WinDbg – Cheat Sheet Version 1.23

Basic / Environnment

Types used in commands	b = byte w = word d = double word q = quad word f = float D = double	s = symbol a = Ascii u = Unicode za = 0 term. Ascii zu = 0 term. Unicode
.hh <command> !<extension>.help	Getting help Help for an extension.	
Double click left, click right click right	Select, copy to clipboard, paste	
.formats <value>	Shows the value in different formats.	
??<expression> ?	C++ expression evaluator Evaluate expression. (i.e. convert numbers)	??&myVar ??0xffff & ~7 ?0n15
!teb	Shows thread environment block	Contains the stack address and limit.
!peb	Shows process environment block. i.e. command line.	Same as dt _PEB @$peb
.tlist	Lists all processes.	
!error <code>	Displays error information.	HRESULT or Windows error code
.effmach x86 !wow64exts.sw	Change processor mode used by debugger to see the 32 bit stack. Switches 32/64 bit mode.	If you debug a 32 bit process running in WOW64 with WinDbg64 .load wow64exts
vertarget	Version of target computer.	Includes OS, uptime, processors, etc.
.cmdtree <file>		
l-t l+t	Disable source mode. Enable source mode.	More source code options available.
.prefer_dml [1|0]	Enables DML for supported commands.	lm, k etc.
@!"..."	Treat whole string as symbol.	bp @!"foo<>"

Symbol management

.symfix[+] .sympath .sympath+ <path>	Sets / adds path to MS symbol server. Shows current symbol path. Adds another path to the symbol path.
!chksym <module>	Shows PDB file that matches module
!sym noisy	Helps resolving symbol loading issues.
.reload <module+> .reaload /unl <module>	Loads symbols (lazy) /f = force /i = ignore pdb mismatch /v = verbose Reloads the unloaded DLL. Useful to decode incomplete call stacks.
ld *	Forces loading all all symbols. Bypasses the lazy loading default behaviour.
.symfix x:\symbols .reload /f ld *	Loads all symbols and stores it in the given folder for later usage.
Symbolpath syntax. Separate entries by semicolon. SRV*<folder>*http://msdl.microsoft.com/download/symbols or All following path entries are cached in <folder>. **cache*<folder>**;srv*http://msdl.microsoft.com/download/symbols Used by debuggers: _NT_SYMBOL_PATH (Slow debugging!)	

Connect / attach

.attach <processId> .detach	Attaches / detaches from running process	Use .tlist to get list of processes.
q qd qq	Quit debug session. Detach only. Also quits remote server (stub).	Terminates target Behaves like q in normal mode.
.restart		
.childdbg 1	Attach automatically to child processes.	Same as -o command line option.
dbgsrv.exe -t tcp:port=<port> [-cs <exe> <args>]	Starts the **process server (stub)** using tcp.	Debugger on client side is called smart. Symbol on client.
tcp:server=<ip>,port=<port>	Connecting to remote stub (not session).	In WinDbg
WinDbg -server tcp:port=<port> CDB -server tcp:port=<port>	Starts debugger as **debug server**. client. You can use WinDbg as stupid client. Connect to remote session (not stub). Symbols on server.	To debug services in session 0 see Inside Windows Debugging chapter 7, page 231ff.
Windbg -c "..." [application]	Executes the specified commands at the initial breakpoint.	Windbg -c "$$><script.txt" notpad.exe
Note: You can enter a debugger in gflags image tab. It gets concatenated with the initial command line. Use devenv /debugexe for Visual Studio. Keep in mind: By attaching by IFEO the debugger runs in the security context of the target.		

Modules / Extensions

.load <path>	Loads a module	
.loadby <module> <loaded module> sxe ld <module> sxe ud <module> Or IFEO: BreakOnDLLLoad	Loads module from same path as another loaded module. Stop when <module> is loaded / unloaded. sxe ld mscorlib sxe ld * sxd ld to disable.	.loadby sos mscorwks .loadby sos clr Let the debugger load the sos dll automatically: sxe -c ".loadby sos mscorwks; g" ld mscorwks
.chain	Show loaded debugger extensions.	
lm lmv lm m <pattern> lmo lm1m	List loaded modules Verbose Filter by a pattern Only loaded modules Only the module names. (.foreach)	lmv m myap* lmv m mscorwks lmv m clr Module name must NOT(!) contain the extension. (I.e .dll)
!dh <base address>	Dump PE header	Base address is obtained via lm.
!dlls !dlls -c <base addr.>	Lists loaded modules.	Includes the load counter. lm

Execution

g [address] gu gc gn	Go [...to address] go up go from conditional breakpoint in same fashion it was hit. Continue without handling the exception.	i.e. skip over prologue EAX / RAX is set i.e. Ctrl-C
p	Step over. Instruction or source line depending on current mode /source availability.	sub routines are handled as single step.
pt pc	Step to next return. Step to next call	But stays inside the function. (Unlike gu)
t	Step into. Instruction or source line.	Sub routines are entered.
wt wt -l <depth> wt -i <module> wt -m <module> -ns	Trace and watch. Executes a function and displays statistics. Ignores code from module. Restrict tracing to module no summary	Must be executed at the beginning of a function wt -i USER32 -i kernel32 -i msvcrt -i msxml4 -i ntdll *Filtering does not work in version 6.12.0002.633.*

