

## WinDbg — the Fun Way: Part 1

A while ago WinDbg added support for a new [debugger data model](#), a change that completely changed the way we can use WinDbg. No more horrible MASM commands and obscure syntax. No more copying addresses or parameters to a notepad file so that you can use them in the next commands without scrolling up. No more running the same command over and over with different addresses to iterate over a list or an array.

This is part 1 of this guide, because I didn't actually think anyone would read through 8000 words of me explaining WinDbg commands. So you get 2 posts of 4000 words! That's better, right?

In this first post we will learn the basics of how to use this new data model — using custom registers and new built-in registers, iterating over objects, searching them and filtering them and customizing them with anonymous types. And finally we will learn how to parse arrays and lists in a much nicer and easier way than you're used to.

And in the net post we'll learn the more complicated and fancier methods and features that this data model gives us.

Now that we all know what to expect and grabbed another cup of coffee, let's start!

This data model, accessed in WinDbg through the `dx` command, is an extremely powerful tool, able to define custom variables, structures, functions and use a wide range of new capabilities. It also lets us search and filter information with LINQ — an infrastructure similar to SQL.

This data model is documented and even has usage examples on github. Also all of its modules have documentation that can be viewed in the debugger with `dx -v <method>` (though you will get the same documentation if you run `dx <method>` without the `-v` flag):

### **`dx -v Debugger.Utility.Collections.FromListEntry`**

`Debugger.Utility.Collections.FromListEntry [FromListEntry(ListEntry, [<ModuleName | ModuleObject>], TypeName, FieldExpression) – Method which converts a LIST_ENTRY specified by the 'ListEntry' parameter of types whose name is specified by the string 'TypeName' and whose embedded links within that type are accessed via an expression specified by the string 'FieldExpression' into a collection object. If an optional module name or object is specified, the type name is looked up in the context of such module]`

There has also been some [external](#) documentation, but I felt like there were things that needed further explanation and that this feature is worth more attention than it receives.

## Custom Registers

First, NatVis adds the option for custom registers. Kind of like MASM had `@$t1`, `@$t2`, `@$t3`, etc. Only now you can call them whatever name you want, and they can have a type of your choice:

```
dx @$myString = "My String"
dx @$myInt = 123
```

We can see all our variables with `dx @$vars` and remove them with `dx @$vars.Remove("var name")`, or clear all with `dx @$vars.Clear()`. We can also use `dx` to show handle more complicated structures, such as an `EPROCESS`. As you might know, symbols in public PDBs don't have types. That's why with the old debugger we had to print the System process like this:

```
0: kd> x nt!PsInitialSystemProcess
fffff807`4ef843a0 nt!PsInitialSystemProcess = <no type information>
0: kd> dt nt!_EPROCESS fffff807`4ef843a0
+0x000 Pcb : _KPROCESS
+0x2e0 ProcessLock : _EX_PUSH_LOCK
+0x2e8 UniqueProcessId : (null)
...
```

We first had to get the address of `PsInitialSystemProcess` and then print it as an `EPROCESS`. With `dx` we can be smarter and cast symbols directly, using c-style casts. But when we try to cast `nt!PsInitialSystemProcess` to a pointer to an `EPROCESS`:

```
dx @$systemProc = (nt!_EPROCESS*)nt!PsInitialSystemProcess
Error: No type (or void) for object at Address 0xfffff8074ef843a0
```

We get an error.

Like we said, symbols have no type. And we can't cast something with no type. So we need to take the address of the symbol, and cast it to a pointer to the type we want (In this case, `PsInitialSystemProcess` is already a pointer to an `EPROCESS` so we need to cast its address to a pointer to a pointer to an `EPROCESS`).

```
dx @$systemProc = *(nt!_EPROCESS**)& nt!PsInitialSystemProcess
```

Now that we have a typed variable, we can access its fields like we would do in C:

```
0: kd> dx @$systemProc->ImageFileName
@$systemProc->ImageFileName [Type: unsigned char [15]]
[0] : 0x53 [Type: unsigned char]
[1] : 0x79 [Type: unsigned char]
[2] : 0x73 [Type: unsigned char]
[3] : 0x74 [Type: unsigned char]
[4] : 0x65 [Type: unsigned char]
[5] : 0x6d [Type: unsigned char]
[6] : 0x0 [Type: unsigned char]
```

And we can cast that to get a nicer output:

```
dx (char*)@$systemProc->ImageFileName
(char*)@$systemProc->ImageFileName : 0xfffffc10c8e87e710 :
"System" [Type: char *]
```

We can also use `ToDisplayString` to cast it from a `char*` to a string. We have two options — `ToDisplayString("s")`, which will cast it to a string and keep the quotes as part of the string, or `ToDisplayString("sb")`, which will remove them:

```
dx ((char*)@$systemProc->ImageFileName).ToDisplayString("s")
((char*)@$systemProc->ImageFileName).ToDisplayString("s") : "System"
Length : 0x8
dx ((char*)@$systemProc->ImageFileName).ToDisplayString("sb")
((char*)@$systemProc->ImageFileName).ToDisplayString("sb") : System
Length : 0x6
```

## Built-in Registers

This is fun, but for processes (and a few other things) there is an even easier way. Together with `NatVis` implementation in `WinDbg` we got some “free” registers already containing some useful information — `curframe`, `curprocess`, `cursession`, `curstack` and `curthread`. It’s not hard to guess their contents by the names, but let’s take a look:

### @\$curframe

Gives us information about the current frame. I never actually used it myself, but it might be useful:

```
dx -r1 @$curframe.Attributes
@$curframe.Attributes
InstructionOffset : 0xffffffff8074ebda1e1
ReturnOffset : 0xffffffff80752ad2b61
FrameOffset : 0xffffffff80751968830
StackOffset : 0xffffffff80751968838
FuncTableEntry : 0x0
Virtual : 1
FrameNumber : 0x0
```

### @\$curprocess

A container with information about the current process. This is not an `EPROCESS` (though it does contain it). It contains easily-accessible information about the current process, like its threads, loaded modules, handles, etc.

```
dx @$curprocess
@$curprocess : System [Switch To]
KernelObject [Type: _EPROCESS]
Name : System
Id : 0x4
Handle : 0xf0f0f0f0
Threads
Modules
Environment
Devices
Io
```

In `KernelObject` we have the `EPROCESS`, but we can also use the other fields. For example, we can access all the handles held by the process through `@$curprocess.Io.Handles`, which will lead us to an array of handles, indexed by their handle number:

```
dx @$curprocess.Io.Handles
```

```
@$curprocess.Io.Handles
```

```
[0x4]
```

```
[0x8]
```

```
[0xc]
```

```
[0x10]
```

```
[0x14]
```

```
[0x18]
```

```
[0x1c]
```

```
[0x20]
```

```
[0x24]
```

```
[0x28]
```

```
...
```

System has a lot of handles, these are just the first few! Let's just take a look at the first one (which we can also access through `@$curprocess.Io.Handles[0x4]`):

```
dx @$curprocess.Io.Handles.First()
```

```
@$curprocess.Io.Handles.First()
```

```
Handle : 0x4
```

```
Type : Process
```

```
GrantedAccess : Delete | ReadControl | WriteDac | WriteOwner |
```

```
Synch | Terminate | CreateThread | VMOp | VMRead | VMWrite | DupHandle |
```

```
CreateProcess | SetQuota | SetInfo | QueryInfo | SetPort
```

```
Object [Type: _OBJECT_HEADER]
```

We can see the handle, the type of object the handle is for, its access and even have a pointer to the object itself!

There are plenty more things to find under this register, and we encourage you to investigate them, but we will not show all of them.

By the way, have we mentioned already that `dx` allows tab completion?

```
@$cursession
```

As its name suggests, this register gives us information about the current debugger session:

```
dx @$cursession
```

```
@$cursession : Remote KD:
```

```
KdSrv:Server=@{<Local>},Trans=@{NET:Port=55556,Key=1.2.3.4,Target=192.168.251.21}
```

```
Processes
```

```
Id : 0
```

```
Devices
```

```
Attributes
```

So we can get information about our debugger session, which is always fun. But there are more useful things to be found here, such as the Processes field, which is an array of all running processes, indexed by their PID. Let's pick one of them:

```
dx @$cursession.Processes[0x1d8]
@$cursession.Processes[0x1d8]           : smss.exe [Switch To]
  KernelObject      [Type: _EPROCESS]
  Name              : smss.exe
  Id                : 0x1d8
  Handle            : 0xf0f0f0f0
  Threads
  Modules
  Environment
  Devices
  Io
```

Now we can get all that useful information about every single process! We can also search for processes containing specific things, or filter processes based on a search (such as name, specific modules loaded into them, strings in their command line, etc. But we will explain all of that later.

[@\\$curstack](#)

This register contains a single field — frames — which shows us the current stack in an easily-handled way:

```
dx @$curstack.Frames
@$curstack.Frames
  [0x0]           : nt!DbgBreakPointWithStatus + 0x1 [Switch To]
  [0x1]           : kdnic!TXTransmitQueuedSends + 0x125 [Switch To]
  [0x2]           : kdnic!TXSendCompleteDpc + 0x14d [Switch To]
  [0x3]           : nt!KiProcessExpiredTimerList + 0x169 [Switch To]
  [0x4]           : nt!KiRetireDpcList + 0x4e9 [Switch To]
  [0x5]           : nt!KiIdleLoop + 0x7e [Switch To]
```

[@\\$curthread](#)

Gives us information about the current thread, similar to @\$curprocess:

```
dx @$curthread
@$curthread           : nt!DbgBreakPointWithStatus+0x1
(fffff807`4ebda1e1) [Switch To]
  KernelObject      [Type: _ETHREAD]
  Id                : 0x0
  Stack
  Registers
  Environment
```

It contains the ETHREAD in KernelObject, but also contains the TEB in Environment, and can show us the thread ID, stack and registers.

```
dx @$curthread.Registers
```

```
@$curthread.Registers
```

```
User
```

```
Kernel
```

```
SIMD
```

```
FloatingPoint
```

We have them conveniently separated to user, kernel, SIMD and FloatingPoint registers, and we can look at each separately:

```
dx -r1 @$curthread.Registers.Kernel
```

```
@$curthread.Registers.Kernel
```

```
cr0      : 0x3e00220074006e
cr2      : 0x650074005f006d
cr3      : 0x6c004100740078
cr4      : 0x6d006e00670069
cr8      : 0xa000d003e006d
gdtr     : 0x3e006d00650074
gdtl     : 0x49
idtr     : 0x720074006e006f
idt1     : 0x43
tr       : 0x6f
ldtr     : 0x6c
kmxcsr   : 0x29002a
kdr0     : 0x3c0074006e0065
kdr1     : 0x6500740049002f
kdr2     : 0xa000d003e006d
kdr3     : 0x20002000200020
kdr6     : 0xffffe0ff0
kdr7     : 0x400
xcr0     : 0xa000d000a000d
```

## Searching and Filtering

A very useful thing that NatVis allows us to do, which we briefly mentioned before, is searching, filtering and ordering information in an SQL-like way through Select, Where, OrderBy and more.

