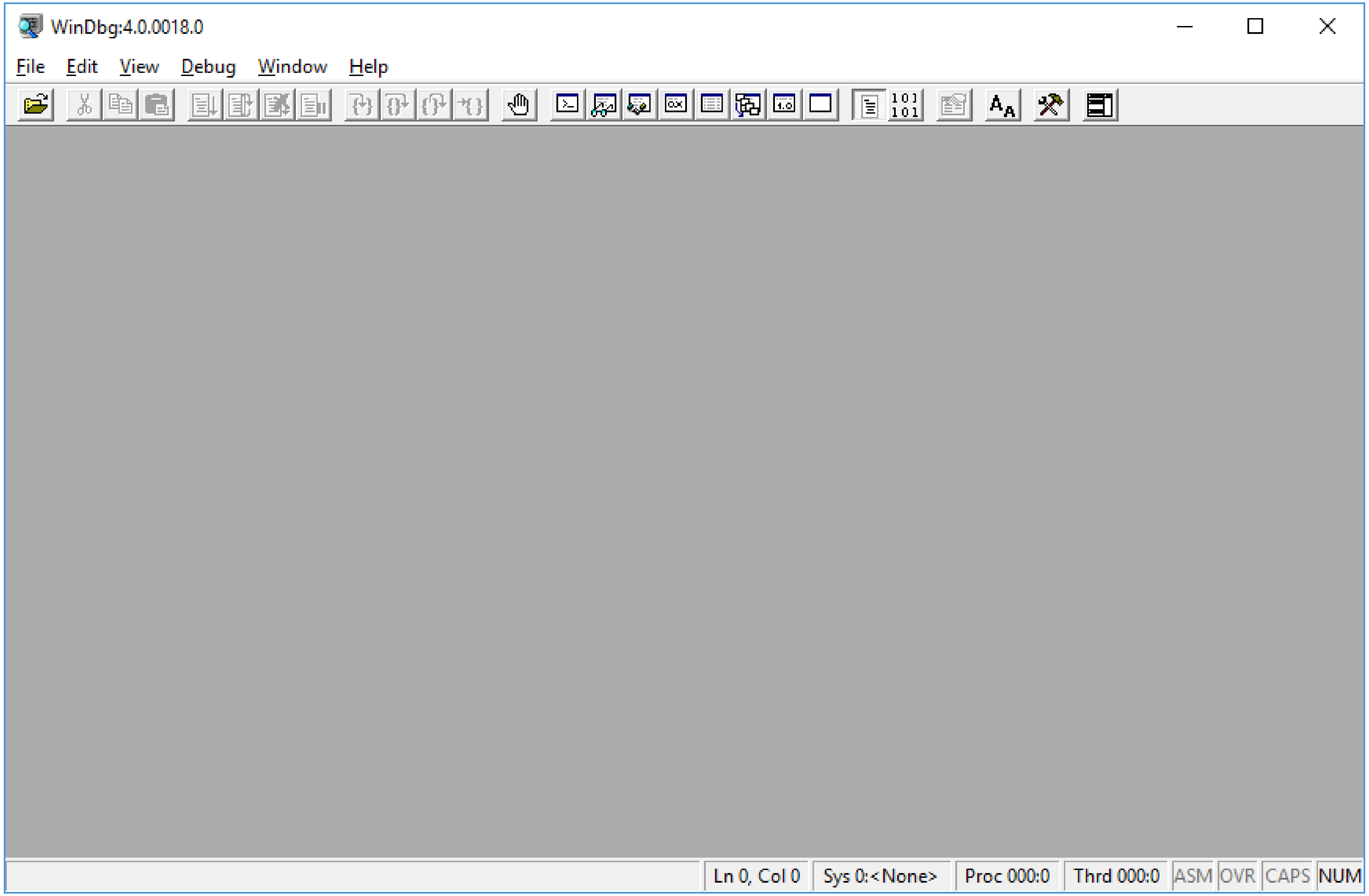
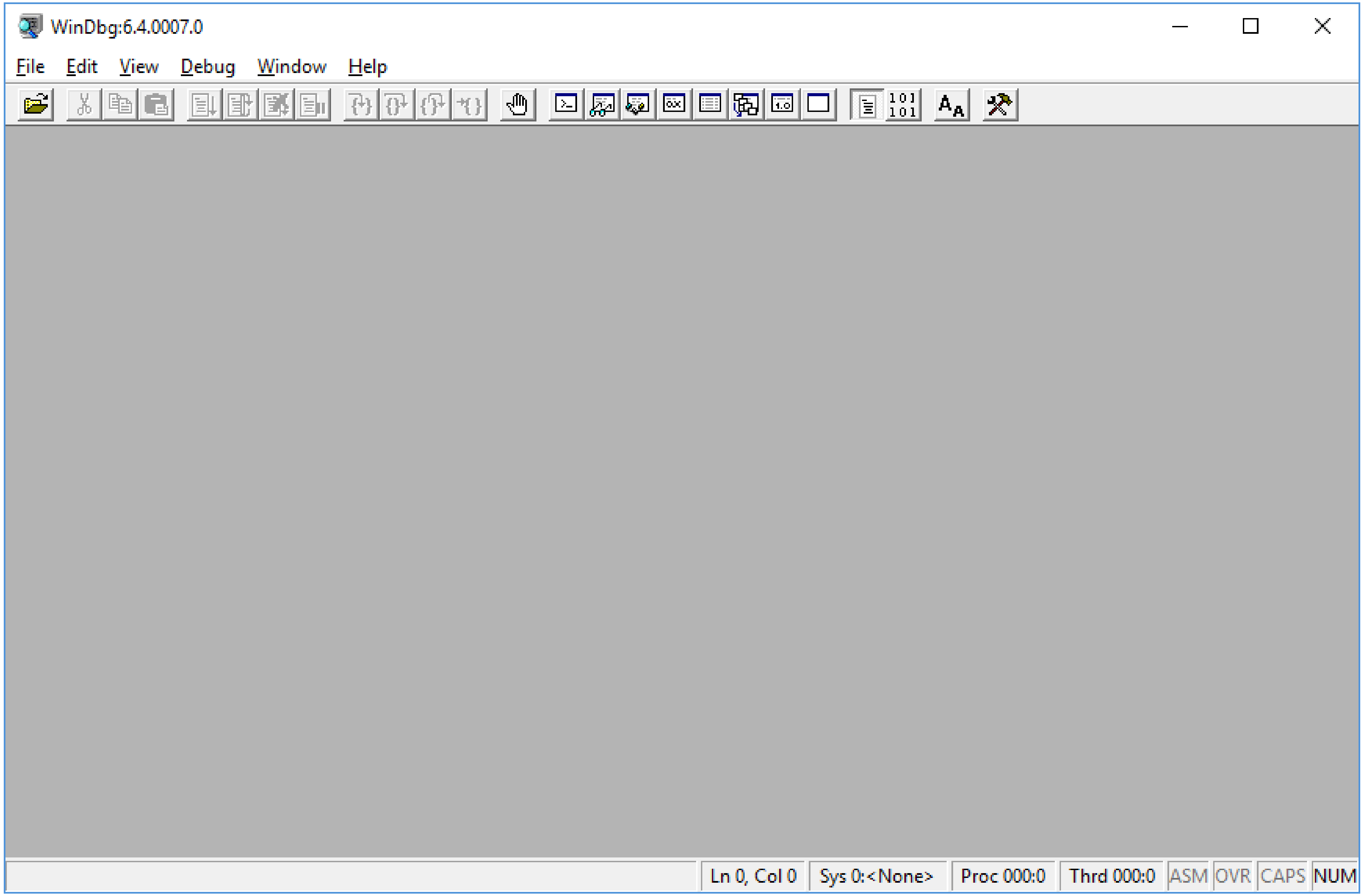