Exceptions

sxe <event code> sxd <event code> sxn <event code> sx sxr	Enable – stop on first chance Disable – stop on second chance Notify only Show summary. Reset all to default.	Stop on first chance clr exceptions: sxe clr Stop when mscorwks is loaded: sxe ld mscorwks
!pe	Print exception.	
.exr <address> -exr -1	Show exception record. Show most recent exception.	Parameter[1] contains the thrown object. If code is CLR then !pe is valid.
.cxr <context>	Displays and sets register context. Register context is second member of EXCEPTION_POINTERS.	Used in kernel32! UnhandledExceptionFilter(EXCEPTION_POINTERS) to find the causing stack.
!soe -create[2] <type> <pseudoreg nr.> !soe -derived <type> <pseudoreg nr.>	Stop on exception -derived catches the exception plus the derived ones.	!soe -create System.ArgumentNullException 1

Analyzing problems

!analyze -v !analyze -v -hang	Exception analysis Analyzes blocked threads.
.lastevent	Shows the last event like an exception.
!gle !gle -all	Display last error for current stack For all stacks.
!avrf	Shows enabled application verifier options and hints for the just found problem.

Dumps

.dump -ma <file>	Mini dump with all options.

Logging

.logopen <file> .logclose	Writes everything on the debugger console to the specified file.	

Threads

~ ~<logical thread ID>s ~* e [command] ~ <thread>f ~ <thread>u ~# |	List all threads Set thread active. For all threads execute the command. i.e. k Freeze thread Unfreeze thread Thread that caused current event. Lists debugged processes.	You can use also the thread ID: ~~[96A]s The e is necessary to execute extension commands starting with dot or bang. ~* e !clrstack. Keep in mind that the logical thread id may change between subsequent dumps. In a dump lists the dumped process.
!runaway	Shows user mode time for all threads. (Which consumes most?)	
!threads !threads -special	List managed threads. Lists special threads.	Includes count of locks. i.e. finalizer thread etc.
!threadpool	Infos about the thread pool usage.	

Stacks

k kp (private symbols) kP kb kf k = <base pointer> [<stack pointer> <instruction pointer>]	Basic With parameters With parameters formatted. (Newlines) First three parameters. Shows consumed stack memory. Reconstruct corrupt x86 callstack. Base pointer comes from try and error on stack dump. See !teb	Showing possibly all frames: k 1000 n shows additionally the frame numbers: kpn Base pointer alone may lead to wrong result if FPO frames are present.
.frame <frame> .frame /r <frame>	Set local context to frame. Shows the available registers at the frame.	See kn .frame 0n10;dv
!uniqstack	Shows stacks for all threads excluding duplicates.	!uniqstack -pn shows parameter information and frame numbers.
!findstack <module>	Finds threads which contains calls to module.	
!clrstack !clrstack -a	Shows managed only stack. With parameters and local variables. Only in debug build reliable.	Alternative: !dumpstack -EE Not usable for EBP chain.
!dumpstack	Show managed and unmanaged stack. Includes the method descriptor. X86 output: <ebp> <ret> X is calling Y	Not a real stack trace. Just looking for symbols on the stack. Shows child EBP! Important: <ebp> is the new EBP for Y <ret> is the return address to X

Breakpoints

bp <address>	Set breakpoint Works also with jitted code address! Use return address from call stack to complete methods.	bp ntdll!RtlAllocateHeap bp /1 mod!func creates a one shot breakpoint.
bm <pattern>	Set breakpoint by pattern.	bm module!Foo::bar gets all overloads.
ba <access><width> <address>	Set break on access. Access : r, w, e	ba w4 gGlobal
bu <symbolic ref.>	Set unresolved breakpoint.	
bl	List all breakpoints	Including the managed ones.
bc <breakpoint>	Clear breakpoint	bc *; bc 2
be <breakpoint> bd <breakpoint>	Enable breakpoint Disable breakpoint	be * bd *
!bpmd -md <method descr. address> !bpmd <module> <name>	Set breakpoint regardless if method is jitte dor not.	Breakpoint is pending until method is jitted. Method name must be fully qualified. Module name must contain the extension. (i.e. module.dll!...)

