = &@peb->Ldr->InLoadOrderModuleList

$$ Iterate over all modules in list.
  .for (r? $t1 = *(ntdll!_LDR_DATA_TABLE_ENTRY**)@$t0;
     ($t1 != 0) & ($t1 != @$t0);
  r? $t1 = (ntdll!_LDR_DATA_TABLE_ENTRY*)@$t1->InLoadOrderLinks.Flink)
  {
    $$ Get base address in $Base.
    as /x ${/v:$Base} @@c++(@$t1->DllBase)
    $$ Get full name into $Mod.
    as /msu ${/v:$Mod} @@c++(&@$t1->FullDllName)
    .block
    {   .echo ${$Mod} at ${$Base}
    }  
ad ${/v:$Base}
ad ${/v:$Mod}
```
## Useful Breakpoint Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bl</td>
<td>Breakpoint list</td>
</tr>
<tr>
<td>bp</td>
<td>Set Breakpoint</td>
</tr>
<tr>
<td>bu</td>
<td>Set Unresolved Breakpoint: defers the actual setting of the breakpoint until the module is loaded</td>
</tr>
<tr>
<td>ba</td>
<td>Break on Access</td>
</tr>
<tr>
<td>bc</td>
<td>Breakpoint Clear</td>
</tr>
<tr>
<td>be, bd</td>
<td>Breakpoint Enable, Disable</td>
</tr>
</tbody>
</table>

### Example – Setting Simple Breakpoints

```
0:000> bu kernel32!LoadLibraryExW
0:000> bu kernel32!CreateProcessW
0:000> bu kernel32!CreateThread
0:000> ba r4 0012fe34 // break on access (read or write); monitor 4 bytes
0:000> ba w2 0012fe38 // break on access (write); monitor 2 bytes
0:000> bl
0 d 7c801af1 0001 (0001) 0:**** kernel32!LoadLibraryExW
1 d 7c802332 0001 (0001) 0:**** kernel32!CreateProcessW
2 d 7c810637 0001 (0001) 0:**** kernel32!CreateThread
3 d 0012fe34 r 4 0001 (0001) 0:****
4 d 0012fe38 2 2 0001 (0001) 0:****

0:000> bd 0,2 // disable breakpoints 0 and 2
0:000> bc 4 // clear breakpoint 4

0:000> bl
0 d 7c801af1 0001 (0001) 0:**** kernel32!LoadLibraryExW
1 d 7c802332 0001 (0001) 0:**** kernel32!CreateProcessW
2 d 7c810637 0001 (0001) 0:**** kernel32!CreateThread
3 d 0012fe38 r 4 0001 (0001) 0:****
```
Example – More Complex Breakpoints

- Break at specified source code line
  0:000> bp `mod!source.c:12`

- Breakpoint that will starts hitting after 5 passes
  0:000> bu kernel32!LoadLibraryExW 5
  0:001> bl // after 3 passes (0002=remaining count)
  0 e 7c801af1 0002 (0005) 0:**** kernel32!LoadLibraryExW

- Break only if called from thread ~1
  0:000> ~1 bu kernel32!LoadLibraryExW
  0:001> bl
  0 e 7c801af1 0001 (0001) 0:**** kernel32!LoadLibraryExW

- Break at all symbols with pattern myFunc*
  0:000> bp mod!myFunc*
  - SymbolPattern is equivalent to using x SymbolPattern

- Break on member methods
  0:000> bp @@c++( MyClass::MyMethod )
  - Useful if the same method is overloaded and thus present on several addresses

Example – Breakpoints With Commands

- Skip execution of WinMain
  0:000> bu MyApp!WinMain "reip = poi(@esp); r esp = @esp + 0x14; .echo WinSpy!WinMain entered; gc"
  - Right at a function’s entry point the value found on the top of the stack contains the return address
    - r eip = poi(@esp)  Set EIP (instruction pointer) to the value found at offset 0x0
    - WinMain has 4x4 byte parameters = 0x10 bytes + 4 bytes for the return address = 0x14
      - r esp = @esp + 0x14  Add 0x14 to ESP, effectively unwinding the stack pointer

- Break only if LoadLibrary is called for MyDLL
  0:000> bu kernel32!LoadLibraryExW ";as /mu ${/v:MyAlias}.poi(@esp+4); .if ( $spat("${MyAlias}","*MYDLL\*") != 0 ) { kn; } .else { gc }"
  - The first parameter to LoadLibrary (at address ESP + 4) is a string pointer to the DLL name in question.
  - The MASM $spat operator will compare this pointer to a predefined string-wildcard, this is "*MYDLL*" in our example.
  - Unfortunately $spat can accept aliases or constants, but no memory pointers. This is why we store our pointer in question to an alias (MyAlias) first.
  - Our kernel32!LoadLibraryExW breakpoint will hit only if the pattern compared by $spat matches. Otherwise the application will continue executing.
## Exception Analysis Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.lastevent</td>
<td>first-change or second-change?</td>
</tr>
<tr>
<td>!analyze -v</td>
<td>Displays detailed information about the current exception</td>
</tr>
<tr>
<td>.exr -1</td>
<td>Display most recent exception</td>
</tr>
<tr>
<td>.exr Addr</td>
<td>Display exception at Addr</td>
</tr>
<tr>
<td>!cppexr</td>
<td>Display c++ exception at address 7c901230</td>
</tr>
<tr>
<td>g, gH</td>
<td>Go with Exception Handled</td>
</tr>
<tr>
<td>gN</td>
<td>Go with Exception Not Handled</td>
</tr>
</tbody>
</table>

## Example - Exceptions