For example, let's try to find all the processes that don't enable high entropy ASLR. This information is stored in the EPROCESS->MitigationFlags field, and the value for HighEntropyASLREnabled is 0x20 (all values can be found [here](#) and in the public symbols).

First we'll declare a new register with that value, just to make things more readable:

```
0: kd> dx @$highEntropyAslr = 0x20
```

```
@$highEntropyAslr = 0x20 : 32
```

And then create our query to iterate over all processes and only pick ones where the HighEntropyASLREnabled bit is not set:

```

dx -r1 @$cursession.Processes.Where(p => (p.KernelObject.MitigationFlags &
@$highEntropyAslr) == 0)
@$cursession.Processes.Where(p => (p.KernelObject.MitigationFlags &
@$highEntropyAslr) == 0)
    [0x910]      : spoolsv.exe [Switch To]
    [0xb40]      : IpOverUsbSvc.exe [Switch To]
    [0x1610]     : explorer.exe [Switch To]
    [0x1d8c]     : OneDrive.exe [Switch To]

```

Or we can check the flag directly through MitigationFlagsValues and get the same results:

```

dx -r1 @$cursession.Processes.Where(p =>
(p.KernelObject.MitigationFlagsValues.HighEntropyASLREnabled == 0))
@$cursession.Processes.Where(p =>
(p.KernelObject.MitigationFlagsValues.HighEntropyASLREnabled == 0))
    [0x910]      : spoolsv.exe [Switch To]
    [0xb40]      : IpOverUsbSvc.exe [Switch To]
    [0x1610]     : explorer.exe [Switch To]
    [0x1d8c]     : OneDrive.exe [Switch To]

```

We can also use Select() to only show certain attributes of things we iterate over. Here we choose to see only the number of threads each process has:

```

dx @$cursession.Processes.Select(p => p.Threads.Count())
@$cursession.Processes.Select(p => p.Threads.Count())
    [0x0]      : 0x6
    [0x4]      : 0xeb
    [0x78]     : 0x4
    [0x1d8]    : 0x5
    [0x244]    : 0xe
    [0x294]    : 0x8
    [0x2a0]    : 0x10
    [0x2f8]    : 0x9
    [0x328]    : 0xa
    [0x33c]    : 0xd
    [0x3a8]    : 0x2c
    [0x3c0]    : 0x8
    [0x3c8]    : 0x8
    [0x204]    : 0x15
    [0x300]    : 0x1d
    [0x444]    : 0x3f
    ...

```

We can also see everything in decimal by adding “, d” to the end of the command, to specify the output format (we can also use b for binary, o for octal or s for string):

```

dx @$cursession.Processes.Select(p => p.Threads.Count()), d
@$cursession.Processes.Select(p => p.Threads.Count()), d
    [0]      : 6
    [4]      : 235
    [120]    : 4

```

```

[472]          : 5
[580]          : 14
[660]          : 8
[672]          : 16
[760]          : 9
[808]          : 10
[828]          : 13
[936]          : 44
[960]          : 8
[968]          : 8
[516]          : 21
[768]          : 29
[1092]         : 63
...

```

Or, in a slightly more complicated example, see the ideal processor for each thread running in a certain process (I chose a process at random, just to see something that is not the System process):

```

dx -r1 @$cursession.Processes[0x1b2c].Threads.Select(t =>
t.Environment.EnvironmentBlock.CurrentIdealProcessor.Number)
@$cursession.Processes[0x1b2c].Threads.Select(t =>
t.Environment.EnvironmentBlock.CurrentIdealProcessor.Number)
[0x1b30]       : 0x1 [Type: unsigned char]
[0x1b40]       : 0x2 [Type: unsigned char]
[0x1b4c]       : 0x3 [Type: unsigned char]
[0x1b50]       : 0x4 [Type: unsigned char]
[0x1b48]       : 0x5 [Type: unsigned char]
[0x1b5c]       : 0x0 [Type: unsigned char]
[0x1b64]       : 0x1 [Type: unsigned char]

```

We can also use `OrderBy` to get nicer results, for example to get a list of all processes sorted by alphabetical order:

```

dx -r1 @$cursession.Processes.OrderBy(p => p.Name)
@$cursession.Processes.OrderBy(p => p.Name)
[0x1848]       : ApplicationFrameHost.exe [Switch To]
[0x0]          : Idle [Switch To]
[0xb40]        : IpOverUsbSvc.exe [Switch To]
[0x106c]        : LogonUI.exe [Switch To]
[0x754]         : MemCompression [Switch To]
[0x187c]        : MicrosoftEdge.exe [Switch To]
[0x1b94]        : MicrosoftEdgeCP.exe [Switch To]
[0x1b7c]        : MicrosoftEdgeSH.exe [Switch To]
[0xb98]         : MsMpEng.exe [Switch To]
[0x1158]        : NisSrv.exe [Switch To]
[0x1d8c]        : OneDrive.exe [Switch To]
[0x78]          : Registry [Switch To]
[0x1ed0]        : RuntimeBroker.exe [Switch To]
...

```

If we want them in a descending order we can use `OrderByDescending`.



But what if we want to pick more than one attribute to see? There is a solution for that too.

## Anonymous Types

We can declare a type of our own, that will be unnamed and only used in the scope of our query, using this syntax: `Select(x => new { var1 = x.A, var2 = x.B, ... })`.

We'll try it out on one of our previous examples. Let's say for each process we want to show a process name and its thread count:

```
dx @$cursession.Processes.Select(p => new {Name = p.Name, ThreadCount =
p.Threads.Count()})
@$cursession.Processes.Select(p => new {Name = p.Name, ThreadCount =
p.Threads.Count()})
[0x0]
[0x4]
[0x78]
[0x1d8]
[0x244]
[0x294]
[0x2a0]
[0x2f8]
[0x328]
...
```

But now we only see the process container, not the actual information. To see the information itself we need to go one layer deeper, by using `-r2`. The number specifies the output recursion level. The default is `-r1`, `-r0` will show no output, `-r2` will show two levels, etc.

```
dx -r2 @$cursession.Processes.Select(p => new {Name = p.Name, ThreadCount
= p.Threads.Count()})
@$cursession.Processes.Select(p => new {Name = p.Name, ThreadCount =
p.Threads.Count()})
[0x0]
    Name           : Idle
    ThreadCount    : 0x6
[0x4]
    Name           : System
    ThreadCount    : 0xeb
[0x78]
    Name           : Registry
    ThreadCount    : 0x4
[0x1d8]
    Name           : smss.exe
    ThreadCount    : 0x5
[0x244]
    Name           : csrss.exe
    ThreadCount    : 0xe
[0x294]
    Name           : wininit.exe
    ThreadCount    : 0x8
```

```

[0x2a0]
  Name           : csrss.exe
  ThreadCount    : 0x10
...

```

This already looks much better, but we can make it look even nicer with the new graphic module, accessed through the -g flag:

```

dx -g @$cursession.Processes.Select(p => new {Name = p.Name, ThreadCount =
p.Threads.Count()})

```

	Name	ThreadCount
[0x0]	Idle	0x6
[0x4]	System	0xeb
[0x78]	Registry	0x4
[0x1d8]	smss.exe	0x5
[0x244]	csrss.exe	0xe
[0x294]	wininit.exe	0x8
[0x2a0]	csrss.exe	0x10
[0x2f8]	winlogon.exe	0x9
[0x328]	services.exe	0xa
[0x33c]	lsass.exe	0xd
[0x3a8]	svchost.exe	0x2c
[0x3c0]	fontdrvhost.exe	0x8
[0x3c8]	fontdrvhost.exe	0x8
[0x204]	svchost.exe	0x15
[0x300]	dwm.exe	0x1d
[0x444]	svchost.exe	0x3f
[0x450]	svchost.exe	0x4a
[0x4b0]	svchost.exe	0x1a
[0x4cc]	svchost.exe	0x17
[0x4d4]	svchost.exe	0x19
[0x58c]	svchost.exe	0x14
[0x5e4]	svchost.exe	0x22
[0x610]	svchost.exe	0x9
[0x6e0]	VSSVC.exe	0x9
[0x754]	MemCompression	0x39
[0x7a8]	svchost.exe	0x21
[0x7d4]	svchost.exe	0x16
[0x670]	svchost.exe	0xb
[0x854]	svchost.exe	0x8
[0x870]	svchost.exe	0x12
[0x910]	spoolsv.exe	0x11
[0x964]	svchost.exe	0x17
[0x9e8]	svchost.exe	0x19
[0xa6c]	svchost.exe	0x8

Ok, this just looks awesome. And yes, these headings are clickable and will sort the table!

And if we want to see the PIDs and thread numbers in decimal we can just add “, d” to the end of the command:

```
dx -g @$cursession.Processes.Select(p => new {Name = p.Name, ThreadCount = p.Threads.Count()}),d
```

### [Arrays and Lists](#)

DX also gives us a new, much easier way, to handle arrays and lists with new syntax.

Let’s look at arrays first, where the syntax is `dx *(TYPE(*)[Size])<pointer to array start>`.

For this example we will dump the contents on `PsInvertedFunctionTable`, which contains an array of up to 256 cached modules in its `TableEntry` field.