Inspect Memory / Objects / Symbols

dd <address> [Lxxx] dq <address> [Lxxx] da <address> du <address> db <address> [Lxxx] df <address> dD [Lxxx]	Dump double word Dump quad word Dump Ascii Dump Unicode Dump Byte+Ascii Dump float Dump Double	xxx is the number of objects to dump da @esp L100 finds strings on the stack.
dps <address> [Lxxx] dpa <address> [Lxxx] dpu <address> [Lxxx] dpp <address> [Lxxx] s = symbol a = Ascii u = Unicode p = pointer	Dumps pointer sized memory in the given range. dps interprets and presents the memory as symbols. The other version interprets the memory as pointer, dereference it and present the resulting location in different formats. Alternatively you can specify start and end address.	Examples: dps <address> shows stack trace database. dpu @esp scans stack for unicode strings allocated on heap. dpp @esp shows pointers referenced on stack. Finds pointer to objects that have a vtable.
Tip: Scanning a range for strings	d* <address> enter d* enter enter.	
dv dv /V /t /i	Display local variables. /V displays (virtual) addresses or register location. /i shows if local is parameter or function argument. /t shows the type.	!for_each_frame dv shows the local variables for all stack frames. (Private symbols provided!)
dt <type> [<address>] Options: -b = recursive -v = includes size dt OXIDEntry <addr>	Display structures contained in PDB files. Without address it just prints the data type layout. Figure out target of call to STA.	dt _PEB @$peb dt this OXIDEntry is first argument of GetToSTA. (see kp)
ln <address>	Shows nearest symbols that match the address.	Tip If you have an object address you can try to apply on its vtable to figure out its type.
x <module!symbol> Options: /t = Show data types if possible	Examine symbol. Allows use of wildcards.	x module!Foo::bar x module!* x /t module!* The module must be present needed!
by (<address>) wo (<address>) dwo (<address>) qwo (<address>)	Dereference and get the value.	(low) byte / word / dword / quad word from address.
poi(<address>)	Pointer sized data from address. (Dereference and get the pointer sized value)	du poi(@esp+4) The string pointer resides on stack, the string itself is on the heap.
!do <address>	Dump managed (reference) object.	Includes the owning thread! Shows members of derived exceptions.
!dso	Dump managed stack objects.	Including exceptions.
!dumpvc <mt> <address>	Shows fields of value type.	Method table address comes form field information dumped via !do
!da <array>	Dump array	

Edit memory

e<type> <address> [values] Type: [b | w | d | q | a | u | za | zu | f | D] Values are separated by space. f <start> <end> <value> Start and end address are inclusive.	*Examples:* eb <address> 90 90 enters two NOP instruction. eb <address> 'h' 'i Let ntll break if a certain Win32 error code happens: ed ntdll! g_dwLastErrorToBreakOn 5 Fill memory. Value is a byte. I.e. 0x90 for a NOP instruction to remove a call.

Browsing managed types

!dumpdomain	Dump all domains and the loaded assemblies.	
!dumpassembly <assembly address>	Dumps all modules in this assembly.	For address see !dumpdomain
!dumpmodule -mt <module address> !dumpmodule <module address>	Dumps all types in this module including method table address. Shows assembly name and location in file system.	For address see !dumpassembly or !ip2md Not seen with !clrstack.
!dumpmt -md <method table address>	Dumps all methods of the type with method descriptors.	For address see !dumpmodule
!dumpmd <method descriptor address>	Dumps information about method descriptor.	i.e. Is jitted or not and code address.
!name2ee <module.ext> <fullQualifiedName>	Gets method table address (type info) or method descriptor address (method info). Method descriptor leads to jitted code address.	!name2ee *!Foo.Bar Name can be a full qualified method or class name. Module is case sensitive.
!ip2md <code address>	Displays method descriptor, method table and method code address from a jitted code address.	Includes also the module address. See !dumpmodule
!objsize <address>	Calculates size of an object.	Including the child objects.
!dumparray <address>	TODO simple types vs reference type?	-details

Native Heaps

!heap -x <address> !heap -i <block> !heap –p –a <address>	Searches the heap block containing address. Does not work with page heap! Shows information about a heap block. Requests page heap information. Includes the stack trace if available For stack trace on ***double free*** don't rely on reported appverif's block address!	Address returned from new, malloc etc. Keep in mind that this has additional header. -a takes care that the block containing the address is searched. Use kb instead to get the freed address from call parameters.
!heap -v <heap> !heap -s !heap -stat -h <heap> !heap -flt s <size> !heap -flt r <min> <max> !heap -a <heap>	Validates the heap. Lists all heaps and summary of usage. Breaks down all(!) blocks in size and number of allocations. Show all heap blocks filtered by UserSize. This is the size passed to ::HeapAlloc. Which is more than passed to malloc. Shows every single block.	 See !eeheap for managed usage. x *!_crtheap* finds the CRT related heaps. Pick one to see what's in it. Ctrl+Break(!)
dt _dph_block_information <address>	Displays the page heap header information of a heap block.	To figure out the header address: dd / dq the allocation address minus offset. (i.e. 80) Block starts with magic bytes: (00000000)abcdbbbb and ends with dcbabbbb(00000000)
dt _HEAP_ENTRY <address>	Displays the 8 byte block header meta data. Note: The size inside this structure is in units. Multiply with 8.	Use the !heap command because the data is scrambled since Vista.