```
0:000> .lastevent
Last event: dac.154c: Stack overflow - code c00000fd (first chance)
   debugger time: Wed Aug 29 16:04:15.367 2007 (GMT+2)

0:000> .exr -1
ExceptionAddress: 00413fb7 (TestApp!_chkstk+0x00000027)
   ExceptionCode: c00000fd (Stack overflow)
   ExceptionFlags: 00000000
   NumberParameters: 2
   Parameter[0]: 00000000
   Parameter[1]: 001e2000

0:000> !analyze -v
FAULTING_IP:
TestApp!_chkstk+27 [F:\SP\vctools\crt_bld\SELF_X86\crt\arc\intel\chkstk.asm @ 99]
00413fb7 8500 test    dword ptr [eax],eax
...```
Remote Debugging with WinDbg

- Target computer (server)
  - Copy dbgsrv.exe, dbgeng.dll and dbghelp.dll to the remote computer
  - Disable the firewall for "dbgsrv.exe"
  - Run → dbgsrv.exe -t tcp:port=1025

Windows Vista: Start dbgsrv.exe with admin privileges to see all processes.

- Host computer (client)
  - Run → WinDbg.exe -premote tcp:server=TargetIP_or_Name,port=1025
  - File (Menu) → Attach to Process → Select Process on Target Computer that you would like to debug

WinDbg Commands for Remote Debugging

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cdb.exe –QR server(IP or Name)</td>
<td>Lists all debugging servers running on the specified network server.</td>
</tr>
<tr>
<td>.detach</td>
<td>Detach from Process</td>
</tr>
<tr>
<td>.endpsrv</td>
<td>End dbgsrv.exe on remote computer. This command will kill the debugged process if you don’t detach first.</td>
</tr>
<tr>
<td>.tlist</td>
<td>lists all processes running on the (remote) system</td>
</tr>
</tbody>
</table>
Monitoring Events

- The debugger engine provides facilities for monitoring and responding to events in the target application.

- Events are generally divided into:
  - Exception events
    - Breakpoint, Access Violation, Stack Overflow, division-by-zero, etc.
    - For a full list see: Specific Exceptions.
  - Non-exception events
    - Create Process, Create Thread, Load Module, Unload Module.
    - For a full list see DEBUG_FILTER_XXX.