First we will get the pointer of this symbol and cast it to `_INVERTED_FUNCTION_TABLE`:

```
dx @$inverted = (nt!_INVERTED_FUNCTION_TABLE*)&nt!PsInvertedFunctionTable
@$inverted = (nt!_INVERTED_FUNCTION_TABLE*)&nt!PsInvertedFunctionTable
: 0xffffffff8074ef9b010 [Type: _INVERTED_FUNCTION_TABLE *]
    [+0x000] CurrentSize      : 0xbe [Type: unsigned long]
    [+0x004] MaximumSize     : 0x100 [Type: unsigned long]
    [+0x008] Epoch           : 0x19e [Type: unsigned long]
    [+0x00c] Overflow         : 0x0 [Type: unsigned char]
    [+0x010] TableEntry       [Type: _INVERTED_FUNCTION_TABLE_ENTRY [256]]
```

Now we can create our array. Unfortunately, the size of the array has to be static and can’t use a register, so we need to input it manually, based on `CurrentSize` (or just set it to 256, which is the size of the whole array). And we can use the graphic module to print it nicely:

```
dx -g @$tableEntry =
*(nt!_INVERTED_FUNCTION_TABLE_ENTRY(*)[0xbe])@$inverted->TableEntry
```

	(+) FunctionTable	(+) DynamicTable	ImageBase	SizeOfImage	SizeOfTable
[0]	0xfffff8074ef21000	0xfffff8074ef21000	0xfffff8074ea10000	0xab7000	0x5e380
[1]	0xfffffec36ca334000	0xfffffec36ca334000	0xfffffec36ca000000	0x3a2000	0x1e168
[2]	0xfffffec36ca504000	0xfffffec36ca504000	0xfffffec36ca4b0000	0x8c000	0x7650
[3]	0xfffffec36cb24a000	0xfffffec36cb24a000	0xfffffec36cb030000	0x2a6000	0x1821c
[4]	0xfffffec36cb31d000	0xfffffec36cb31d000	0xfffffec36cb2e0000	0x48000	0x1a88
[5]	0xfffff8074f539000	0xfffff8074f539000	0xfffff8074f4c7000	0xa4000	0x6504
[6]	0xfffff8074f605000	0xfffff8074f605000	0xfffff8074f60000	0xc000	0x1a4
[7]	0xfffff8074f651000	0xfffff8074f651000	0xfffff8074f610000	0x49000	0xa50
[8]	0xfffff8074f66a000	0xfffff8074f66a000	0xfffff8074f660000	0x201000	0xd8
[9]	0xfffff8074f876000	0xfffff8074f876000	0xfffff8074f870000	0x11000	0x27c
[10]	0xfffff8074f8a6000	0xfffff8074f8a6000	0xfffff8074f890000	0x2a000	0x144c
[11]	0xfffff8074f8d6000	0xfffff8074f8d6000	0xfffff8074f8c0000	0x60000	0x3bf4
[12]	0xfffff8074f937000	0xfffff8074f937000	0xfffff8074f930000	0x27000	0x12d8
[13]	0xfffff8074f980000	0xfffff8074f980000	0xfffff8074f960000	0x68000	0x405c
[14]	0xfffff8074f9da000	0xfffff8074f9da000	0xfffff8074f9d0000	0x1a000	0x444
[15]	0xfffff8074f9f6000	0xfffff8074f9f6000	0xfffff8074f9f0000	0xb000	0xf0
[16]	0xfffff8074fda5000	0xfffff8074fda5000	0xfffff8074fd00000	0x105000	0x4524
[17]	0xfffff8074fe3c000	0xfffff8074fe3c000	0xfffff8074fe10000	0x71000	0x456c
[18]	0xfffff8074fe94000	0xfffff8074fe94000	0xfffff8074fe90000	0xe000	0x108
[19]	0xfffff8074fea0000	0xfffff8074fea0000	0xfffff8074fea0000	0xc000	0xb4
[20]	0xfffff8074fee8000	0xfffff8074fee8000	0xfffff8074feb0000	0xdd000	0x5ebc
[21]	0xfffff8075003c000	0xfffff8075003c000	0xfffff8074ff90000	0xbc000	0x4824
[22]	0xfffff8075005b000	0xfffff8075005b000	0xfffff80750050000	0x13000	0x780
[23]	0xfffff80750077000	0xfffff80750077000	0xfffff80750070000	0xf000	0x2c4
[24]	0xfffff80750087000	0xfffff80750087000	0xfffff80750080000	0x10000	0x270
[25]	0xfffff807500b5000	0xfffff807500b5000	0xfffff807500a0000	0x25000	0xfa8
[26]	0xfffff807500d9000	0xfffff807500d9000	0xfffff807500d0000	0x1a000	0x330
[27]	0xfffff807500f4000	0xfffff807500f4000	0xfffff807500f0000	0xc000	0x114
[28]	0xfffff807501b0000	0xfffff807501b0000	0xfffff80750100000	0xd5000	0xa974
[29]	0xfffff807501ef000	0xfffff807501ef000	0xfffff807501e0000	0x42000	0x102c

Alternatively, we can use the Take() method, which receives a number and prints that amount of elements from a collection, and get the same result:

```
dx -g @$inverted->TableEntry->Take(@$inverted->CurrentSize)
```

We can also do the same thing to see the UserInvertedFunctionTable (right after we switch to user that's not System), starting from nt!KeUserInvertedFunctionTable:

```
dx @$inverted =
*(nt!_INVERTED_FUNCTION_TABLE*)&nt!KeUserInvertedFunctionTable
@$inverted =
*(nt!_INVERTED_FUNCTION_TABLE*)&nt!KeUserInvertedFunctionTable
: 0x7ffa19e3a4d0 [Type: _INVERTED_FUNCTION_TABLE *]
    [+0x000] CurrentSize      : 0x2 [Type: unsigned long]
    [+0x004] MaximumSize     : 0x200 [Type: unsigned long]
    [+0x008] Epoch           : 0x6 [Type: unsigned long]
    [+0x00c] Overflow        : 0x0 [Type: unsigned char]
    [+0x010] TableEntry      [Type: _INVERTED_FUNCTION_TABLE_ENTRY [256]]
dx -g @$inverted->TableEntry->Take(@$inverted->CurrentSize)
```

	(+) FunctionTable	(+) DynamicTable	ImageBase	SizeOfImage	SizeOfTable
[0]	0x7ffa19e2b000	0x7ffa19e2b000	0x7ffa19cc0000	0x1f0000	0xe0a0
[1]	0x7ff6dea33000	0x7ff6dea33000	0x7ff6dea10000	0x26000	0xe94

And of course we can use Select() , Where() or other functions to filter, sort or select only specific fields for our output and get tailored results that fit exactly what we need.

The next thing to handle is lists — Windows is full of linked lists, you can find them everywhere. Linking processes, threads, modules, DPCs, IRPs...

Fortunately the new data model has a very useful Debugger method - `Debugger.Utility.Collections.FromListEntry`, which takes in a linked list start point, type and name of the field in this type containing the `LIST_ENTRY` structure, and will return a container of all the list contents.

So for our example let's dump all the handle tables in the system. Our starting point will be the symbol `nt!HandleTableListHead`, the type of the objects in the list is `nt!_HANDLE_TABLE` and the field linking the list is `HandleTableList`:

```
dx -r2
Debugger.Utility.Collections.FromListEntry(*(nt!_LIST_ENTRY*)&nt!HandleTableListHead, "nt!_HANDLE_TABLE", "HandleTableList")
Debugger.Utility.Collections.FromListEntry(*(nt!_LIST_ENTRY*)&nt!HandleTableListHead, "nt!_HANDLE_TABLE", "HandleTableList")
    [0x0] [Type: _HANDLE_TABLE]
        [+0x000] NextHandleNeedingPool : 0x3400 [Type: unsigned long]
        [+0x004] ExtraInfoPages : 0 [Type: long]
        [+0x008] TableCode : 0xffff8d8dcfd18001 [Type: unsigned
__int64]
        [+0x010] QuotaProcess : 0x0 [Type: _EPROCESS *]
        [+0x018] HandleTableList [Type: _LIST_ENTRY]
        [+0x028] UniqueProcessId : 0x4 [Type: unsigned long]
        [+0x02c] Flags : 0x0 [Type: unsigned long]
        [+0x02c ( 0: 0)] StrictFIFO : 0x0 [Type: unsigned char]
        [+0x02c ( 1: 1)] EnableHandleExceptions : 0x0 [Type: unsigned
char]
        [+0x02c ( 2: 2)] Rundown : 0x0 [Type: unsigned char]
        [+0x02c ( 3: 3)] Duplicated : 0x0 [Type: unsigned char]
        [+0x02c ( 4: 4)] RaiseUMExceptionOnInvalidHandleClose : 0x0 [Type:
unsigned char]
        [+0x030] HandleContentionEvent [Type: _EX_PUSH_LOCK]
        [+0x038] HandleTableLock [Type: _EX_PUSH_LOCK]
        [+0x040] FreeLists [Type: _HANDLE_TABLE_FREE_LIST [1]]
        [+0x040] ActualEntry [Type: unsigned char [32]]
        [+0x060] DebugInfo : 0x0 [Type: _HANDLE_TRACE_DEBUG_INFO *]
    [0x1] [Type: _HANDLE_TABLE]
        [+0x000] NextHandleNeedingPool : 0x400 [Type: unsigned long]
        [+0x004] ExtraInfoPages : 0 [Type: long]
        [+0x008] TableCode : 0xffff8d8dcb651000 [Type: unsigned
__int64]
        [+0x010] QuotaProcess : 0xffffb90a530e4080 [Type: _EPROCESS *]
        [+0x018] HandleTableList [Type: _LIST_ENTRY]
        [+0x028] UniqueProcessId : 0x78 [Type: unsigned long]
        [+0x02c] Flags : 0x10 [Type: unsigned long]
        [+0x02c ( 0: 0)] StrictFIFO : 0x0 [Type: unsigned char]
        [+0x02c ( 1: 1)] EnableHandleExceptions : 0x0 [Type: unsigned
char]
        [+0x02c ( 2: 2)] Rundown : 0x0 [Type: unsigned char]
        [+0x02c ( 3: 3)] Duplicated : 0x0 [Type: unsigned char]
        [+0x02c ( 4: 4)] RaiseUMExceptionOnInvalidHandleClose : 0x1 [Type:
unsigned char]
        [+0x030] HandleContentionEvent [Type: _EX_PUSH_LOCK]
        [+0x038] HandleTableLock [Type: _EX_PUSH_LOCK]
```

```

[+0x040] FreeLists           [Type: _HANDLE_TABLE_FREE_LIST [1]]
[+0x040] ActualEntry         [Type: unsigned char [32]]
[+0x060] DebugInfo           : 0x0 [Type: _HANDLE_TRACE_DEBUG_INFO *]
...

```

See the QuotaProcess field? That field points to the process that this handle table belongs to. Since every process has a handle table, this allows us to enumerate all the processes on the system in a way that's not widely known. This method has been used by rootkits in the past to enumerate processes without being detected by EDR products. So to implement this we just need to Select() the QuotaProcess from each entry in our handle table list. To create a nicer looking output we can also create an anonymous container with the process name, PID and EPROCESS pointer:

```

dx -r2
(Debugger.Utility.Collections.FromListEntry(*(nt!_LIST_ENTRY*)&nt!HandleTableListHead, "nt!_HANDLE_TABLE", "HandleTableList")).Select(h => new {
Object = h.QuotaProcess, Name = (char*)h.QuotaProcess->ImageFileName, PID = (__int64)h.QuotaProcess->UniqueProcessId})
(Debugger.Utility.Collections.FromListEntry(*(nt!_LIST_ENTRY*)&nt!HandleTableListHead, "nt!_HANDLE_TABLE", "HandleTableList")).Select(h => new {
Object = h.QuotaProcess, Name = (char*)h.QuotaProcess->ImageFileName, PID = (__int64)h.QuotaProcess->UniqueProcessId})
[0x0]           : Unable to read memory at Address 0x2e8
[0x1]
Object          : 0xfffffb90a530e4080 [Type: _EPROCESS *]
Name            : 0xfffffb90a530e44d0 : "Registry" [Type: char *]
PID             : 120 [Type: __int64]
[0x2]
Object          : 0xfffffb90a574c9400 [Type: _EPROCESS *]
Name            : 0xfffffb90a574c9850 : "smss.exe" [Type: char *]
PID             : 472 [Type: __int64]
[0x3]
Object          : 0xfffffb90a573a7240 [Type: _EPROCESS *]
Name            : 0xfffffb90a573a7690 : "csrss.exe" [Type: char *]
PID             : 580 [Type: __int64]
[0x4]
Object          : 0xfffffb90a587a8080 [Type: _EPROCESS *]
Name            : 0xfffffb90a587a84d0 : "wininit.exe" [Type: char *]
PID             : 660 [Type: __int64]
[0x5]
Object          : 0xfffffb90a587bb080 [Type: _EPROCESS *]
Name            : 0xfffffb90a587bb4d0 : "csrss.exe" [Type: char *]
PID             : 672 [Type: __int64]
[0x6]
Object          : 0xfffffb90a59407080 [Type: _EPROCESS *]
Name            : 0xfffffb90a594074d0 : "winlogon.exe" [Type: char *]
PID             : 760 [Type: __int64]
...