Managed Heaps / Leaks

!dumpheap !dumpheap -stat !dumpheap -stat -type <type> !dumpheap -mt <mt> !dumpheap -min 85000 !dumpheap <minAddress> <maxAddress>	Show each single managed object and statistics. Statistics only Finds also a part of the type. Do not(!) use wildcards. Dumps objects on the LOH.	Sees also unrooted objects not yet collected. Be aware. Don't confuse **Free** (managed heap) with the free memory seen from the operating system. See !dumpheap -type Free for largest free block. See !eeheap -gc for range of LOH
!gcroot <address>	Shows the object's roots	Strong GCHandle is typically a static variable.
!eeheap -gc !eeheap -loader	Shows overall memory usage of unmanaged heap(s): GC heap size. Types and internal structures.	Includes start addresses of generations and LOH. See !heap -s for native usage.
!finalizequeue	Lists objects with finalizer ready to clean-up. Not f-reachable queue!	Increasing objects may be due to a blocked finalizer thread. Ready for finalization > 0 may already be a hint!
!gchandles	Statistics of all used handles.	Includes pinned objects.
!gchandleleaks	Do references exist to all strong and pinned gc handles?	Finds the addresses of a reference: !do poi(<foundAt>)
!verifyheap See also Mda	Validates the CLR heap(s) and shows last good object.	Does not work when garbage collection is active.

Search

Range definition: **Absolute range:** <start> <end> **Relative range** <start> L<width> <start> L-<width>	<width> specifies the objects to inspect. This depends on the type. For ranges > 256MB you have to use the ? syntax: L?<width>	
s -<type> <range> object Type: [b | w | d | q | a | u]	Searches for objects of given type.	Variation: Show only the addresses s -[1]a 0 FFFF "Hello"
s -sa <range> s -su <range>	Search any Ascii or Unicode string in range.	Specify a minimum lenth (xxx) of the strings. s -[lxxx]sa <range>
Search context record on stack: s -d esp L1000 1003f => Studio: .s -D <esp> L1000 0x1003f Can sometimes be 0x1001f . Multiple matches may indicate nested exceptions. Searches whole 32 bit space for the string "Hallo" s -a 0 L?ffffffff "Hallo" Brute force searching for objects by vtable. x moduele!class::* → module!class::'vftable' s -d 0 L?0xffffffff <vtable address> Searching addresses without caring about memory alignment s -b 0 L?0xffffffff **<LSB> < ... > <MSB>**		

Locks

!ntsdexts.locks !ntsdexts.locks -o	Shows all locked critical sections. Looks for orphaned critical sections.	
!cs [-o] <address> !cs -l	Show information about a critical section. Includes the current owner thread. Lists all locked critical sections.	-o shows the current owner stack trace. This requires symbol information.
!syncblk !syncblk <index> !dumpheap -thinlock	Searches sync block table for locked objects. Shows also the waiting threads. Shows single table entry. If table entry does not exist (thin lock). In this case !do shows also the owning thread.	Object header resides 4 bytes before the object reference. (recursion count –1) / 2 = Number of waiting threads on that sync block. If mask 0x08000000 is set the rest is the index into the table. (1 based) The owning thread refers to the ThreadOBJ listed with !threads.

Handles (Kernel objects)

!htrace -enable	Enable and make first snapshot
!htrace -snapshot	Make further snapshot.
!htrace -diff	Show outstanding handle allocation stack traces since last snapshot. Read carefully! If you make a new snapshot the same leaking stack trace is not shown any more. Even if it produces more leaks. Be also aware of handles created in other processes (DuplicateHandle).
!htrace -disable	Don't forget when used on a running server.
!htrace	Shows all information.
!htrace <handle>	Shows allocation stack of handle.
!handle	Show all handles
!handle <handle> f	Show all info about a single handle. f represents all possible options.
!token <handle>	Details about the token handle.

Loading script files

$$><script.wds	Loads and executes the script.	bp <address> "$$><f:\script.txt"
$$>a<script.wds arg1	$arg1, $arg2 etc Argument is not resolved, just passed as text.	.echo ${$arg1} .if (${/d:$arg1} != 0)

Example Script

```
.echo "Running script"

$$ Arguments are passed as text i.e. poi(esp+40)

.if (${/d:$arg1} != 0)
{
        .echo ${$arg1}
        .echo $arg1

        du $arg1

        as /mu ${/v:myVar} $arg1

        .if ($scmp(@"${myVar}", @"D:\") == 0)
        {
                .echo true
        }
        .else
        {
                .echo false
                g;
        }
}
```

COM related

!comstate	Shows all threads and to which apartment they belong.

Conditional Breakpoints by command strings

Command string is executed only when hit with g.