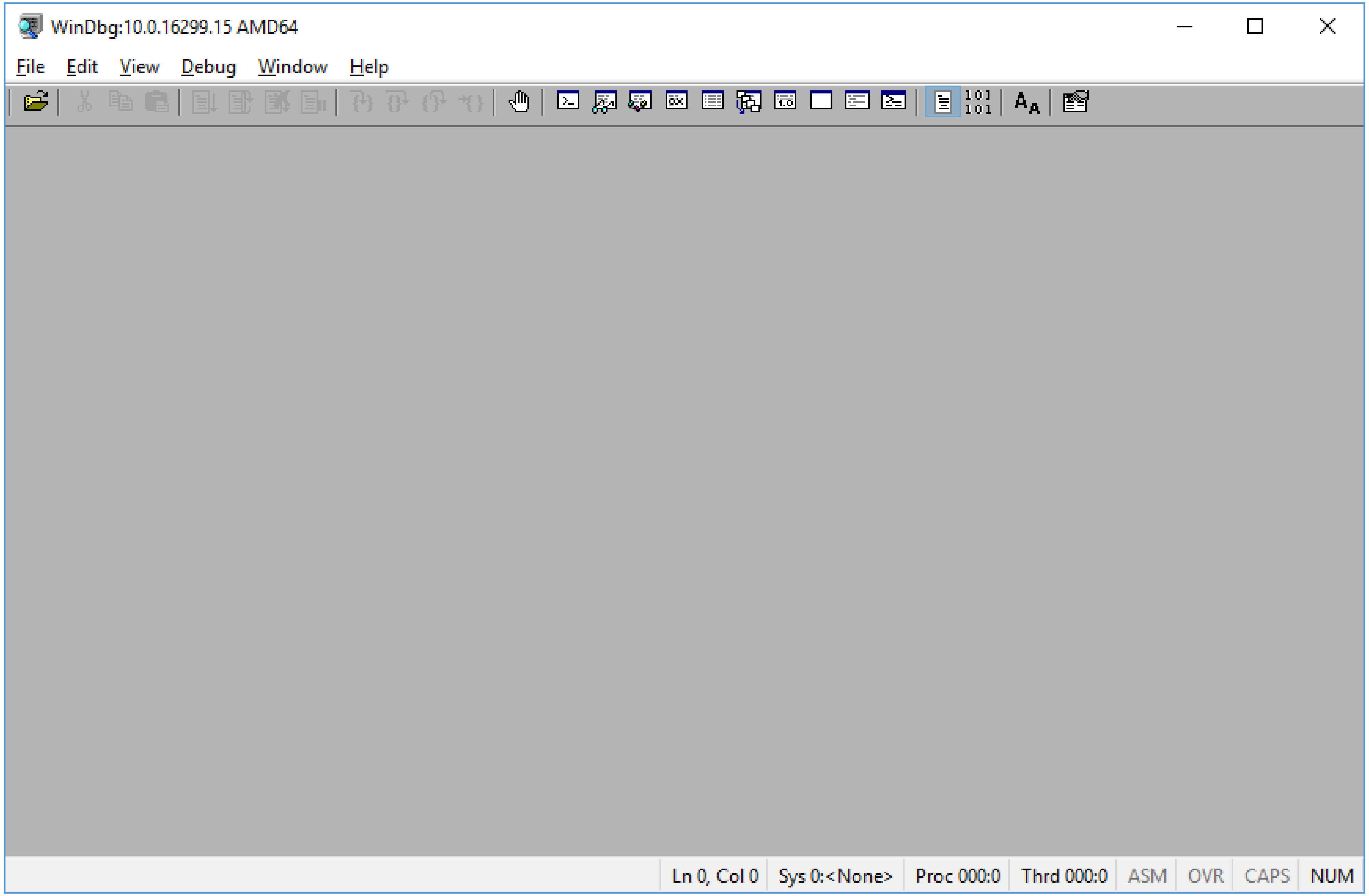
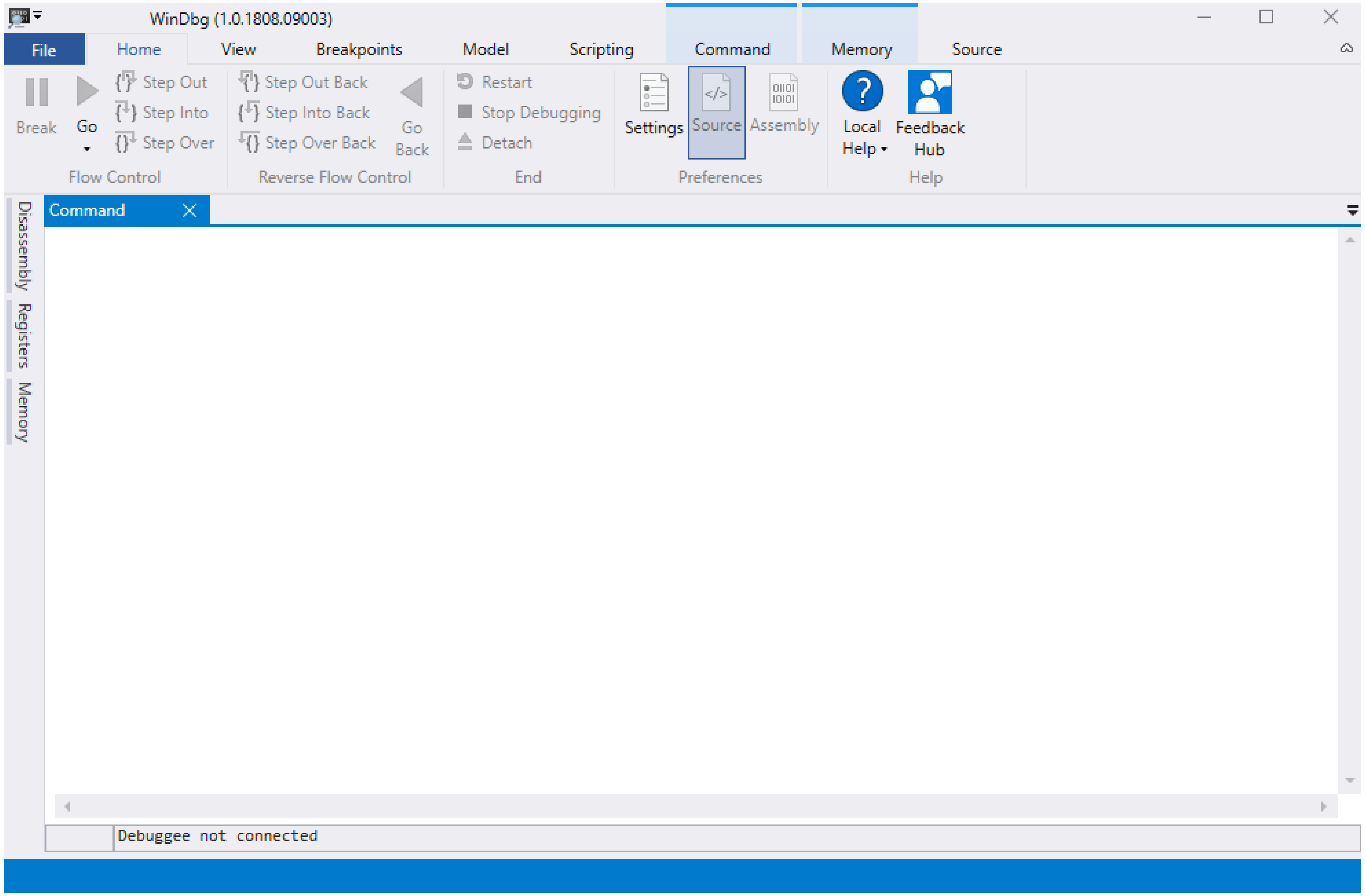
Extend the New Windbg to Build Your Own Dream Debugging Tool