```

The first result is the table belonging to the System process and it does not have a QuotaProcess, which is the reason this query returns an error for it. But it should work

perfectly for every other entry in the array. If we want to make our output prettier, we can filter out entries where `QuotaProcess == 0` before we do the `Select()`:

```
dx -r2
(Debugger.Utility.Collections.FromListEntry(*(nt!_LIST_ENTRY*)&nt!HandleTableListHead, "nt!_HANDLE_TABLE", "HandleTableList")).Where(h =>
h.QuotaProcess != 0).Select(h => new { Object = h.QuotaProcess, Name =
(char*)h.QuotaProcess->ImageFileName, PID = h.QuotaProcess->UniqueProcessId})
```

As we already showed before, we can also print this list in a graphic view or use any LINQ queries to make the output match our needs.

This is the end of our first part, but don't worry, the second part is right here, and it contains all the fancy new dx methods such as a new disassembler, defining our own methods, conditional breakpoints that actually work, and more.

## WinDbg — the Fun Way: Part 2

Welcome to part 2 of me trying to make you enjoy debugging on Windows (wow, I'm a nerd)!

In the first part we got to know the basics of the new debugger data model — Using the new objects, having custom registers, searching and filtering output, declaring anonymous types and parsing lists and arrays. In this part we will learn how to use legacy commands with `dx`, get to know the amazing new disassembler, create synthetic methods and types, see the fancy changes to breakpoints and use the filesystem from within the debugger.

This sounds like a lot. Because it is. So let's start!

### Legacy Commands

This new data model completely changes the debugging experience. But sometimes you do need to use one of the old commands or extensions that we all got used to, and that don't have a matching functionality under `dx`.

But we can still use these under `dx` with `Debugger.Utility.Control.ExecuteCommand`, which lets us run a legacy command as part of a `dx` query. For example, we can use the legacy `u` command to unassemble the address that is pointed to by RIP in our second stack frame.

Since `dx` output is decimal by default and legacy commands only take hex input we first need to convert it to hex using `ToDisplayString("x")`:

```
dx Debugger.Utility.Control.ExecuteCommand("u " +
@$curstack.Frames[1].Attributes.InstructionOffset.ToDisplayString("x"))
Debugger.Utility.Control.ExecuteCommand("u " +
@$curstack.Frames[1].Attributes.InstructionOffset.ToDisplayString("x"))
[0x0]      : kdnic!TXTransmitQueuedSends+0x125:
[0x1]      : fffff807`52ad2b61 4883c430      add     rsp,30h
[0x2]      : fffff807`52ad2b65 5b          pop     rbx
[0x3]      : fffff807`52ad2b66 c3          ret
[0x4]      : fffff807`52ad2b67 4c8d4370    lea     r8,[rbx+70h]
[0x5]      : fffff807`52ad2b6b 488bd7      mov     rdx,rdi
[0x6]      : fffff807`52ad2b6e 488d4b60    lea     rcx,[rbx+60h]
[0x7]      : fffff807`52ad2b72 4c8b15d7350000 mov     r10,qword ptr
[kdnic!_imp_ExInterlockedInsertTailList (fffff807`52ad6150)]
[0x8]      : fffff807`52ad2b79 e8123af8fb  call   nt!ExInterlockedInsertTailList (fffff807`4ea56590)
```

Another useful legacy command is `!irp`. This command supplies us with a lot of information about IRPs, so no need to work hard to recreate it with `dx`.

So we will try to run `!irp` for all IRPs in `lsass.exe` process. Let's walk through that:



First, we need to find the process container for lsass.exe. We already know how to do that using Where(). Then we'll pick the first process returned. Usually there should only be one lsass anyway, unless there are server silos on the machine:

```
dx @$lsass = @$cursession.Processes.Where(p => p.Name ==
"lsass.exe").First()
```

Then we need to iterate over IrpList for each thread in the process and get the IRPs themselves. We can easily do that with FromListEntry() that we've seen already. Then we only pick the threads that have IRPs in their list:

```
dx -r4 @$irpThreads = @$lsass.Threads.Select(t => new {irp =
Debugger.Utility.Collections.FromListEntry(t.KernelObject.IrpList,
"nt!_IRP", "ThreadListEntry"))}.Where(t => t.irp.Count() != 0)
@$irpThreads = @$lsass.Threads.Select(t => new {irp =
Debugger.Utility.Collections.FromListEntry(t.KernelObject.IrpList,
"nt!_IRP", "ThreadListEntry"))}.Where(t => t.irp.Count() != 0)
    [0x384]
        irp
            [0x0] [Type: _IRP]
                [<Raw View>] [Type: _IRP]
                    IoStack : Size = 12, Current
IRP_MJ_DIRECTORY_CONTROL / 0x2 for Device for "\FileSystem\Ntfs"
                    CurrentThread : 0xfffffb90a59477080 [Type: _ETHREAD *]
            [0x1] [Type: _IRP]
                [<Raw View>] [Type: _IRP]
                    IoStack : Size = 12, Current
IRP_MJ_DIRECTORY_CONTROL / 0x2 for Device for "\FileSystem\Ntfs"
                    CurrentThread : 0xfffffb90a59477080 [Type: _ETHREAD *]
```

We can stop here for a moment, click on IoStack for one of the IRPs (or run with -r5 to see all of them) and get the stack in a nice container we can work with:

```
dx @$irpThreads.First().irp[0].IoStack
@$irpThreads.First().irp[0].IoStack : Size = 12, Current
IRP_MJ_DIRECTORY_CONTROL / 0x2 for Device for "\FileSystem\Ntfs"
    [0] : IRP_MJ_CREATE / 0x0 for {...} [Type:
_IO_STACK_LOCATION]
    [1] : IRP_MJ_CREATE / 0x0 for {...} [Type:
_IO_STACK_LOCATION]
    [2] : IRP_MJ_CREATE / 0x0 for {...} [Type:
_IO_STACK_LOCATION]
    [3] : IRP_MJ_CREATE / 0x0 for {...} [Type:
_IO_STACK_LOCATION]
    [4] : IRP_MJ_CREATE / 0x0 for {...} [Type:
_IO_STACK_LOCATION]
    [5] : IRP_MJ_CREATE / 0x0 for {...} [Type:
_IO_STACK_LOCATION]
    [6] : IRP_MJ_CREATE / 0x0 for {...} [Type:
_IO_STACK_LOCATION]
    [7] : IRP_MJ_CREATE / 0x0 for {...} [Type:
_IO_STACK_LOCATION]
```

```

    [8] : IRP_MJ_CREATE / 0x0 for {...} [Type:
_IO_STACK_LOCATION]
    [9] : IRP_MJ_CREATE / 0x0 for {...} [Type:
_IO_STACK_LOCATION]
    [10] : IRP_MJ_DIRECTORY_CONTROL / 0x2 for Device for
"\FileSystem\Ntfs" [Type: _IO_STACK_LOCATION]
    [11] : IRP_MJ_DIRECTORY_CONTROL / 0x2 for Device for
"\FileSystem\FltMgr" [Type: _IO_STACK_LOCATION]

```

And as the final step we will iterate over every thread, and over every IRP in them, and ExecuteCommand !irp <irp address>. Here too we need casting and ToDisplayString("x") to match the format expected by legacy commands (the output of !irp is very long so we trimmed it down to focus on the interesting data):

```

dx -r3 @$irpThreads.Select(t => t.irp.Select(i =>
Debugger.Utility.Control.ExecuteCommand("!irp " +
((__int64)&i).ToDisplayString("x"))))
@$irpThreads.Select(t => t.irp.Select(i =>
Debugger.Utility.Control.ExecuteCommand("!irp " +
((__int64)&i).ToDisplayString("x"))))
    [0x384]
        [0x0]
            [0x0] : Irp is active with 12 stacks 11 is current
(= 0xfffffb90a5b8f4d40)
            [0x1] : No Mdl: No System Buffer: Thread
fffffb90a59477080: Irp stack trace.
            [0x2] : cmd flg cl Device File
Completion-Context
            [0x3] : [N/A(0), N/A(0)]
            ...
            [0x34] : Irp Extension present at
0xfffffb90a5b8f4dd0:
            [0x1]
            [0x0] : Irp is active with 12 stacks 11 is current
(= 0xfffffb90a5bd24840)
            [0x1] : No Mdl: No System Buffer: Thread
fffffb90a59477080: Irp stack trace.
            [0x2] : cmd flg cl Device File
Completion-Context
            [0x3] : [N/A(0), N/A(0)]
            ...
            [0x34] : Irp Extension present at
0xfffffb90a5bd248d0:

```

Most of the information given to us by !irp we can get by parsing the IRPs with dx and dumping the IoStack for each. But there are a few things we might have a harder time to get but receive from the legacy command such as the existence and address of an IrpExtension and information about a possible Mdl linked to the Irp.