Execute the function, then evaluate the return value.
bp Sample!Foo::Calc "gu;.if (eax!=1) {g}"

If the condition is more complex executing a script is more convenient. Note: poi(...) is passed as text.
bp kernel32!CreateFileW "$$>a<d:\script.txt poi(esp+0n36)"

Dumps on x86 the opened file when CreateFileW is called. ($csp points to return address when function is entered. Prologue was not executed yet!)
bp Kernel32!CreateFileW "du poi(esp + 4);g"

On x64 print message depending on first method argument.
bp Foo::Func ".if (@rdx == 0) {.echo Zero;g} .else {.echo Not Zero;g}"

Stop when breakpoint hits the second time
r @$t0 = 0
bp mod!func "r @$t0=@$t0+1; if (@$t0==2) {.echo Hit} .else {gc}"

Stop when an event handle is set. You can use also dwo instead of poi.
bp KernelBase!SetEvent ".if (poi(esp+4) == 468) {.echo foo};g"

Comparing strings inside a condition: You first have to create an alias. $scmp / $sicmp work like the C functions strcmp / stricmp.
as /mu ${/v:myAlias} <address>
.if($scmp("${myAlias}", @"d.\file.txt") == 0) {.echo true} .else {.echo false}

If you want to set another breakpoint in the command string you have to escape the quotes. (\")

Extended commands / streamlining tasks

.foreach parses the output of commands and uses the values it as input for other commands.

.foreach [opt] (<var> {inCmds}) {outCmds}
.foreach [opt] /s (<var> "inString") {outCmds}
.foreach [opt] /f (<var> "inFile") {outCmds}

Options:
/pS <num> = Skips number of initial tokens.
/ps <num> = After each processed token <num> tokens are skipped.

Example (useless, only for demo)
.foreach (**obj** {s -[1]a 0 FFFFFF "Cpp"}) {da **{$obj}**}

.shell [options] [cmd] Options -ci "cmd1; cmd2; ..."	Executes a process and redirects output to WinDbg. Executes command and redirects output to shell process.	.shell -ci "da 0x003a93b2" **FIND "whatever" /i** /i is a case insensitive option for FIND
.if (cond) {cmds} .elseif (cond) {cmds} .else {cmds}	Conditional branches for scripts and command strings.	.if (@eax = 0) {.echo TRUE} .else {.echo FALSE; g}
command; z (expr)	Execute command while expression is true	
Scripts misc	Commands can be separated by semicolon. Comments are introduced via $$	$scmp(s1, s2) $sicmp(s1, s2) $spat(s, pattern)
as [type] ${/v:<name>} <value>	Simplified pattern for declaring variables ("aliases")	Type: /ma = Ascii string /mu = Unicode string
String literals	@ disables escaping in strings like in C#	@"${myVar}"
.echo text	same as .echo "text"	

Unassemble

uf /c /D <address>	Shows only the calls (/c) to other methods as links! <address> accepts also symbolic names.	uf /c /D kernel32! GetFullPathNameW uf @eip Finds start and end automatically.
u <address> ub <address>	General disassemble Backward	u @eip ub . i.e. verify call stacks
!u <address>	Unassemble and resolve managed calls.	
!dumpil <method descr. address>		

Security

!token -n	Shows access token of current thread context.	-n resolves the cryptic SIDs (users and groups)

Important Registers and Pseuderegisters

x86	x64	
ECX	RCX	this pointer on a method call
EAX	RAX	Return value
	R8...R15	New additional general purpose
	RCX, RDX, R8 and R9	The first parameters on a function call. In case a method is called the this pointer is stored in RCX.
FS FS[0]		Contains pointer to TIB Points to exception record.
Pseudo registers		
$peb $teb $exp $retreg $csp $ip $ra $tid $t0 … $t19 $ptrsize	 Last evaluated expression eax on x86 rax on x64 esp on x86 rsp on x64 Return address Thread ID. (Not number!) General purpose registers Size of a pointer.	As with normal registers, pseudo registers must be escaped with @ in expressions. dt _PEB @$peb So the debugger can distinguish them from normal symbols. Example: r $csp is the same as ? $csp
r RAX = 0		Sets value to register

API Logger

Manifest files reside in folder winext\manifest. Include in main.h

!logexts.loge	Inject and enable logger	Shows output folder.
!logc !logc d * !logc e <numb>	Show all categories Disable all categories Enable category. See !logc.	
!logo !logo e t !logo e d	Show output options like output folder. Enables output into trace file. Enables output into debugger	
!logd	Disables logging.	

Command Tree Template

```
windbg ANSI Command Tree 1.0
title {"Common Commands"}
body
{"All Commands"}
   {"Logging"}
      {"Open Log"} {".logopen"}
   {"Modules"}
      {"All Modules"} {"lm"}
```

Memory Usage

!address -summary !address <address>	Shows memory usage summary. Information about the address.	RegionUsageIsVAD = VirtualAlloc RegionUsageHeap = Allocated by a heap manager. RegionUsageStack RegionUsageImage RegionUsageFree etc.
!vadump	Dumps all regions in virtual address space.	Log to file!

Kernel Debugging Cheat Sheet

Enable Kernel Debugging: bcdedit -debug on

Kernel debugger: Switching Context		
bcdedit -debug on	Enables Kernel debugging	Reboot the system.