Kevin Gosse @kookiz


WinDbg:4.0.0018.0
File
Edit
View
Debug
Window
Help
Ln 0, Col 0
Sys 0:<None>
Proc 000:0
Thrd 000:0
ASM
OVR
CAPS
NUM


WinDbg:6.4.0007.0
File
Edit
View
Debug
Window
Help
Ln 0, Col 0
Sys 0:<None>
Proc 000:0
Thrd 000:0
ASM
OVR
CAPS
NUM


WinDbg:10.0.16299.15 AMD64
File
Edit
View
Debug
Window
Help
Ln 0, Col 0
Sys 0:<None>
Proc 000:0
Thrd 000:0
ASM
OVR
CAPS
NUM



```
Debuggee not connected
```

A new WinDBG, with the same flaws

Shiny new UI

New features for native debugging (time travel debugging)

Single window editor

No history

Cryptic scripting

How many times have you typed .loadby sos clr?

But something has changed!

As a rule of thumb, if an application has a ribbon, it is very likely written in .NET

Any .NET application can be hacked with reasonable effort (custom loader, replacing assemblies, editing the IL, …)

Searching for extension points

MEF in a nutshell

`[Export]` A class marked with that attribute will be added to MEF's "catalog"

`[Import]` When marking a field with this attribute, MEF will try to find a matching type in its catalog and assign it