- Whenever a debugging session is accessible, there is a last event
  - Command: .lastevent

Events Filters in WinDbg

- Provide simple event filtering
- Influence how the debugger engine proceeds after an event occurs in a target
- To list all events: sx

- Break or execution status:
  - Influences whether the debugger will break into the target
    - First-chance break on event (sxe)
    - Second-chance break event (sxd)
    - Debugger message output on event (sxn)
    - Ignore event (sxi)

- Handling or Continue status:
  - Determines whether an exception event should be considered handled (gH) or not-handled (gN) in the target
# Events Filters Dialog

**Execution:**
- **Enabled**: first-chance break (sxe)
- **Disabled**: second-chance break (sxd)
- **Output**: message output on event (sxn)
- **Ignore**: ignore event (sxi)

**Continue:**
- **Handled**: Consider event handled when execution resumes
- **Not-Handled**: Consider event not-handled when execution resumes

## Event Arguments

- Some filters take arguments that restrict which events they match
- No arguments → No restriction

<table>
<thead>
<tr>
<th>Event</th>
<th>Match criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create Process</td>
<td>The name of the created process must match the argument.</td>
</tr>
<tr>
<td>Exit Process</td>
<td>The name of the exited process must match the argument.</td>
</tr>
<tr>
<td>Load Module</td>
<td>The name of the loaded module must match the argument.</td>
</tr>
<tr>
<td>Target Output</td>
<td>The debug output from the target must match the argument.</td>
</tr>
<tr>
<td>Unload Module</td>
<td>The base address of the unloaded module must be the same as the argument.</td>
</tr>
</tbody>
</table>
Behind the Scenes

Using WinDbg

Global Flags

- Application Verifier
- Process Dumps

Flags? GFlags? Global Flags!

- GFlags enables and disables features by editing the Windows registry
- GFlags can set system-wide or image-specific settings
- Image specific settings are stored in:
  - HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\ImageFileName\GlobalFlag
- The OS reads these settings and adopts its functionality accordingly
- GFlags can be run from the command line or by using a dialog box
- We can also use !gflags in WinDbg to set or display the global flags
- With GFlags we can enable:
  - heap checking
  - heap tagging
  - Loader snaps
  - Debugger for an Image (automatically attached each time an Image is started)
  - Application verifier
  - Etc.
GFlags Dialog

- System Registry: System-wide settings that affect all processes running on Windows. They remain effective until you change them. Restart Windows to make the changes effective.
- Kernel Flags: Run-time settings that affect the entire system. They take effect immediately without rebooting, but they are lost if you shut down or restart the system.
- Image File: They affect instances of the specified program that start after the command completes. They are saved in the registry and remain effective until you change them.

GFlags: “Show loader snaps” Enabled

WinDbg Output:

LDR: LdrLoadDll, loading samlib.dll from C:\WINDOWS\system32\C:\WINDOWS\system32\samlib.dll
LDR: Loading (DYNAMIC, NON_REDIRECTED) C:\WINDOWS\system32\samlib.dll
ModLoad: 71bf0000 71c03000 C:\WINDOWS\system32\samlib.dll
LDR: samlib.dll bound to ntdll.dll
LDR: samlib.dll bound to ADVAPI32.dll
LDR: samlib.dll has correct binding to ADVAPI32.dll
LDR: samlib.dll has correct binding to RPCRT4.dll
LDR: samlib.dll bound to RPCRT4.dll
LDR: samlib.dll has correct binding to RPCRT4.dll
LDR: samlib.dll bound to KERNEL32.dll
LDR: samlib.dll has stale binding to KERNEL32.dll
LDR: samlib.dll bound to ntdll.dll via forwarder(s) from kernel32.dll
LDR: samlib.dll has correct binding to ntdll.dll
LDR: samlib.dll has stale binding to ntdll.dll
LDR: samlib.dll has correct binding to ntdll.dll
LDR: Stale Bind KERNEL32.dll from samlib.dll
LDR: samlib.dll has correct binding to KERNEL32.dll
LDR: LdrGetProcAddress by NAME - RtlAllocateHeap
LDR: LdrGetProcAddress by NAME - RtlFreeHeap
LDR: LdrGetProcAddress by NAME - RtlGetLastWin32Error
LDR: LdrGetProcAddress by NAME - RtlIsAllocateHeap
LDR: samlib.dll bound to USER32.dll
LDR: samlib.dll has correct binding to USER32.dll
LDR: samlib.dll bound to USER32.dll
LDR: Stale Bind USER32.dll from samlib.dll
[d58,690] LDR: Real INIT LIST for process C:\Development\Sources\TestApp\Release\TestApp.exe pid 3416 0xd58
[d58,690] C:\WINDOWS\system32\samlib.dll init routine 003A0F30
[d58,690] LDR: samlib.dll loaded - Calling init routine at 003A0F30
Get Even More: Enable Application Verifier