Kernel debugger: Switching Context		
~<cpu>	Set CPU context. Do not confuse with ~ in user mode (thread context).	~0 Switches to first CPU

Kernel debugger: Processes		
!process <pid or address> <flags> [image]	Lists running processes and related infos. <flags> = 0 Minimum output. <flags> = 7 Maximum output, including stack traces of all threads.	Instead of the PID you can also specify the address of the process kernel object (nt!_EPROCESS), if known. This address is shown in the output as PROCESS. Note: Process ID (PID) == Process Client ID (CID)
!process 0 0 !process -1 0 !process 0 0 calc.exe	Show all processes. Show current process running on CPU when debugger break happened.. Show all calculator processes.	
.process /r /p <address>	Switches process context. /r Reloads user mode symbols. /p Command takes address of process kernel object.	Now you can use lm for example to get the loaded modules.
.process /i <address> .reload /user	Intrusive switch to process. Only live debugging. This actually makes the process active. Needed to set user mode breakpoints. Reloads user symbols after intrusive switch.	You need to run (g) first, before breaking back soon with the new default context.

Kernel debugger: Threads		
!thread [address] Alternative: .thread <address> k	Shows stack trace of the given optional thread address. Includes the owning process. First set the thread context, then you can use the k command.	The address is shown in the output of !process <pid> 7 as THREAD.

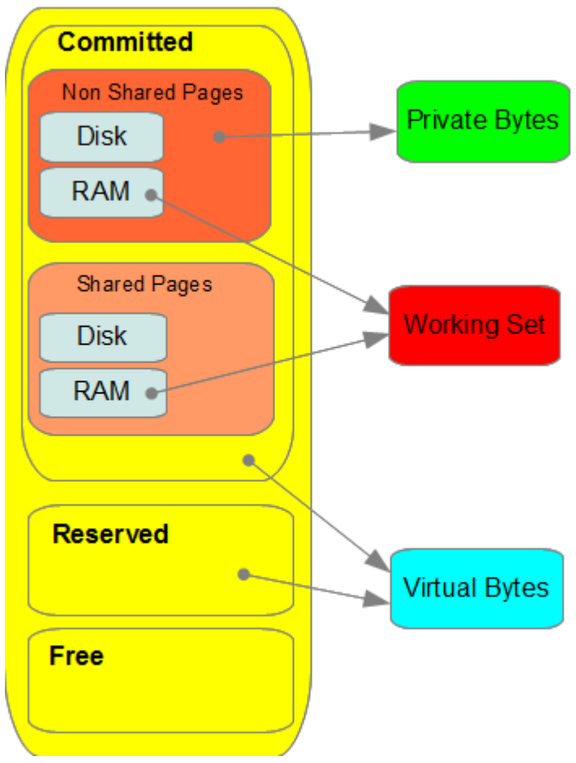
Kernel debugger: Breakpoints		
For kernel mode code use the known commands. For user mode code first switch default process context. bp /p <pid> <symbol>	x, bp etc. .process /i <address> .reload /user x <symbolic name> bp <address>	 Breakpoints in user mode are set relative the default process! This is the process that was active when stopped.

Kernel debugger: Local		
livekd -w	-w starts WinDbg instead of kd.	Copy livekd beside WinDbg. Does not allow to influence execution. i.e. setting breakpoints.

Kernel debugger: Inspect Objects		
!object <address>	Shows object type, object header address and handle count of the given object.	dt nt! _OBJECT_HEADER <header address> dt nt! _OBJECT_TYPE <type address>

Reference

(Virtual) Memory classification	
Working Set aka Mem Usage	Pages that are currently in RAM. Shared Pages are included. So you cannot simply add the Working Sets of two processes. You would count Shared Pages twice. Standby page list is not included here (no longer in used but not paged out yet).
Private Bytes aka VMSize (< Win7) aka Commit Size	Total private virtual memory allocated. This is committed memory. It does **not** include Shared Pages. But be aware that memory allocated by shared libraries is accounted! So you cannot say if an increase in private bytes is due to the application itself or one of its DLLs. It does not matter if the pages are in memory or on disk. The standby page list is accounted here. Notes: Steady increasing value is a good indicator for a leak! But it is no proof! It can be that private bytes increase while virtual memory is stable for a while and then makes a sudden increase. This can be explained by a heap manager reserving a new segment.
Virtual Bytes	Total size of the virtual address space. These are the committed and reserved pages together. Shared Pages are included. Because it contains also the reserved pages this value can be much larger than private bytes. The only thing this value tells you is how close you are to the limit. Notes: If virtual bytes increases but private bytes do not it could mean that somebody either reserves memory directly via VirtualAlloc or it could be a sign of Heap fragmentation.
Committed bytes	Total number of bytes in the committed state.
Shared Pages	Pages shared between processes like most DLLs and memory mapped files. Read only data like code or constants can be shared between processes and are therefore typically not included in the private bytes. Read / write data like a Heap is not shareable.
System Commit Limit aka Commit Charge (Limit)	Size of the page file plus physical memory. Since the page file can grow (or shrink) this value is not constant. Loaded images are not charged against the limit because they are backed up by the file already.



References:
http://stackoverflow.com/questions/1984186/what-is-private-bytes-virtual-bytes-working-set

.NET reference object's layout	
There are two double words overhead.	
<address> - 4	Sync block
<address>	Method table
<address> + 4	Object data. For a one dimensional array the data starts with the array size.