## Disassembler

We used the `u` command as an example, though in this case there actually is functionality implementing this in `dx`, through `Debugger.Utility.Code.CreateDisassembler` and `DisassembleBlock`, creating iterable and searchable disassembly:

```
dx -r3
Debugger.Utility.Code.CreateDisassembler().DisassembleBlocks(@$curstack.Fr
ames[1].Attributes.InstructionOffset)
Debugger.Utility.Code.CreateDisassembler().DisassembleBlocks(@$curstack.Fr
ames[1].Attributes.InstructionOffset)
    [0xffffffff80752ad2b61] : Basic Block [0xffffffff80752ad2b61 -
0xffffffff80752ad2b67)
        StartAddress      : 0xffffffff80752ad2b61
        EndAddress        : 0xffffffff80752ad2b67
        Instructions
            [0xffffffff80752ad2b61] : add            rsp,30h
            [0xffffffff80752ad2b65] : pop            rbx
            [0xffffffff80752ad2b66] : ret
        [0xffffffff80752ad2b67] : Basic Block [0xffffffff80752ad2b67 -
0xffffffff80752ad2b7e)
            StartAddress    : 0xffffffff80752ad2b67
            EndAddress      : 0xffffffff80752ad2b7e
            Instructions
                [0xffffffff80752ad2b67] : lea            r8,[rbx+70h]
                [0xffffffff80752ad2b6b] : mov            rdx,rdi
                [0xffffffff80752ad2b6e] : lea            rcx,[rbx+60h]
                [0xffffffff80752ad2b72] : mov            r10,qword ptr
[kdnic!__imp_ExInterlockedInsertTailList (fffff80752ad6150)]
                [0xffffffff80752ad2b79] : call
ntkrnlmp!ExInterlockedInsertTailList (fffff8074ea56590)
            [0xffffffff80752ad2b7e] : Basic Block [0xffffffff80752ad2b7e -
0xffffffff80752ad2b80)
                StartAddress : 0xffffffff80752ad2b7e
                EndAddress   : 0xffffffff80752ad2b80
                Instructions
                    [0xffffffff80752ad2b7e] : jmp
kdnic!TXTransmitQueuedSends+0xd0 (fffff80752ad2b0c)
            [0xffffffff80752ad2b80] : Basic Block [0xffffffff80752ad2b80 -
0xffffffff80752ad2b81)
                StartAddress : 0xffffffff80752ad2b80
                EndAddress   : 0xffffffff80752ad2b81
                Instructions
                    ...
```

And the cleaned-up version, picking only the instructions and flattening the tree:

```
dx -r2
Debugger.Utility.Code.CreateDisassembler().DisassembleBlocks(@$curstack.Fr
ames[1].Attributes.InstructionOffset).Select(b =>
b.Instructions).Flatten()
```

```

Debugger.Utility.Code.CreateDisassembler().DisassembleBlocks(@$curstack.Frames[1].Attributes.InstructionOffset).Select(b =>
b.Instructions).Flatten()
    [0x0]
        [0xffffffff80752ad2b61] : add            rsp,30h
        [0xffffffff80752ad2b65] : pop             rbx
        [0xffffffff80752ad2b66] : ret
    [0x1]
        [0xffffffff80752ad2b67] : lea            r8,[rbx+70h]
        [0xffffffff80752ad2b6b] : mov            rdx,rdi
        [0xffffffff80752ad2b6e] : lea            rcx,[rbx+60h]
        [0xffffffff80752ad2b72] : mov            r10,qword ptr
[kdnic!__imp_ExInterlockedInsertTailList (fffff80752ad6150)]
        [0xffffffff80752ad2b79] : call
ntkrnlmp!ExInterlockedInsertTailList (fffff8074ea56590)
    [0x2]
        [0xffffffff80752ad2b7e] : jmp
kdnic!TXTransmitQueuedSends+0xd0 (fffff80752ad2b0c)
    [0x3]
        [0xffffffff80752ad2b80] : int            3
    [0x4]
        [0xffffffff80752ad2b81] : int            3
    ...

```

## Synthetic Methods

Another functionality that we get with this debugger data model is to create functions of our own and use them, with this syntax:

```

0: kd> dx @$multiplyByThree = (x => x * 3)
@$multiplyByThree = (x => x * 3)
0: kd> dx @$multiplyByThree(5)
@$multiplyByThree(5) : 15

```

Or we can have functions taking multiple arguments:

```

0: kd> dx @$add = ((x, y) => x + y)
@$add = ((x, y) => x + y)
0: kd> dx @$add(5, 7)
@$add(5, 7) : 12

```

Or if we want to really go a few levels up, we can apply these functions to the disassembly output we saw earlier to find all writes into memory in ZwSetInformationProcess. For that there are a few checks we need to apply to each instruction to know whether or not it's a write into memory:

- Does it have at least 2 operands?

For example, `ret` will have zero and `jmp <address>` will have one. We only care about cases where one value is being written into some location, which will always require two operands. To verify that we will check for each instruction `Operands.Count() > 1`.

· Is this a memory reference?

We are only interested in writes into memory and want to ignore instructions like `mov r10, rcx`. To do that, we will check for each instruction its

`Operands[0].Attributes.IsMemoryReference == true`.

We check `Operands[0]` because that will be the destination. If we wanted to find memory reads we would have checked the source, which is in `Operands[1]`.

· Is the destination operand an output?

We want to filter out instructions where memory is referenced but not written into. To check that we will use `Operands[0].IsOutput == true`.

· As our last filter we want to ignore memory writes into the stack, which will look like `mov [rsp+0x18], 1` or `mov [rbp-0x10], rdx`.

We will check the register of the first operand and make sure its index is not the `rsp` index (0x14) or `rbp` index (0x15).

We will write a function, `@$isMemWrite`, that receives a block and only returns the instructions that contain a memory write, based in these checks. Then we can create a disassembler, disassemble our target function and only print the memory writes in it:

```
dx -r0 @$rspId = 0x14
dx -r0 @$rbpId = 0x15
dx -r0 @$isMemWrite = (b => b.Instructions.Where(i => i.Operands.Count() >
1 && i.Operands[0].Attributes.IsOutput && i.Operands[0].Registers[0].Id !=
@$rspId && i.Operands[0].Registers[0].Id != @$rbpId &&
i.Operands[0].Attributes.IsMemoryReference))
dx -r0 @$findMemWrite = (a =>
Debugger.Utility.Code.CreateDisassembler().DisassembleBlocks(a).Select(b
=> @$isMemWrite(b)))
dx -r2 @$findMemWrite(&nt!ZwSetInformationProcess).Where(b => b.Count() !=
0)
@$findMemWrite(&nt!ZwSetInformationProcess).Where(b => b.Count() != 0)
[0xffffffff8074ebd23d4]
[0xffffffff8074ebd23e9] : mov          qword ptr [r10+80h],rax
[0xffffffff8074ebd23f5] : mov          qword ptr [r10+44h],rax
[0xffffffff8074ebd2415]
[0xffffffff8074ebd2421] : mov          qword ptr [r10+98h],r8
[0xffffffff8074ebd2428] : mov          qword ptr [r10+0F8h],r9
[0xffffffff8074ebd2433] : mov          byte ptr gs:[5D18h],al
[0xffffffff8074ebd25b2]
[0xffffffff8074ebd25c3] : mov          qword ptr [rcx],rax
[0xffffffff8074ebd25c9] : mov          qword ptr [rcx+8],rax
[0xffffffff8074ebd25d0] : mov          qword ptr [rcx+10h],rax
[0xffffffff8074ebd25d7] : mov          qword ptr [rcx+18h],rax
[0xffffffff8074ebd25df] : mov          qword ptr [rcx+0A0h],rax
[0xffffffff8074ebd263f]
[0xffffffff8074ebd264f] : and          byte ptr [rax+5],0FDh
[0xffffffff8074ebd26da]
[0xffffffff8074ebd26e3] : mov          qword ptr [rcx],rax
[0xffffffff8074ebd26e9] : mov          qword ptr [rcx+8],rax
[0xffffffff8074ebd26f0] : mov          qword ptr [rcx+10h],rax
```

```

[0xffffffff8074ebd26f7] : mov      qword ptr [rcx+18h],rax
[0xffffffff8074ebd26ff] : mov      qword ptr [rcx+0A0h],rax
[0xffffffff8074ebd2708] : mov      word ptr [rcx+72h],ax
...

```

As another project combining almost everything mentioned here, we can try to create a version of !apc using dx. To simplify we will only look for kernel APCs. To do that, we have a few steps:

- Iterate over all the processes using @\$cursession.Processes to find the ones containing threads where KTHREAD.ApcState.KernelApcPending is set to 1.
- Make a container in the process with only the threads that have pending kernel APCs. Ignore the rest.
- For each of these threads, iterate over KTHREAD.ApcState.ApcListHead[0] (contains the kernel APCs) and gather interesting information about them. We can do that with the FromListHead() method we've seen earlier.  
To make our container as similar as possible to !apc, we will only get KernelRoutine and RundownRoutine, though in your implementation you might find there are other fields that interest you as well.
- To make the container easier to navigate, collect process name, ID and EPROCESS address, and thread ID and ETHREAD address.
- In our implementation we implemented a few helper functions:  
**@\$println** — runs the legacy command ln with the supplied address, to get information about the symbol  
**@\$extractBetween** — extract a string between two other strings, will be used for getting a substring from the output of @\$println  
**@\$printSymbol** — Sends an address to @\$println and extracts the symbol name only using @\$extractSymbol  
**@\$apcsForThread** — Finds all kernel APCs for a thread and creates a container with their KernelRoutine and RundownRoutine.

We then got all the processes that have threads with pending kernel APCs and saved it into the @\$procWithKernelApcs register, and then in a separate command got the APC information using @\$apcsForThread. We also cast the EPROCESS and ETHREAD pointers to void\* so dx doesn't print the whole structure when we print the final result.

This was our way of solving this problem, but there can be others, and yours doesn't have to be identical to ours!

The script we came up with is:

```

dx -r0 @$println = (a => Debugger.Utility.Control.ExecuteCommand("ln
"+((__int64)a).ToDisplayString("x")))
dx -r0 @$extractBetween = ((x,y,z) => x.Substring(x.IndexOf(y) + y.Length,
x.IndexOf(z) - x.IndexOf(y) - y.Length))
dx -r0 @$printSymbol = (a => @$extractBetween(@$println(a)[3], " ", "|"))
dx -r0 @$apcsForThread = (t => new {TID = t.Id, Object =
(void*)&t.KernelObject, Apcs =
Debugger.Utility.Collections.FromListEntry(*(nt!_LIST_ENTRY*)&t.KernelObj

```

```

ct.Tcb.ApcState.ApcListHead[0], "nt!_KAPC", "ApcListEntry").Select(a =>
new { Kernel = @$printSymbol(a.KernelRoutine), Rundown =
@$printSymbol(a.RundownRoutine)}}))
dx -r0 @$procWithKernelApc = @$cursession.Processes.Select(p => new {Name
= p.Name, PID = p.Id, Object = (void*)&p.KernelObject, ApcThreads =
p.Threads.Where(t => t.KernelObject.Tcb.ApcState.KernelApcPending !=
0))).Where(p => p.ApcThreads.Count() != 0)
dx -r6 @$procWithKernelApc.Select(p => new { Name = p.Name, PID = p.PID,
Object = p.Object, ApcThreads = p.ApcThreads.Select(t =>
@$apcsForThread(t))})