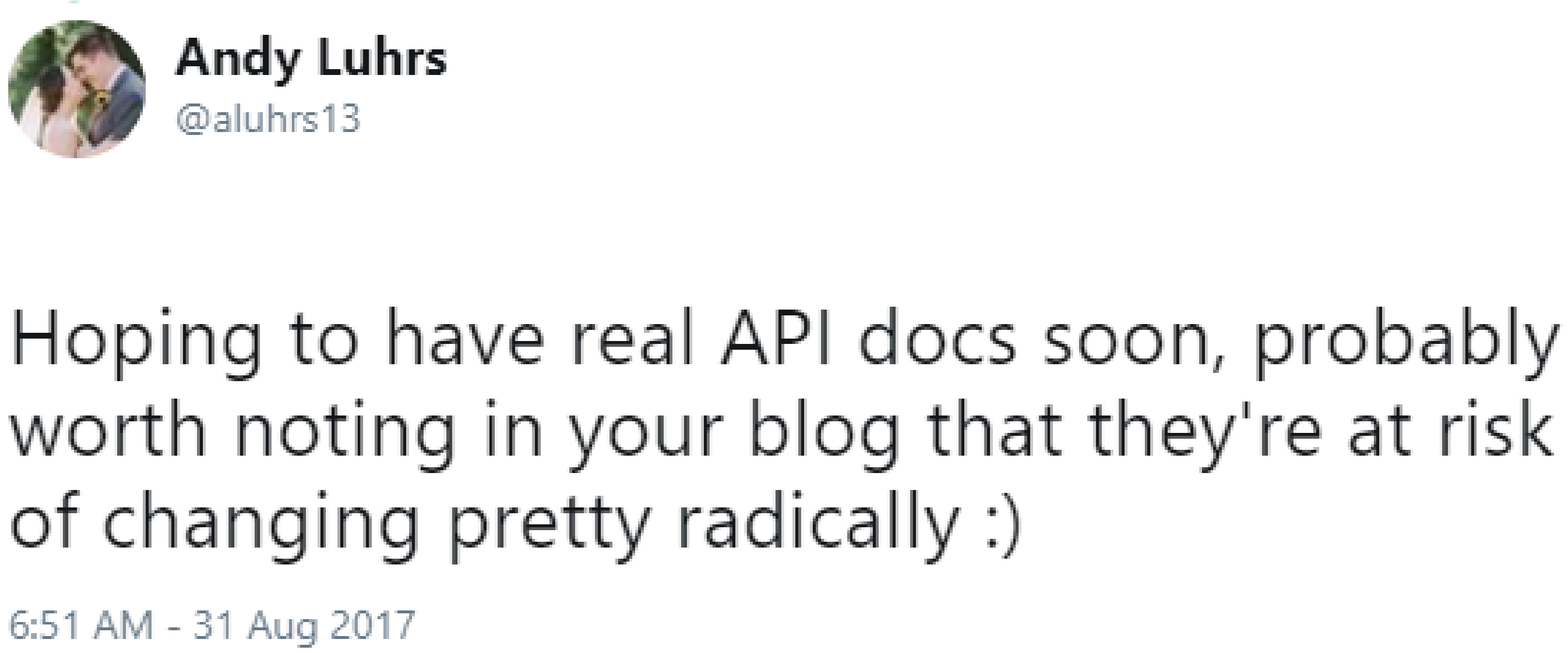
What we discovered so far

WinDbgX is a .NET/WPF application

It uses MEF-based composition

It has an undocumented API to write user extensions

Disclaimer



Andy Luhrs
@aluhrs13

Hoping to have real API docs soon, probably worth noting in your blog that they're at risk of changing pretty radically :)

6:51 AM - 31 Aug 2017

Finding interfaces

DbgX.Interfaces.dll: interfaces to extend the UI

DbgX.Interfaces.Internal.dll: interfaces to extend the UI or interact with the application/debugger

Storing extensions

.\Extensions: Only loads DbgX.*.dll files. Restrictive permissions

%LOCALAPPDATA%\DBG\UIExtensions: Loads any dll files

Extension 1: button to load SOS

When clicked, execute .loadby sos clr

After command is executed (by the button or by the user), gray-out the button

Step 1: Adding a button to the ribbon

WinDbgX uses the FluentRibbon library

First, create your ribbon control as you normally would

Then, implement and export IDbgRibbonTabGroupExtension to expose the button to WinDbgX

RibbonTabGroupExtensionMetadata allows you to decide in what tab/group your button should be put

Step 2: Executing a command

Import the IDbgConsole interface

ExecuteLocalCommandAsync: send a command to the debugger

ExecuteCommandAsync: try first to interpret as a WinDbg command (for example: .hh), then forward to the debugger

Execute(Local)CommandAndCaptureOutputAsync: execute the command but return the output instead of displaying it

PrintTextToConsole

Step 3: Listening to commands

Implement and export the IDbgCommandExecutionListener interface

OnCommandExecuted is called with the exact command executed

Step 4: (bonus) Monitoring debugging engine state

Know when the command can be executed

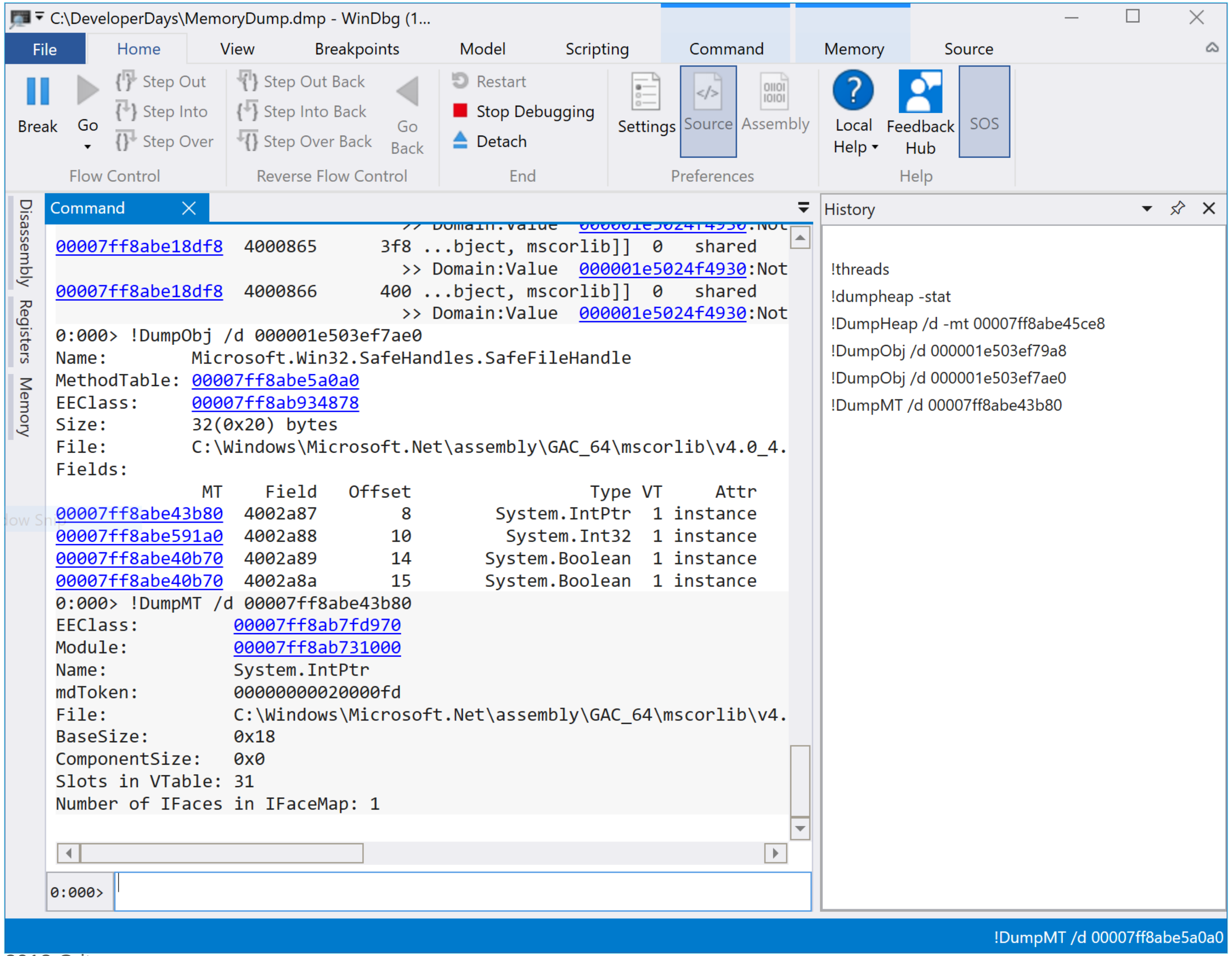
IDbgEngineState gives the state of the debugging engine, so that we know if we can actually execute commands