Stack

Stack pointer @csp points always on the last element. Decremented before the next value is pushed. The value size is variable.
Stack grows from top to bottom. Limits: See !teb

X86 (_stdcall)

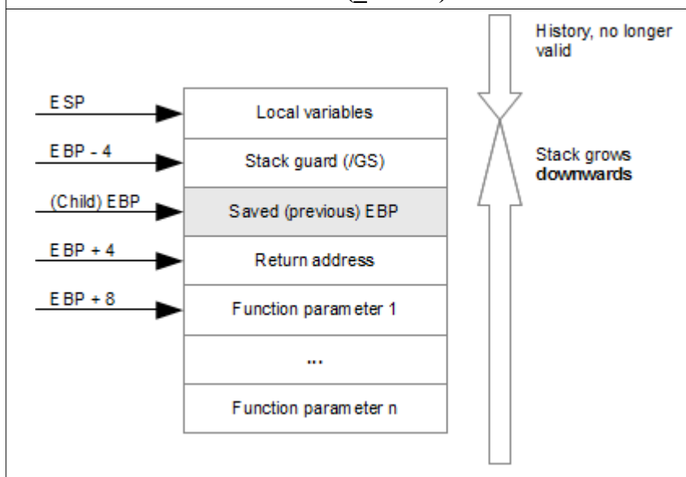


Image shows direction of WinDbg output.
Breakpoint at start of function before the prologue is executed: ESP points to the return address. ESP+4 catches a function argument.

Child EBP in WinDbg shows the stack address of the saved previous frame pointer. Or at the beginning of a function: where the value is going to be stored after the prologue. WinDbg is smart enough to show the correct value even at function entry.
Therefore: Value of Child **EBP = EBP** for current frame.
Set hardware breakpoint on stack guard to catch buffer overruns.
Use dps <stack addr> to manually reconstruct the call chain.

x64

Only one calling convention!
Caller uses RCX, RDX, R8 and R9 in this order to pass arguments.
RSP is stable after the prologue.

Calling Conventions for x86	
this	Parameters pushed right to left. Removed by caller. Default for C++ classes. This parameter is passed in ECX.
__stdcall	Parameters pushed right to left. Removed by callee. Default for most system functions. Decoration example: _Foo@12 The number is the decimal bytes in argument list.
__cdecl	Parameters pushed right to left. Removed by caller. Only convention that allows variable function arguments. Default for C / C++ functions. Decoration example: _Foo
__fastcall	First two DWORD parameters are passed in ECX, EDX. Remaining ones are pushed right to left. Removed by callee. Used by .NET Jit Compiler. Decoration example: @Foo@12

Important Memory Fill Patterns	
0xCC	Uninitialized stack variables. (/GZ Compiler Option)
0xCD	Used by Debug-CRT for uninitialized memory on the heap. (Memory returned by „new“ or „malloc“ not written to yet.) Clean Memory.
0xDD	Used by Debug-CRT for freed memory. „Dead Memory“. Used to find dangeling pointers.
0xFD	Used by Debug-CRT as guard bytes before and after memory blocks. “Fence Memory”. Before and after 0xCD pattern.
0xAB	::HeapAlloc suffix (guard) bytes. When started under debugger.
0xFEEEFEEE	::HeapFree freed memory. When started under debugger.
	For the heap manager guards the application must be started under the debugger. For the CRT heap guards the application must be compiled in debug mode.

WOW64 Virtual Environment
Calls to registry are virtualized. i.e.: HLCR**Wow6432Node** HKLM\SOFTWARE**Wow6432Node** HKLM\SOFTWARE**Classes\Wow6432Node** HKCU\Software**Wow6432Node** HKCU\Software\Classes**Wow6432Node** The virtualization includes COM object registration.
File system access to **%SystemRoot%\system32** is redirected to **%SystemRoot%\sysWOW64**

- Tools and System Configuration -

Just in Time debugger

Automatically start debugger when an application crashes.
Note that Windows Error Reporting has precedence!

Via Visual Studio (Admin)

Open **Options | Debugging | Just-In-Time**. There select the type of code for Just-In-Time debugging. Next time you get a crash, the Visual Studio Just-In-Time debugger will come up

Via WinDbg

Run from command line: Windbg.exe -I

Directly editing the Registry

`HKLM\SOFTWARE[\Wow6432Node]\Microsoft\Windows NT\CurrentVersion\AeDebug`

Auto (SZ)	1: Start debugger without message box
Debugger (SZ)	"C:\debuggers\windbg.exe" -p %ld -e %ld -g

Getting a crash dump

Procdump

-e Writes dump on second chance (not handled) exception.

-e 1 Writes dumps on first chance exception.

-f Filters first chance exceptions. Supports wildcard *. Example: **-f "AccessViolation"**. Empty filter **-f ""** only monitors the occurring exceptions on the screen.

-ma Writes all process memory.

-t Writes a dump when process terminates. Useful when application exits without exception on errors.