- Application Verifier:
  - is a runtime verification tool for Windows applications
  - is monitoring an application's interaction with the OS
  - profiles and tracks:
    - Microsoft Win32 APIs (heap, handles, locks, threads, DLL load/unload, and more)
    - Exceptions
    - Kernel objects
    - Registry
    - File system
  - with `!avrf` we get access to this tracking information

Note: Under the hood Application Verifier injects a number of DLLs (verifier.dll, vrfcore.dll, vfbasics.dll, vfcompat.dll, and more) into the target application. More precisely: It sets a registry key according to the selected tests for the image in question. The windows loader reads this registry key and loads the specified DLLs into the applications address space while starting it.
Application Verifier Variants

GFlags Application Verifier
- Only verifier.dll is injected into the target process
- verifier.dll is installed with Windows XP
- Offers a very limited subset of Application Verifier options
- Probably this option in GFlags is obsolete and will eventually be removed (?)

Application Verifier
- Can freely be downloaded and installed from the MS website
- Additionally installs vrfcore.dll, vfbasics.dll, vfcompat.dll, and more into Windows\System32
- Enables much more test options and full functionality of the !avrf extension

Application Verifier Symbols

Application Verifier is installed with PDB’s with full symbol information
- Note the source information included in the disassembly
- These are the only modules from Microsoft that I’ve seen delivered with full symbol information
- In fact, WinDbg must use these symbols rather than the public ones from the server. Otherwise the !avrf extension will not work

```
.reload /f "C:\Windows\System32\verifier.pdb"
```
### Common !avrf Parameters

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!avrf</td>
<td>Displays current Application Verifier options. If an Application Verifier Stop has occurred, reveal the nature of the stop and what caused it.</td>
</tr>
<tr>
<td>!avrf -cs</td>
<td>Displays the critical section delete* log. * DeleteCriticalSection API. ~CCriticalSection calls this implicitly.</td>
</tr>
<tr>
<td>!avrf -hp 5</td>
<td>Displays the heap operation log (last 5 entries). * HeapAlloc, HeapFree, new, delete</td>
</tr>
<tr>
<td>!avrf -dlls</td>
<td>Displays the DLL load/unload log.</td>
</tr>
<tr>
<td>!avrf -ex</td>
<td>Displays the exception log.</td>
</tr>
<tr>
<td>!avrf -cnt</td>
<td>Displays a list of global counters (WaitForSingleObject calls, CreateEvent calls, HeapAllocation calls, ..).</td>
</tr>
<tr>
<td>!avrf -threads</td>
<td>Displays information about threads in the target process. For child threads, the stack size and the CreateThread flags specified by the parent are displayed as well.</td>
</tr>
<tr>
<td>!avrf -trm</td>
<td>Displays a log of all terminated* and suspended threads. * TerminateThread API</td>
</tr>
</tbody>
</table>

### Example – !avrf

```// Right after our application executes:
// HeapAlloc: 0x00140000, 8, dwBytes =0x0A00000 | --> 0x033D1000;
```

```
O:000> !avrf -hp 1
Verifier package version >= 3.00
Dumping last 1 entries from tracker @ 01690fd8 with 1291 valid entries ...
-----------------------------------------------------------------
HeapAlloc: 33D1000 A00000 0 0
  004019cf: TestApp\CMyDialog:OnBtnClicked_HepAlloc+0x4F
  0041a0c1: TestApp\IAfxDispatchCmdMsg+0x3D
  0041a2a6: TestApp\CCmdTarget:OnCmdMsg+0x10A
  0041a76c: TestApp\CDialog:OnCmdMsg+0x1B
  0041d05c: TestApp\CWnd::OnCommand+0x51
  0041d92b: TestApp\CWnd::OnCmdMsg+0x2F
  0041b2eb: TestApp\CWnd::WindowProc+0x22
...
```

```
O:000> !avrf -threads
```

```
Thread ID = 0xNE4
Parent thread ID = 0x3E3C
Start address = 0x00447d82: TestApp\IL7+11647(\ThreadProc\KVKVAEX)
Parameter = 0x0061833c

Thread ID = 0x3E3C
Initial thread

Number of threads displayed: 0x2
```
Process Dumps