Extension 2: Command history

Log every command typed and their output

Open the result of a command in a new window

Use that window as an editor



```
00007ff8abe18df8  4000865      3f8 ...bject, mscorlib]]  0   shared
                                  >> Domain:Value  000001e5024f4930:Not
00007ff8abe18df8  4000866      400 ...bject, mscorlib]]  0   shared
                                  >> Domain:Value  000001e5024f4930:Not
0:000> !DumpObj /d 000001e503ef7ae0
Name:         Microsoft.Win32.SafeHandles.SafeFileHandle
MethodTable: 00007ff8abe5a0a0
EEClass:      00007ff8ab934878
Size:         32(0x20) bytes
File:         C:\Windows\Microsoft.Net\assembly\GAC_64\mscorlib\v4.0_4.
Fields:
              MT    Field   Offset                 Type VT     Attr
00007ff8abe43b80  4002a87        8        System.IntPtr  1 instance
00007ff8abe591a0  4002a88       10         System.Int32  1 instance
00007ff8abe40b70  4002a89       14       System.Boolean  1 instance
00007ff8abe40b70  4002a8a       15       System.Boolean  1 instance
0:000> !DumpMT /d 00007ff8abe43b80
EEClass:          00007ff8ab7fd970
Module:           00007ff8ab731000
Name:             System.IntPtr
mdToken:          00000000020000fd
File:             C:\Windows\Microsoft.Net\assembly\GAC_64\mscorlib\v4.
BaseSize:         0x18
ComponentSize:    0x0
Slots in VTable: 31
Number of IFaces in IFaceMap: 1
```

```
0:000>
```

History:

```
!threads
!dumpheap -stat
!DumpHeap /d -mt 00007ff8abe45ce8
!DumpObj /d 000001e503ef79a8
!DumpObj /d 000001e503ef7ae0
!DumpMT /d 00007ff8abe43b80
```

```
!DumpMT /d 00007ff8abe5a0a0
```

criteo.