-n Number of dumps before exit. Important if first chance exceptions are dumped. Procmon would exit after the first one otherwise. **-n <number>**

-x Launches the executable. Note that the executable comes after the dump folder! Procmon -e -x . foo.exe

-i Installs procdump as post mortem debugger (AeDebug). Only limited options here! Target directory is printed.

procdump [-n exceeds] [-e [1]] [-f <filter,...>] [-ma] [-t] <process name or service name or PID> [dump file/folder]
| -x <dump file/folder> <image file> [arguments]

Examples

Writes one dump now (hang)!
Procdump [-ma] <process name or PID>

Attaches to running process and dumps on not handled exception.
procdump -e [-ma] [-t] <process name or PID> <dump>

Attaches to running process and dumps on first chance exceptions.
procdump -e 1 -n 1000 <ProcessNameWithoutExcention>

Dumps only on AccessViolation crashes.
Procdump [-ma] -e 1 -f "AccessViolation" <process name or PID>

Application is started instead of attached. Note the dot as folder!
procdump -e -x . <image name> <dump>

Installs / uninstalls procdump as AeDebug post mortem debugger
procdump -ma -i
procdump -u

By default a 32 bit dump is created for 32 bit processes on a 64 bit Windows.

Windows Error Reporting

Save User Mode Dumps locally.

`HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\LocalDumps\<module.exe>`

`Add Disabled (DWORD) = 1 to LocalDumps to disable it globally.`

DumpFolder (SZ)	Default= %LOCALAPPDATA%\CrashDumps
DumpType (DWORD)	1: Mini dump 2: Full dump

Dump files for System crashes are configured in “Startup and Recovery” dialogue (“Systemeigenschaften”).

Managed Debugging Assistants (MDA)

Main switch to enable MDA
Environment variable COMPLUS_Mda=1
Separated by semicolon you can also directly set the MDAs there.

Or in Visual Studio: See exception window.

Create xml file and name it <application.exe>.mda.config
Put this file beside the application.

```
<mdaConfig>
    <assistants>
        <mda1 />
        <mda2 />
    </assistants>
</mdaConfig>
```

Important MDAs for native / managed applications

<bindingFailure />	Output appears in debugger. To resolve binding related problems use **fuslogvw.exe**
<gcUnmanagedToManaged /> <gcManagedToUnmanaged />	Triggered on each unmanaged to managed transition. Detect corruptions early.
<pInvokeStackImbalance />	Finds calling convention problems with Pinvoke.
<callbackOnCollectedDelegate listSize="1500" />	Holds list of collected callbacks and checks if the are still used. (PInvoke) listSize is the numer of thunks to keep. If a managed delegate is passed to a synchronous Pinvoke it is pinned automatically.
raceOnRCWCleanup	Finds Marshal.ReleaseCOMObject issues.

Additional environment variables to debug managed heap corruptions i.e. for unsafe code without native transitions:

set COMPLUS_HeapVerify=1	Automatically validates the CLR heap on each garbage collection.
set COMPLUS_GCStress=1	Triggers garbage collection more often. May gets you closer to the problem location. Use variables in combination. Last strategy because of the overhead.

UMDH

https://github.com/ATrefzer/UmdhGui

Prerequisites:
Enable stack trace database for examined application
gflags -i <app>
gflags -i <app> +ust
gflags -i <app> -tracedb <mb>

Ensure the _NT_SYMBOL_PATH environment variable is set.

Note: If stack trace database is not enabled via gflags it gets enabled automatically with the first snapshot. This applies only for the current examined application and is not persistent!

umdh -p:<pid> -f:<file> umdh -d <fileA> <fileB> -f:<fileDiff>	Creates snapshot Resolves symbols and creates difference file.

Registry location for gflags settings

HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options

Visual Studio Watch Window Formats

Append separated with comma in watch window.

wm	Show as window message	0x0010,wm => WM_CLOSE
!	Show as raw format.	Disables available visualizer.
hr	Show Win32 error code or HRESULT as message	err, hr err represents the last error.
x,X	Hexadecimal	15,X => 0xF
d / u	Signed / unsigned decimal integer	

Disable JIT Code Optimization

Environment Variables

Disable usage of pre-compiled code (NGEN)

.NET Framework or Core <= 2.x	COMPlus_ZapDisable =1
Net Core:	COMPlus_ReadyToRun = 0

Debugger Settings

Studio → Debug → Suppress JIT Optimization on module load
Avoid attaching to process. Modules may have been loaded.

Another option:
If you want to disable the JIT optimization for assembly xyz.dll you have to place a file xyz.ini (not xyz.dll.ini) beside the assembly to debug.

```
[.NET Framework Debugging Control]
GenerateTrackingInfo=1    // Not needed any more
AllowOptimize=0
```

Modifying executable flags

Large address awareness

dumpbin.exe /headers <app.exe>
=> Application can handle large (>2GB) addresses

editbin.exe /largeaddressaware <app.exe>

Resolving dependencies

Shows the native dependencies of a module

dumpbin /dependents <PathToModule>