```

And it produces the following output:

```

dx -r6 @$procWithKernelApc.Select(p => new { Name = p.Name, PID = p.PID,
Object = p.Object, ApcThreads = p.ApcThreads.Select(t =>
@$apcsForThread(t))})
@$procWithKernelApc.Select(p => new { Name = p.Name, PID = p.PID, Object =
p.Object, ApcThreads = p.ApcThreads.Select(t => @$apcsForThread(t))})
[0x15b8]
    Name                : SearchUI.exe
    Length              : 0xc
    PID                 : 0x15b8
    Object               : 0xfffffb90a5b1300c0 [Type: void *]
    ApcThreads
        [0x159c]
            TID          : 0x159c
            Object        : 0xfffffb90a5b14f080 [Type: void *]
            Apcs
                [0x0]
                    Kernel      : nt!EmpCheckErrataList
                    Rundown     : nt!EmpCheckErrataList
        [0x1528]
            TID          : 0x1528
            Object        : 0xfffffb90a5aa6b080 [Type: void *]
            Apcs
                [0x0]
                    Kernel      : nt!EmpCheckErrataList
                    Rundown     : nt!EmpCheckErrataList
        [0x16b4]
            TID          : 0x16b4
            Object        : 0xfffffb90a59f1e080 [Type: void *]
            Apcs
                [0x0]
                    Kernel      : nt!EmpCheckErrataList
                    Rundown     : nt!EmpCheckErrataList
        [0x16a0]
            TID          : 0x16a0
            Object        : 0xfffffb90a5b141080 [Type: void *]
            Apcs
                [0x0]
                    Kernel      : nt!EmpCheckErrataList
                    Rundown     : nt!EmpCheckErrataList
        [0x16b8]

```

```

TID                : 0x16b8
Object              : 0xfffffb90a5aab20c0 [Type: void *]
Apcs
    [0x0]
        Kernel      : nt!EmpCheckErrataList
        Rundown      : nt!EmpCheckErrataList
[0x1740]
TID                : 0x1740
Object              : 0xfffffb90a5ab362c0 [Type: void *]
Apcs
    [0x0]
        Kernel      : nt!EmpCheckErrataList
        Rundown      : nt!EmpCheckErrataList
[0x1780]
TID                : 0x1780
Object              : 0xfffffb90a5b468080 [Type: void *]
Apcs
    [0x0]
        Kernel      : nt!EmpCheckErrataList
        Rundown      : nt!EmpCheckErrataList
[0x1778]
TID                : 0x1778
Object              : 0xfffffb90a5b6f7080 [Type: void *]
Apcs
    [0x0]
        Kernel      : nt!EmpCheckErrataList
        Rundown      : nt!EmpCheckErrataList
[0x17d0]
TID                : 0x17d0
Object              : 0xfffffb90a5b1e8080 [Type: void *]
Apcs
    [0x0]
        Kernel      : nt!EmpCheckErrataList
        Rundown      : nt!EmpCheckErrataList
[0x17d4]
TID                : 0x17d4
Object              : 0xfffffb90a5b32f080 [Type: void *]
Apcs
    [0x0]
        Kernel      : nt!EmpCheckErrataList
        Rundown      : nt!EmpCheckErrataList
[0x17f8]
TID                : 0x17f8
Object              : 0xfffffb90a5b32e080 [Type: void *]
Apcs
    [0x0]
        Kernel      : nt!EmpCheckErrataList
        Rundown      : nt!EmpCheckErrataList
[0xb28]
TID                : 0xb28
Object              : 0xfffffb90a5b065600 [Type: void *]
Apcs
    [0x0]
        Kernel      : nt!EmpCheckErrataList

```



```

                                Rundown          : nt!EmpCheckErrataList
[0x1850]
    TID                        : 0x1850
    Object                     : 0xfffffb90a5b6a5080 [Type: void *]
    Apcs
        [0x0]
            Kernel              : nt!EmpCheckErrataList
            Rundown             : nt!EmpCheckErrataList

```

Not as pretty as !apc, but still pretty good.

We can also print it as a table, receive information about the processes and be able to explore the APCs of each process separately:

```

dx -g @$procWithKernelApc.Select(p => new { Name = p.Name, PID = p.PID,
Object = p.Object, ApcThreads = p.ApcThreads.Select(t =>
@$apcsForThread(t))})

```

	Name	PID	Object	(+) ApcThreads
[0x15b8]	SearchUI.exe	0x15b8	0xfffffb90a5b1300c0	{...}
[0x194c]	SkypeApp.exe	0x194c	0xfffffb90a5b7300c0	{...}
[0x1960]	SkypeBackgroundHost.exe	0x1960	0xfffffb90a5b2e23c0	{...}
[0x19d4]	YourPhone.exe	0x19d4	0xfffffb90a5b22e080	{...}
[0x19e0]	backgroundTaskHost.exe	0x19e0	0xfffffb90a5b720240	{...}
[0x1a10]	backgroundTaskHost.exe	0x1a10	0xfffffb90a5b22d240	{...}

But wait, what are all these APCs with nt!EmpCheckErrataList? And why does SearchUI.exe have all of them? What does this process have to do with erratas?

The secret is that these are not actually APCs meant to call nt!EmpCheckErrataList. And no, the symbols are not wrong.

The thing we see here is happening because the compiler is being smart — when it sees a few different functions that have the same code, it makes them all point to the same piece of code, instead of duplicating this code multiple times. You might think that this is not a thing that would happen very often, but lets look at the disassembly for nt!EmpCheckErrataList (the old way this time):

#### u EmpCheckErrataList

nt!EmpCheckErrataList:

```

fffff807`4eb86010 c20000      ret      0
fffff807`4eb86013 cc          int      3
fffff807`4eb86014 cc          int      3
fffff807`4eb86015 cc          int      3
fffff807`4eb86016 cc          int      3

```

This is actually just a stub. It might be a function that has not been implemented yet (probably the case for this one) or a function that is meant to be a stub for a good reason. The function that is the real KernelRoutine/RundownRoutine of these APCs is nt!KiSchedulerApcNop, and is meant to be a stub on purpose, and has been for many years. And we can see it has the same code and points to the same address:

```

u nt!KiSchedulerApcNop
nt!EmpCheckErrataList:
fffff807`4eb86010 c20000          ret      0
fffff807`4eb86013 cc              int      3
fffff807`4eb86014 cc              int      3
fffff807`4eb86015 cc              int      3
fffff807`4eb86016 cc              int      3

```

So why do we see so many of these APCs?

When a thread is being suspended, the system creates a semaphore and queues an APC to the thread that will wait on that semaphore. The thread will be waiting until someone asks the resume it, and then the system will free the semaphore and the thread will stop waiting and will resume. The APC itself doesn't need to do much, but it must have a `KernelRoutine` and a `RundownRoutine`, so the system places a stub there. In the symbols this stub receives the name of one of the functions that have this "code", this time `nt!EmpCheckErrataList`, but it can be a different one in the next version.

Anyone interested in the suspension mechanism can look at [ReactOS](#). The code for these functions changed a bit since, and the stub function was renamed from `KiSuspendNop` to `KiSchedulerApcNop`, but the general design stayed similar.

But I got distracted, this is not what this blog was supposed to be talking about. Let's get back to WinDbg and synthetic functions:

## Synthetic Types

After covering synthetic methods, we can also add our own named types and use them to parse data where the type is not available to us.

For example, let's try to print the `PspCreateProcessNotifyRoutine` array, which holds all the registered process notify routines — function that are registered by drivers and will receive a notification whenever a process starts. But this array doesn't contain pointers to the registered routines. Instead it contains pointers to the non-documented `EX_CALLBACK_ROUTINE_BLOCK` structure.

So to parse this array, we need to make sure WinDbg knows this type — to do that we use [Synthetic Types](#). We start by creating a header file containing all the types we want to define (I used `c:\temp\header.h`). In this case it's just `EX_CALLBACK_ROUTINE_BLOCK`, that we can find in [ReactOS](#):

```

typedef struct _EX_CALLBACK_ROUTINE_BLOCK
{
    _EX_RUNDOWN_REF RundownProtect;
    void* Function;
    void* Context;
} EX_CALLBACK_ROUTINE_BLOCK, *PEX_CALLBACK_ROUTINE_BLOCK;

```

Now we can ask WinDbg to load it and add the types to the `nt` module:

```

dx
Debugger.Utility.Analysis.SyntheticTypes.ReadHeader("c:\\temp\\header.h",
"nt")
Debugger.Utility.Analysis.SyntheticTypes.ReadHeader("c:\\temp\\header.h",
"nt")
ReturnEnumsAsObjects : false
RegisterSyntheticTypeModels : false
Module : ntkrnlmp.exe
Header : header.h
Types

```

This gives us an object which lets us see all the types added to this module.

Now that we defined the type, we can use it with CreateInstance:

```

dx
Debugger.Utility.Analysis.SyntheticTypes.CreateInstance("_EX_CALLBACK_ROUTINE_BLOCK", *(__int64*)&nt!PspCreateProcessNotifyRoutine)
Debugger.Utility.Analysis.SyntheticTypes.CreateInstance("_EX_CALLBACK_ROUTINE_BLOCK", *(__int64*)&nt!PspCreateProcessNotifyRoutine)
RundownProtect [Type: _EX_RUNDOWN_REF]
Function : 0xffffffff8074cbddf50 [Type: void *]
Context : 0x6e496c4102031400 [Type: void *]

```

It's important to notice that CreateInstance only takes \_\_int64 inputs so any other type has to be cast. It's good to know this in advance because the error messages these modules return are not always easy to understand.

Now, if we look at our output, and specifically at Context, something seems weird. And actually if we try to dump Function we will see it doesn't point to any code:

```

dq 0xffffffff8074cbddf50
fffff807`4cbddf50  ????????` ???????? ????????` ????????
fffff807`4cbddf60  ????????` ???????? ????????` ????????
fffff807`4cbddf70  ????????` ???????? ????????` ????????
fffff807`4cbddf80  ????????` ???????? ????????` ????????
fffff807`4cbddf90  ????????` ???????? ????????` ????????

```

So what happened?