Step 1: Opening a toolwindow

Make a control inheriting from ToolWindowView

Implement IDbgToolWindow and return the control in the GetToolWindowView method

Use NamedPartMetadata("CommandHistoryWindow") to expose the toolwindow

Import the IDbgToolWindowManager and use OpenToolWindow to open the toolwindow

Step 2: Capturing the output of the commands

Implement IDbgDmlOutputListener

OnDmlOutput: called with the output of the command

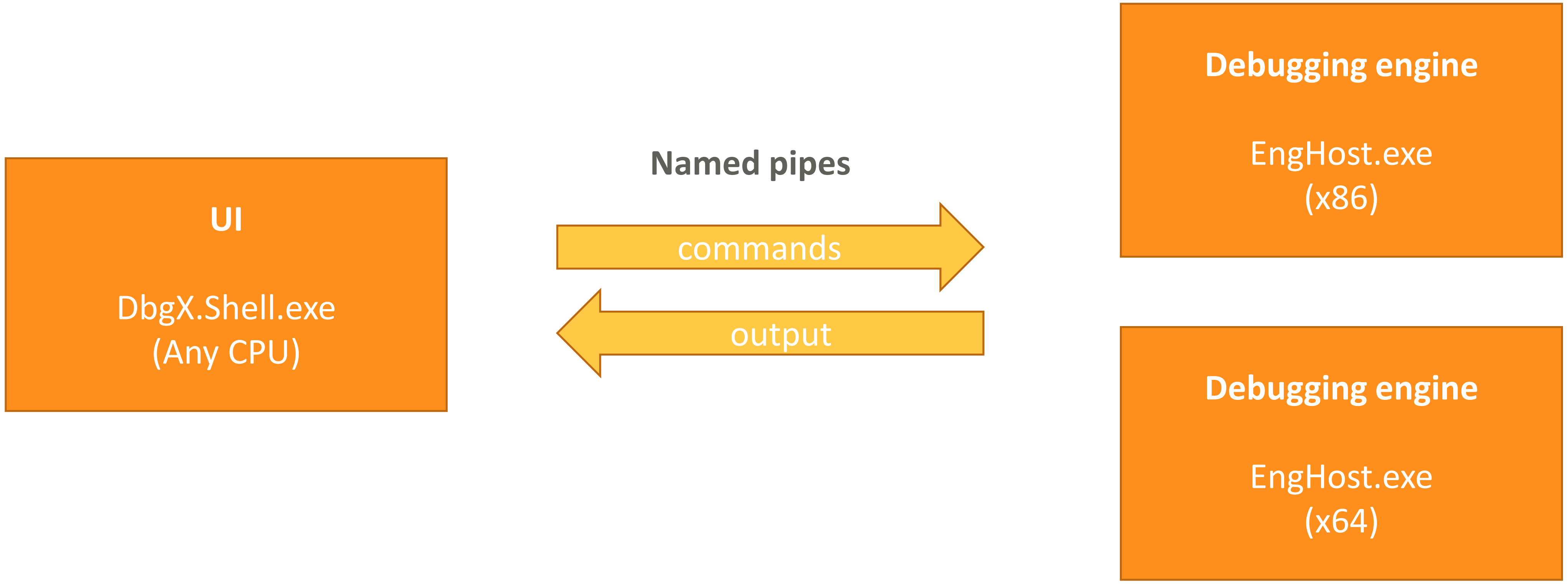
OnCommandCompletion: called when the command is finished

Extension 3: C# scripting

Write scripts in C# directly from WinDbg

Execute them in the debugger with ClrMD

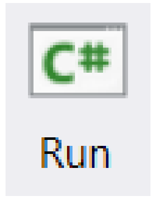
The WinDbg process model



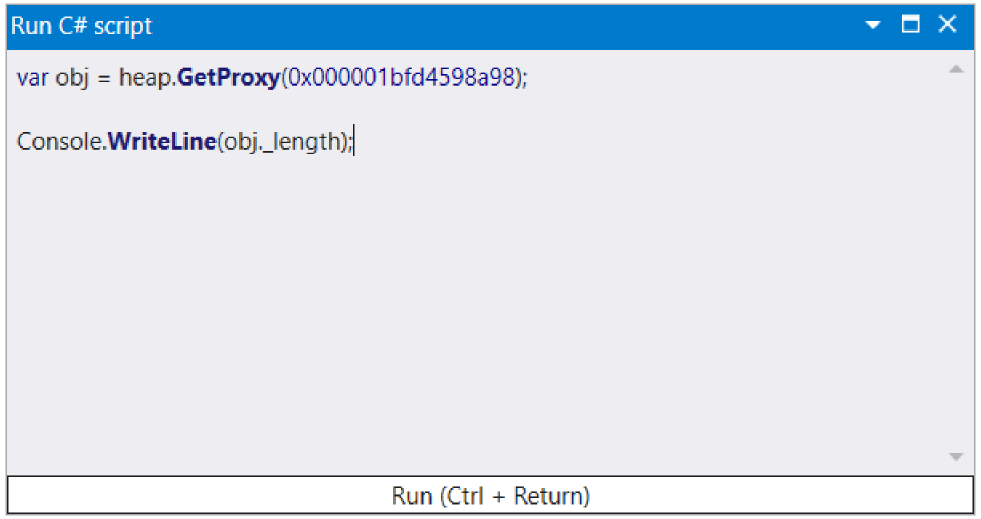
criteo

Step 1: The C# editor

- Add a button in the ribbon: `IDbgRibbonTabGroupExtension`



- When clicked, display a text editor: `IDbgToolWindowManager`



```
var obj = heap.GetProxy(0x000001bfd4598a98);

Console.WriteLine(obj._length);
```

Step 2: Find the bitness

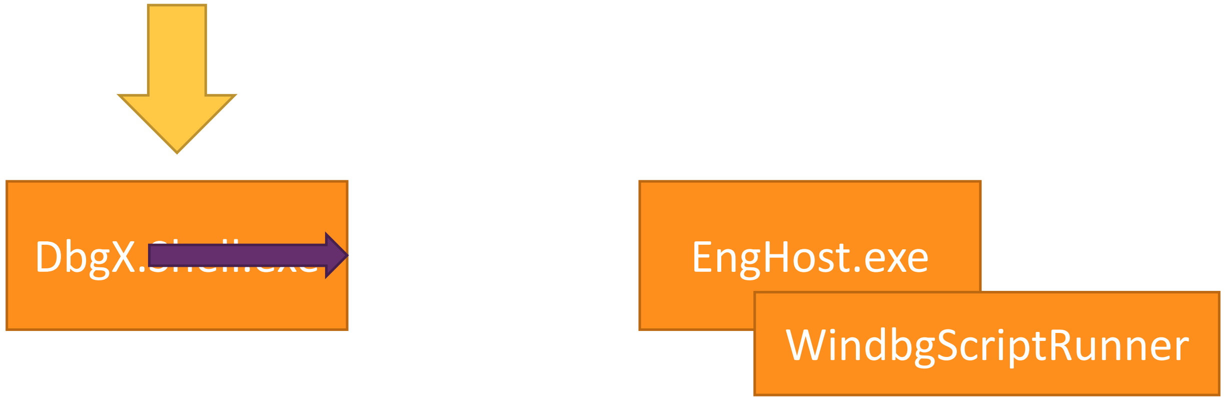
- ClrMD needs to run with same bitness as target
- Get information on the target: `IDbgTargetQuery`

