MODERN DEBUGGING WITH WINDBG PREVIEW

Chris Alladoum (@_hugsy_)
Offensive Security Researcher
SophosLabs

Axel Souchet (@0vercl0k)
Bit flipper

AGENDA

1. WinDbg
   - Basic commands & built-in extensions
   - Dynamic breakpoints
   - Using the Dynamic Markup Language (DML)

2. WinDbg Reloaded
   - Using Kd for remote user and kernel debugging
   - Natural Visualization (NatVis) & LINQ
   - Using the Debugger Data Model

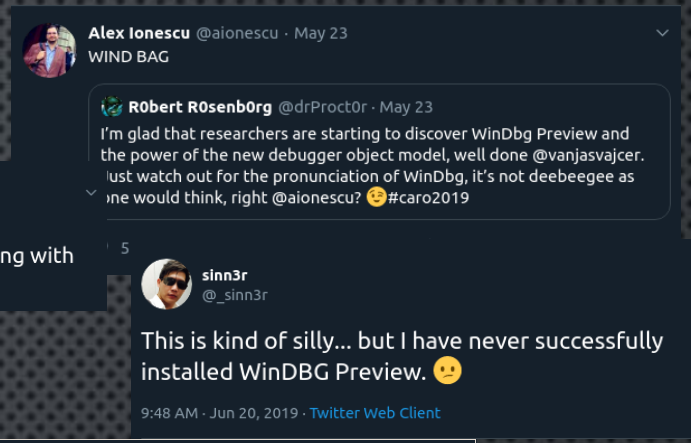
3. WinDbg Revolution
   - Dynamic scripting with JavaScript API
   - Time Travel Debugging

4. Final words and links to useful resources

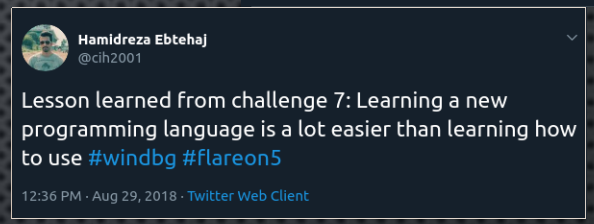
INTRODUCTION

WHY WINDBG ?

- LOTS OF LOVE/HATE AROUND IT



- BUT IN FACT ON WINDOWS, WINDBG IS THE *ONLY* DEBUGGER THAT:
  - SUPPORTS VARIOUS ARCHITECTURES (X86 / X64 / ARM64)
  - SUPPORTS KERNEL DEBUGGING
  - ALLOWS REMOTE DEBUGGING
  - IS EXTENSIBLE
    - THROUGH C++ DLL LOADING (EMBEDS BY DEFAULT TONS OF EXTENSIONS)
    - THROUGH SOME INLINE SCRIPTING (WDS)
    - THROUGH A PROPER SCRIPTING LANGUAGE (JS)



- AND WHY LEARN WINDBG PREVIEW ?
  - WELL BECAUSE IT'S REALLY NICE AND WE'RE IN 2019 SO IT'S OKAY NOT TO RELY ON TOOLS FROM 20 YEARS AGO

SETUP

- If you *really* want to install WinDbg "Legacy"
  - can be installed via
    - Visual Studio Installer (old Windows SDK installer) - in **Debugging Tools**
    - Using package managers like Chocolatey (https://chocolatey.org/)

```
C:\> choco install windbg
```

    - Some standalone MSIs can be found on microsoft.com

- Or better, just install WinDbg Preview
  - can be installed via the App Store (Windows 10 only)
  - You can also copy/paste the install directory
    - C:\Program Files\WindowsApps\Microsoft.WinDbg_$version

Setup – The Symbol Server

- The Optional-but-things-are-much-more-painful-without step

- Microsoft Public Symbol Server
  - https://msdl.microsoft.com/download/symbols

- Can be setup
  - In WinDbg via the .sympath command (or pressing Ctrl-S when stopped)
  - User-Specific / System-wide via _NT_SYMBOL_PATH environment variable

```
C:\> setx _NT_SYMBOL_PATH srv*C:\Symbols*http://msdl.microsoft.com/download/symbols
```