The problem is not our cast to EX\_CALLBACK\_ROUTINE\_BLOCK, but the address we are casting. If we dump the values in PspCreateProcessNotifyRoutine we might see what it is:

```

dx ((void**[0x40])&nt!PspCreateProcessNotifyRoutine).Where(a => a != 0)
((void**[0x40])&nt!PspCreateProcessNotifyRoutine).Where(a => a != 0)
[0] : 0xfffffb90a530504ef [Type: void *]
[1] : 0xfffffb90a532a512f [Type: void *]
[2] : 0xfffffb90a53da9d5f [Type: void *]
[3] : 0xfffffb90a53da9ccf [Type: void *]
[4] : 0xfffffb90a53e5d15f [Type: void *]
[5] : 0xfffffb90a571469ef [Type: void *]
[6] : 0xfffffb90a5714722f [Type: void *]

```

```
[7]          : 0xfffffb90a571473df [Type: void * *]
[8]          : 0xfffffb90a597d989f [Type: void * *]
```

The lower half-byte in all of these is 0xF, while we know that pointers in x64 machines are always aligned to 8 bytes, and usually to 0x10. This is because I oversimplified it earlier — these are not pointers to EX\_CALLBACK\_ROUTINE\_BLOCK, they are actually EX\_CALLBACK structures (another type that is not in the public pdb), containing an EX\_RUNDOWN\_REF. But to make this example simpler we will treat them as simple pointers that have been ORed with 0xF, since this is good enough for our purposes. If you ever choose to write a driver that will handle PspCreateProcessNotifyRoutine please do not use this hack, look into ReactOS and do things properly. ☹️

So to fix our command we just need to align the addresses to 0x10 before casting them. To do that we do:

```
<address> & 0xFFFFFFFFFFFFFFF0
```

Or the nicer version:

```
<address> & ~0xF
```

Let's use that in our command:

```
dx
Debugger.Utility.Analysis.SyntheticTypes.CreateInstance("_EX_CALLBACK_ROUTINE_BLOCK", (*(__int64*)&nt!PspCreateProcessNotifyRoutine) & ~0xf)
Debugger.Utility.Analysis.SyntheticTypes.CreateInstance("_EX_CALLBACK_ROUTINE_BLOCK", (*(__int64*)&nt!PspCreateProcessNotifyRoutine) & ~0xf)
    RundownProtect [Type: _EX_RUNDOWN_REF]
    Function       : 0xfffff8074ea7f310 [Type: void *]
    Context       : 0x0 [Type: void *]
```

This looks better. Let's check that Function actually points to a function this time:

```
In 0xfffff8074ea7f310
Browse module
Set bp breakpoint
(fffff807`4ea7f310) nt!ViCreateProcessCallback | (fffff807`4ea7f330)
nt!RtlStringCbLengthW
Exact matches:
    nt!ViCreateProcessCallback (void)
```

Looks much better! Now we can get define this cast as a synthetic method and get the function addresses for all routines in the array:

```
dx -r0 @$getCallbackRoutine = (a =>
Debugger.Utility.Analysis.SyntheticTypes.CreateInstance("_EX_CALLBACK_ROUTINE_BLOCK", (__int64)(a & ~0xf)))
dx ((void**[0x40])&nt!PspCreateProcessNotifyRoutine).Where(a => a != 0).Select(a => @$getCallbackRoutine(a).Function)
((void**[0x40])&nt!PspCreateProcessNotifyRoutine).Where(a => a != 0).Select(a => @$getCallbackRoutine(a).Function)
```

```

[0]          : 0xffffffff8074ea7f310 [Type: void *]
[1]          : 0xffffffff8074ff97220 [Type: void *]
[2]          : 0xffffffff80750a41330 [Type: void *]
[3]          : 0xffffffff8074f8ab420 [Type: void *]
[4]          : 0xffffffff8075106d9f0 [Type: void *]
[5]          : 0xffffffff807516dd930 [Type: void *]
[6]          : 0xffffffff8074ff252c0 [Type: void *]
[7]          : 0xffffffff807520b6aa0 [Type: void *]
[8]          : 0xffffffff80753a63cf0 [Type: void *]

```

But this will be more fun if we could see the symbols instead of the addresses. We already know how to get the symbols by executing the legacy command `ln`, but this time we will do it with `.printf`. First we will write a helper function `@$getsym` which will run the command `printf "%y", <address>`:

```

dx -r0 @$getsym = (x =>
Debugger.Utility.Control.ExecuteCommand(".printf\"%y\\", " +
((__int64)x).ToDisplayString("x"))[0])

```

Then we will send every function address to this method, to print the symbol:

```

dx ((void**[0x40])&nt!PspCreateProcessNotifyRoutine).Where(a => a !=
0).Select(a => @$getsym(@$getCallbackRoutine(a).Function))
((void**[0x40])&nt!PspCreateProcessNotifyRoutine).Where(a => a !=
0).Select(a => @$getsym(@$getCallbackRoutine(a).Function))
[0]          : nt!ViCreateProcessCallback (fffff807`4ea7f310)
[1]          : cng!CngCreateProcessNotifyRoutine
(fffff807`4ff97220)
[2]          : WdFilter!MpCreateProcessNotifyRoutineEx
(fffff807`50a41330)
[3]          : ksecdd!KsecCreateProcessNotifyRoutine
(fffff807`4f8ab420)
[4]          : tcpip!CreateProcessNotifyRoutineEx
(fffff807`5106d9f0)
[5]          : iorate!IoRateProcessCreateNotify
(fffff807`516dd930)
[6]          : CI!I_PEProcessNotify (fffff807`4ff252c0)
[7]          : dxgkrnl!DxgkProcessNotify (fffff807`520b6aa0)
[8]          : peauth+0x43cf0 (fffff807`53a63cf0)

```

There, much nicer!

## Breakpoints

### Conditional Breakpoint

Conditional breakpoints are a huge pain-point when debugging. And with the old MASM syntax they're almost impossible to use. I spent hours trying to get them to work the way I wanted to, but the command turns out to be so awful that I can't even understand what I was trying to do, not to mention why it doesn't filter anything or how to fix it.

Well, these days are over. We can now use dx queries for conditional breakpoints with the following syntax: `bp /w "dx query" <address>`.

For example, let's say we are trying to debug an issue involving file opens by Wow64 processes. The function `NtOpenProcess` is called all the time, but we only care about calls done by Wow64 processes, which are not the majority of processes on modern systems. So to avoid helplessly going through 100 debugger breaks until we get lucky or struggle with MASM-style conditional breakpoints, we can do this instead:

```
bp /w "@$curprocess.KernelObject.WoW64Process != 0" nt!NtOpenProcess
```

We then let the machine run, and when the breakpoint is hit we can check if it worked:

```
Breakpoint 3 hit
nt!NtOpenProcess:
fffff807`2e96b7e0 4883ec38      sub     rsp,38h
dx @$curprocess.KernelObject.WoW64Process
@$curprocess.KernelObject.WoW64Process      :
0xfffffc10f5163b390 [Type: _EWOV64PROCESS *]
    [+0x000] Peb           : 0xf88000 [Type: void *]
    [+0x008] Machine       : 0x14c [Type: unsigned short]
    [+0x00c] NtdllType     : PsWowX86SystemDll (1) [Type:
_SYSTEM_DLL_TYPE]
dx @$curprocess.Name
@$curprocess.Name : IpOverUsbSvc.exe
    Length         : 0x10
```

The process that triggered our breakpoint is a WoW64 process!

For anyone who has ever tried using conditional breakpoints with MASM, this is a life-changing addition.

## Other Breakpoint Options

There are a few other interesting breakpoint options found under `Debugger.Utility.Control`:

- `SetBreakpointAtSourceLocation` — allowing us to set a breakpoint in a module whose source file is available to us, with this syntax: `dx Debugger.Utility.Control.SetBreakpointAtSourceLocation("MyModule!myfile.cpp", "172")`
- `SetBreakpointAtOffset` — sets a breakpoint at an offset inside a function — `dx Debugger.Utility.Control.SetBreakpointAtOffset("NtOpenFile", 8, "nt")`
- `SetBreakpointForReadWriteFile` — similar to the legacy `ba` command but with more readable syntax, this lets us set a breakpoint to issue a debug break whenever anyone reads or writes to an address. It has default configuration of `type = Hardware Write` and `size = 1`.  
For example, let's try to break on every read of `Ci!g_CiOptions`, a variable whose size is 4 bytes:

```
dx Debugger.Utility.Control.SetBreakpointForReadWrite(&Ci!g_CiOptions,  
"Hardware Read", 0x4)
```

We let the machine keep running and almost immediately our breakpoint is hit:

```
0: kd> g  
Breakpoint 0 hit  
CI!CiValidateImageHeader+0x51b:  
ffffff807`2f6fcb1b 740c je CI!CiValidateImageHeader+0x529  
(ffffff807`2f6fcb29)
```

CI!CiValidateImageHeader read this global variable when validating an image header. In this specific example, we will see reads of this variable very often and writes into it are the more interesting case, as it can show us an attempt to tamper with signature validation.

An interesting thing to notice about these commands is that they don't just set a breakpoint, they actually return it as an object we can control, which has attributes like `IsEnabled`, `Condition` (allowing us to set a condition), `PassCount` (telling us how many times this breakpoint has been hit) and more.

## FileSystem

Under `Debugger.Utility` we have the `FileSystem` module, letting us query and control the file system on the host machine (*not* the machine we are debugging) from within the debugger:

### **dx -r1 Debugger.Utility.FileSystem**

```
Debugger.Utility.FileSystem  
CreateFile [CreateFile(path, [disposition]) - Creates a file at the  
specified path and returns a file object. 'disposition' can be one of  
'CreateAlways' or 'CreateNew']  
CreateTempFile [CreateTempFile() - Creates a temporary file in the  
%TEMP% folder and returns a file object]  
CreateTextReader [CreateTextReader(file | path, [encoding]) - Creates a  
text reader over the specified file. If a path is passed instead of a  
file, a file is opened at the specified path. 'encoding' can be 'Utf16',  
'Utf8', or 'Ascii'. 'Ascii' is the default]  
CreateTextWriter [CreateTextWriter(file | path, [encoding]) - Creates a  
text writer over the specified file. If a path is passed instead of a  
file, a file is created at the specified path. 'encoding' can be 'Utf16',  
'Utf8', or 'Ascii'. 'Ascii' is the default]  
CurrentDirectory : C:\WINDOWS\system32  
DeleteFile [DeleteFile(path) - Deletes a file at the specified path]  
FileExists [FileExists(path) - Checks for the existence of a file at  
the specified path]  
OpenFile [OpenFile(path) - Opens a file read/write at the  
specified path]  
TempDirectory : C:\Users\yshafir\AppData\Local\Temp
```

We can create files, open them, write into them, delete them or check if a file exists in a certain path. To see a simple example, let's dump the contents of our current directory — C:\Windows\System32:

### **dx -r1 Debugger.Utility.FileSystem.CurrentDirectory.Files**

```
Debugger.Utility.FileSystem.CurrentDirectory.Files
[0x0] : C:\WINDOWS\system32\07409496-a423-4a3e-b620-2cfb01a9318d_HyperV-ComputeNetwork.dll
[0x1] : C:\WINDOWS\system32\1
[0x2] : C:\WINDOWS\system32\103
[0x3] : C:\WINDOWS\system32\108
[0x4] : C:\WINDOWS\system32\11
[0x5] : C:\WINDOWS\system32\113
...
[0x44] : C:\WINDOWS\system32\93
[0x45] : C:\WINDOWS\system32\98
[0x46] : C:\WINDOWS\system32\@AppHelpToast.png
[0x47] : C:\WINDOWS\system32\@AudioToastIcon.png
[0x48] : C:\WINDOWS\system32\@BackgroundAccessToastIcon.png
[0x49] : C:\WINDOWS\system32\@bitlockertoastimage.png
[0x4a] : C:\WINDOWS\system32\@edpttoastimage.png
[0x4b] : C:\WINDOWS\system32\@EnrollmentToastIcon.png
[0x4c] : C:\WINDOWS\system32\@language_notification_icon.png
[0x4d] : C:\WINDOWS\system32\@optionalfeatures.png
[0x4e] : C:\WINDOWS\system32\@VpnToastIcon.png
[0x4f] : C:\WINDOWS\system32\@WiFiNotificationIcon.png
[0x50] : C:\WINDOWS\system32\@windows-hello-V4.1.gif
[0x51] : C:\WINDOWS\system32\@WindowsHelloFaceToastIcon.png
[0x52] :
C:\WINDOWS\system32\@WindowsUpdateToastIcon.contrast-black.png
[0x53] :
C:\WINDOWS\system32\@WindowsUpdateToastIcon.contrast-white.png
[0x54] : C:\WINDOWS\system32\@WindowsUpdateToastIcon.png
[0x55] : C:\WINDOWS\system32\@WirelessDisplayToast.png
[0x56] : C:\WINDOWS\system32\@WwanNotificationIcon.png
[0x57] : C:\WINDOWS\system32\@WwanSimLockIcon.png
[0x58] : C:\WINDOWS\system32\aadauthhelper.dll
[0x59] : C:\WINDOWS\system32\aadcloudap.dll
[0x5a] : C:\WINDOWS\system32\aadjcsp.dll
[0x5b] : C:\WINDOWS\system32\aadtb.dll
[0x5c] : C:\WINDOWS\system32\aadWamExtension.dll
[0x5d] : C:\WINDOWS\system32\AboutSettingsHandlers.dll
[0x5e] : C:\WINDOWS\system32\AboveLockAppHost.dll
[0x5f] : C:\WINDOWS\system32\accessibilitycpl.dll
[0x60] : C:\WINDOWS\system32\accountaccessor.dll
[0x61] : C:\WINDOWS\system32\AccountsRt.dll
[0x62] : C:\WINDOWS\system32\AcGenral.dll
...
```

We can choose to delete one of these files:

```
dx -r1 Debugger.Utility.FileSystem.CurrentDirectory.Files[1].Delete()
```



Or delete it through DeleteFile:

```
dx Debugger.Utility.FileSystem.DeleteFile("C:\\WINDOWS\\system32\\71")
```

Notice that in this module paths have to have double backslash ("\\"), as they would if we had called the Win32 API ourselves.

As a last exercise we'll put together a few of the things we learned here — we're going to create a breakpoint on a kernel variable, get the symbol that accessed it from the stack and write the symbol the accessed it into a file on our host machine.

Let's break it down into steps:

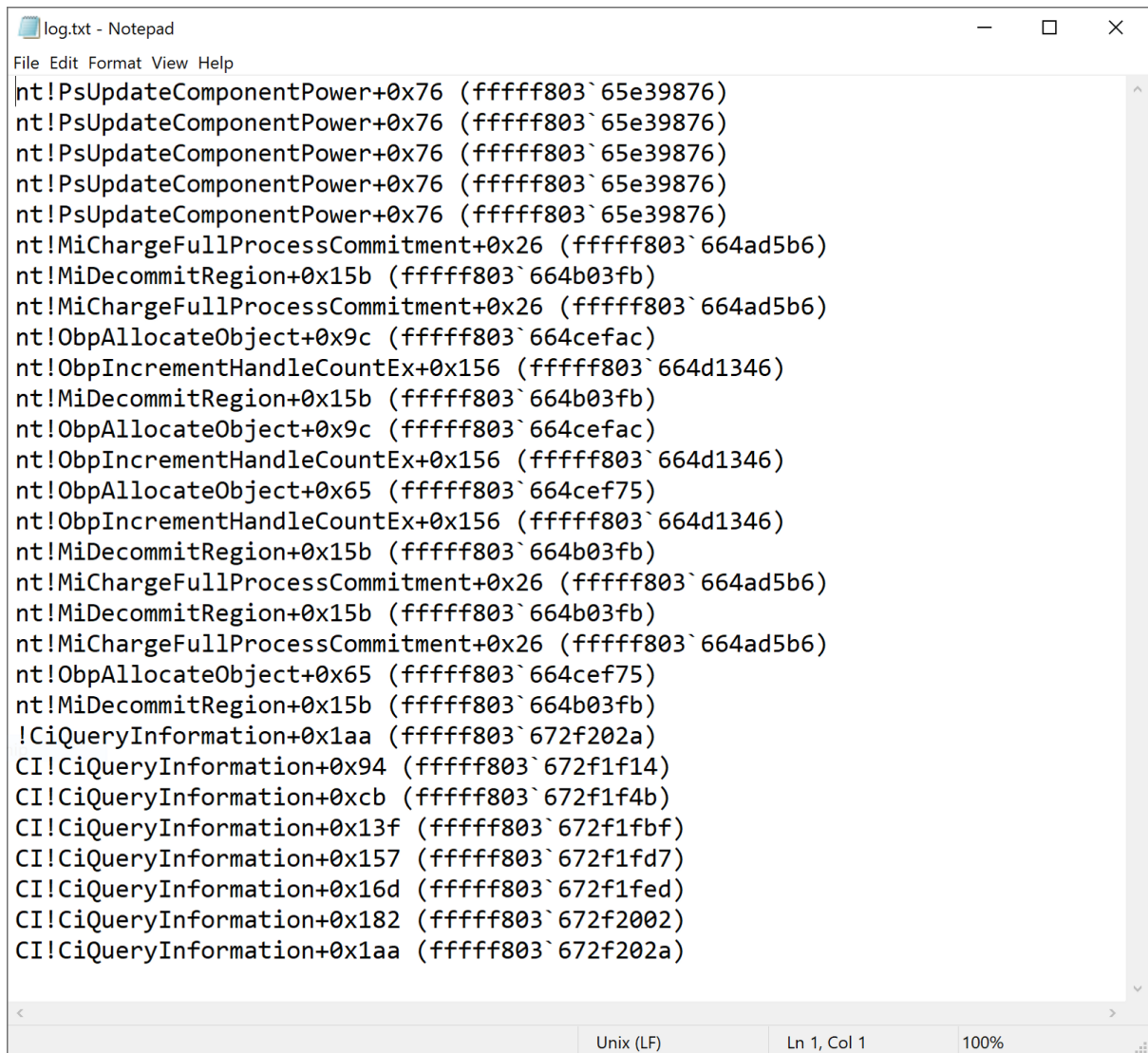
- Open a file to write the results to.
- Create a text writer, which we will use to write into the file.
- Create a breakpoint for access into a variable. In this case we'll choose `nt!PsInitialSystemProcess` and set a breakpoint for read access. We will use the old MASM syntax to run a dx command every time the breakpoint is hit and move on: `ba r4 <address> "dx <command>; g"`  
Our command will use `@$curstack` to get the address that accessed the variable, and then use the `@$getsym` helper function we wrote earlier to find the symbol for it. We'll use our text writer to write the result into the file.
- Finally, we will close the file.

Putting it all together:

```
dx -r0 @$getsym = (x =>
Debugger.Utility.Control.ExecuteCommand(".printf\"%y\\", " +
((__int64)x).ToDisplayString("x"))[0])
dx @$tmpFile =
Debugger.Utility.FileSystem.TempDirectory.OpenFile("log.txt")
dx @$txtWriter = Debugger.Utility.FileSystem.CreateTextWriter(@$tmpFile)
ba r4 nt!PsInitialSystemProcess "dx
@$txtWriter.WriteLine(@$getsym(@$curstack.Frames[0].Attributes.Instruction
Offset)); g"
```

We let the machine run for as long as we want, and when we want to stop the logging we can disable or clear the breakpoint and close the file with `dx @$tmpFile.Close()`.

Now we can open our `@$tmpFile` and look at the results:



```
log.txt - Notepad
File Edit Format View Help
nt!PsUpdateComponentPower+0x76 (fffff803`65e39876)
nt!PsUpdateComponentPower+0x76 (fffff803`65e39876)
nt!PsUpdateComponentPower+0x76 (fffff803`65e39876)
nt!PsUpdateComponentPower+0x76 (fffff803`65e39876)
nt!PsUpdateComponentPower+0x76 (fffff803`65e39876)
nt!MiChargeFullProcessCommitment+0x26 (fffff803`664ad5b6)
nt!MiDecommitRegion+0x15b (fffff803`664b03fb)
nt!MiChargeFullProcessCommitment+0x26 (fffff803`664ad5b6)
nt!ObpAllocateObject+0x9c (fffff803`664cefac)
nt!ObpIncrementHandleCountEx+0x156 (fffff803`664d1346)
nt!MiDecommitRegion+0x15b (fffff803`664b03fb)
nt!ObpAllocateObject+0x9c (fffff803`664cefac)
nt!ObpIncrementHandleCountEx+0x156 (fffff803`664d1346)
nt!ObpAllocateObject+0x65 (fffff803`664cef75)
nt!ObpIncrementHandleCountEx+0x156 (fffff803`664d1346)
nt!MiDecommitRegion+0x15b (fffff803`664b03fb)
nt!MiChargeFullProcessCommitment+0x26 (fffff803`664ad5b6)
nt!MiDecommitRegion+0x15b (fffff803`664b03fb)
nt!MiChargeFullProcessCommitment+0x26 (fffff803`664ad5b6)
nt!ObpAllocateObject+0x65 (fffff803`664cef75)
nt!MiDecommitRegion+0x15b (fffff803`664b03fb)
!CiQueryInformation+0x1aa (fffff803`672f202a)
CI!CiQueryInformation+0x94 (fffff803`672f1f14)
CI!CiQueryInformation+0xcb (fffff803`672f1f4b)
CI!CiQueryInformation+0x13f (fffff803`672f1fbf)
CI!CiQueryInformation+0x157 (fffff803`672f1fd7)
CI!CiQueryInformation+0x16d (fffff803`672f1fed)
CI!CiQueryInformation+0x182 (fffff803`672f2002)
CI!CiQueryInformation+0x1aa (fffff803`672f202a)
Unix (LF) Ln 1, Col 1 100%
```

That's it! What an amazingly easy way to log information about the debugger!

So that's the end of our WinDbg series! All the scripts in this series will be uploaded to a github repo, as well as some new ones not included here. I suggest you investigate this data model further, because we didn't even cover all the different methods it contains. Write cool tools of your own and share them with the world :)

And as long as this guide was, these are not even all the possible options in the new data model. And I didn't even mention the new support for Javascript! You can get more information about using Javascript in WinDbg and the new and exciting support for TTD (time travel debugging) in this excellent [post](#).