- A Process’s dump
  - is quite similar to a non-invasive attach
  - represents a snapshot of a process at a given time
  - varies in size, depending on what contents and information it includes

- With a dump
  - we can examine memory as well as other internal structures of a process
  - we cannot set breakpoints or step through the program execution

- Dump a dump
  - we can always "shrink" a dump with more information to a dump with less information
  - use the .dump command as you would with a live process
Types of Dumps

1) Kernel-mode dumps
   Variants: Complete Memory Dump, Kernel Memory Dump, Small Memory Dump

2) Full User-mode dumps
   - Generated by WinDbg with ".dump /f" or by Dr. Watson on Windows 2000
   - Includes the entire memory space of a process, the program's executable image itself, the handle table
   - Widely used in the past, MS is slowly dropping support for it

3) Minidumps
   - .dump /m??
   - The modern dump format
   - Fine-grained control about what is included in the dump (see MSDN: MINIDUMP_TYPE)
   - Despite their names, the largest minidump file actually contains more information than a full user-mode dump. For example, .dump /mf or .dump /ma creates a larger and more complete file than ".dump /f"

Determine Type of a Dump

You can load the dump in question into WinDbg. WinDbg will call a minidump a "User Mini Dump File," and the old style crash dump will be called a "User Dump File."
Example – “.dump” Command

0:000> .dump /ma d:\large.dmp
Creating d:\large.dmp - mini user dump
Dump successfully written

0:000> .dump /m d:\small.dmp
Creating d:\small.dmp - mini user dump
Dump successfully written

* all possible data: full memory, code sections, PEB and TEB’s, handle data, thread time information, unloaded module lists, and more

* only basic information: module information (signatures), thread and stack information

Choosing the Best Tool

<table>
<thead>
<tr>
<th>Scenario / Options</th>
<th>ADPlus</th>
<th>Dr. Watson</th>
<th>CDB and WinDbg</th>
<th>UserDump</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application crash (postmortem debugging)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Application &quot;hang&quot; (stops responding but does not actually crash)</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Application encounters an exception</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Application is running normally</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Application that fails during startup (i.e. missing DLL dependency)</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Shrinking an existing dump file</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Dump all running applications with the same image name at once</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Control what information is included in the dump file</td>
<td>No</td>
<td>No^1</td>
<td>Yes</td>
<td>No^2</td>
</tr>
</tbody>
</table>

1: Always creates a small Minidump –MiniDumpNormal – with basic information only. It is usually less than 20KB in size.
2: Always creates a Minidump with Full Memory information. It is usually 20-200MB in size.
Your Homework

- Read WinDbg's documentation
  - Memory leaks, handles, deadlocks, breakpoints with conditions, and more. Everything is explained there.

- Learn assembly
  - It will greatly improve your debugging skills
  - Besides WinDbg assembly will be your best friend when it comes to debugging situations

Questions? Suggestions?

- You have a question about WinDbg?
- You are interested in a WinDbg lab or seminar?
- You think that something in "WinDbg. From A to Z!" could be improved?
- Or you would just like to say WOW, this presentation was really useful?

- Feel free to drop a line at: mailrkuster@windbg.info
  The actual email address does not contain the word "mail" – spam prevention.
References

• WinDbg’s Documentation, MSDN

• Common WinDbg Commands (Thematically Grouped)
  http://software.rkuster.com/windbg/printcmd.htm

• Matching Debug Information
  http://www.debuginfo.com/articles/debuginfomatch.html

• Generating Debug Information with Visual C++
  http://www.debuginfo.com/articles/gendebuginfo.html

• Microsoft Windows Internals, Fourth Edition

• Advanced Kernel Debugging
  Andre Vachon, PowerPoint, WinHec 2004

• Application Verifier’s Documentation