- `IsPointer64BitAsync`: returns true if target is 64 bits

Step 3: Load the extension into the debugger

- Load the custom Windbg commands extension: `IDbgConsole`

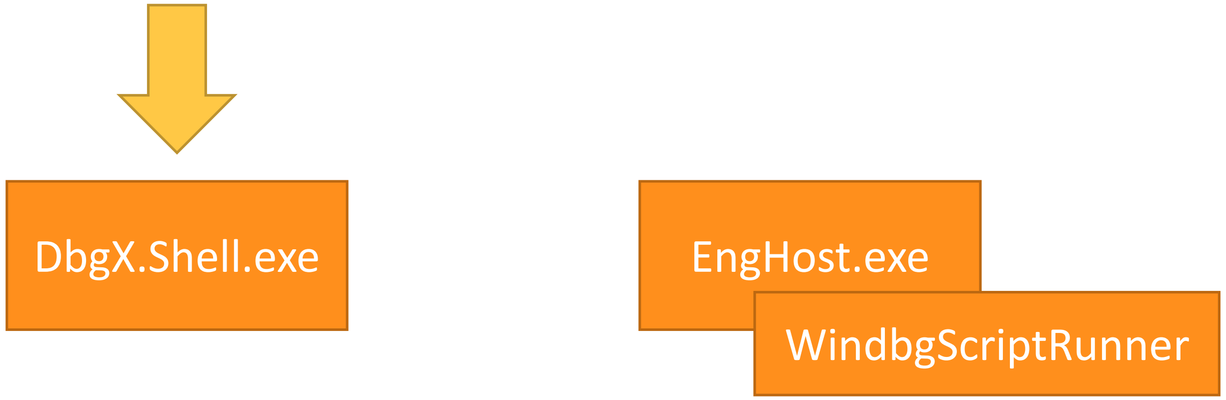
```
console.ExecuteLocalCommandAndCaptureOutputAsync(
 $".load C:\WindbgScriptRunner{bitness}.dll");
```



Step 4: Save the file and call the custom command

- Save the script in a temporary .cs file
- Give the path to the extension:

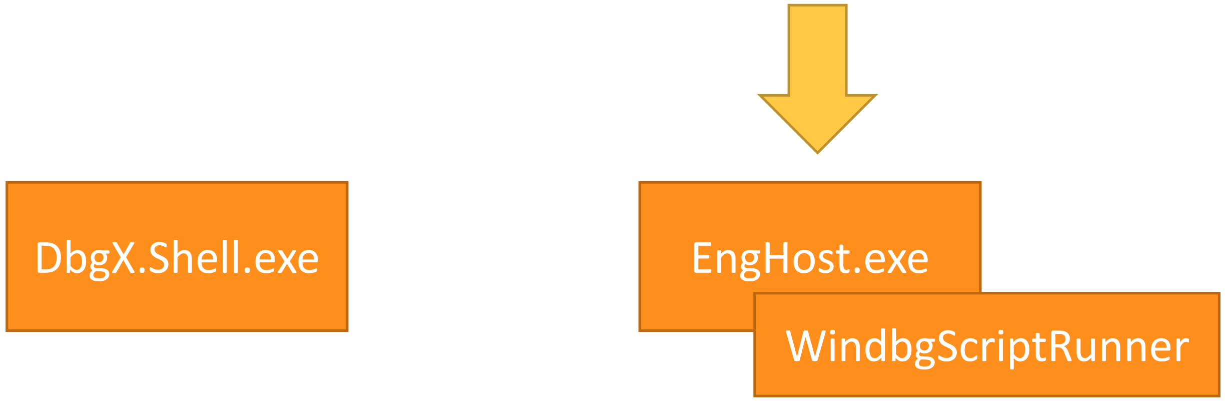
```
console.ExecuteCommandAndCaptureOutputAsync(
  $"!compileandrun {file}");
```



Step 5: Compile and execute

- Compile the C# code file:
  `Microsoft.CSharp.CSharpCodeProvider`

- Initialize the ClrHeap object in ClrMD: see Christophe Nasarre’s talk!



What’s next?

Still waiting for Microsoft to communicate on the extension points

Start today, write your own extensions

Demonstrate the use-case so that the extension points can be designed properly

Maybe someday an extension store?

Additional resources

Github: https://github.com/kevingosse/windbg-extensions/

- Source code of the three extensions, with more features
- Link to blog articles for additional info
- Twitter: @kookiz

criteo.