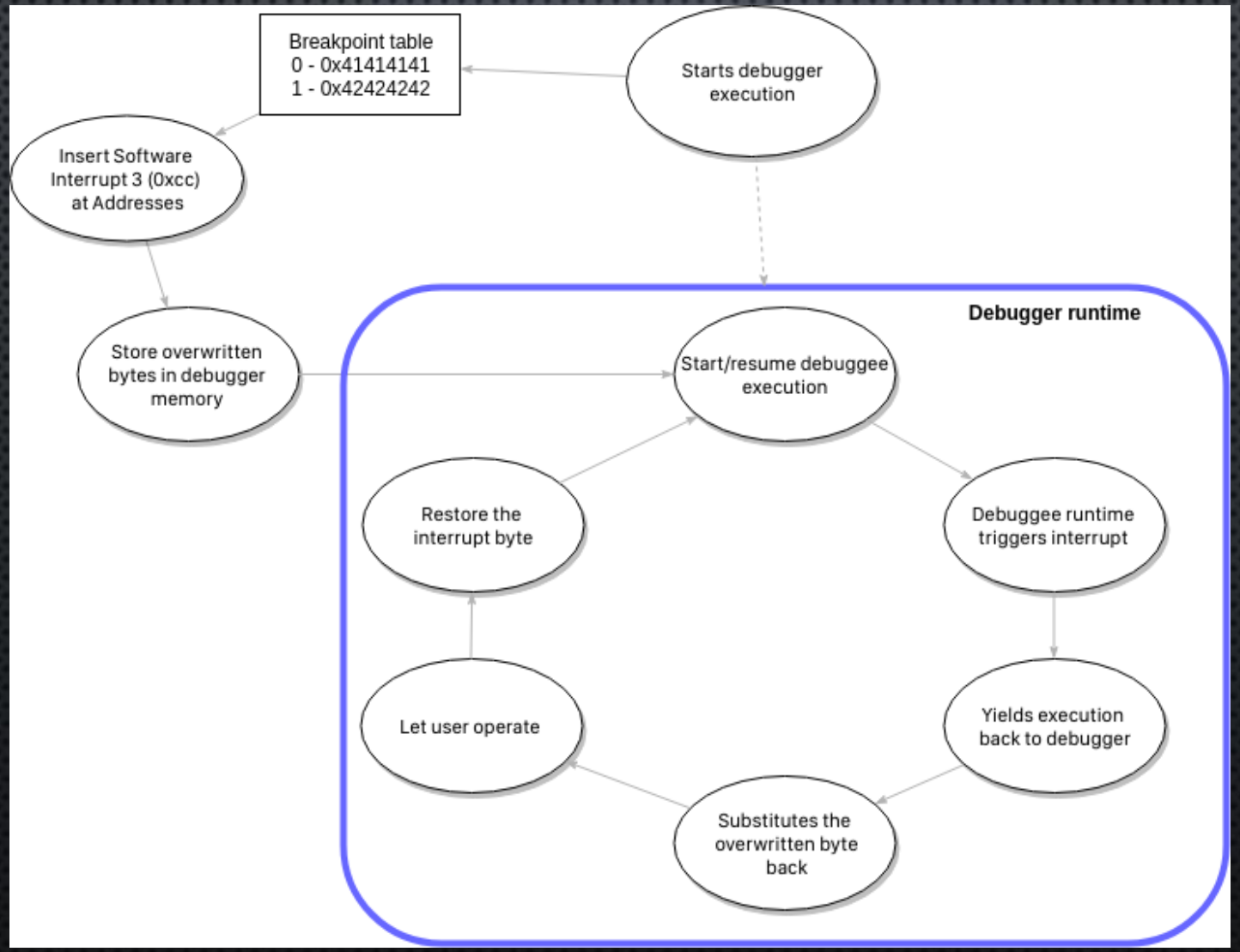
- If heavily used, it is also possible to share the symbols
  - Via a SMB share
  - By setting up a local SYMPROXY relay (in your company, at home)

What you'll need for the practical cases

- To best enjoy the workshop, you should have
  - Windows 10 Host (even trial is fine) - anything but windows 10 s is fine
  - WinDbg Preview
  - A descent editor (vscode, sublime, notepad++, emacs, vim...)
    - not wordpad !
  - Tools:
    - a hypervisor (hyper-v, virtualbox, vmware)
    - A Windows 10 VM with KdNet enabled to the host (or debug mode enabled on the host)
- Info
  - A copy of the materials will be shared

BASIC WINDBG

WHAT IS A DEBUGGER ?



WINDOWS DEBUGGING API

```
/**
 * My first non-functional Windows Debugger !
 */

DWORD dwProcessCreationFlags |= DEBUG_PROCESS;

CreateProcess("MyProcess.exe",..., dwProcessCreationFlags, ...);

while (TRUE)
{
  WaitForDebugEvent(&event, ...);

  switch (event)  // Only 9 events
  {
    case CREATE_PROCESS_DEBUG_EVENT: // there is also CREATE_THREAD_DEBUG_EVENT
      // Handle new process loaded (resp. thread)
      break;
    case EXCEPTION_DEBUG_EVENT:
      // Handle current exception (triggered from debuggee or debugger - break)
      break;
    case EXIT_PROCESS_DEBUG_EVENT: // there is also EXIT_THREAD_DEBUG_EVENT
      // Handle of process (resp. thread)
      break;
  }

  ContinueDebugEvent(...);
}
```

WinDbg

- Originally (i.e. back in MS-DOS days), **debug.com** (now open-source!)

- Now 2 major tools:
  - WinDbg (GUI)
    - **%ProgramFiles%\Windows Kits\10\Debuggers\x86\windbg.exe (x86)**
    - **%ProgramFiles(x86)%\Windows Kits\10\Debuggers\x64\windbg.exe (x64)**

  - CDB & NTSD (Console)
    - **%ProgramFiles%\Windows Kits\10\Debuggers\x86\cdb.exe (x86)**
    - **%ProgramFiles(x86)%\Windows Kits\10\Debuggers\x64\cdb.exe (x64)**

- Both connect to a KD server (which can be local or embedded)
  - kd.exe is a standalone binary that can be used for remote debugging
    - **%ProgramFiles%\Windows Kits\10\Debuggers\x86\kd.exe (x86)**
    - **%ProgramFiles(x86)%\Windows Kits\10\Debuggers\x64\kd.exe (x64)**

- Current WinDbg version is 10.0.17 (from SDK / WDK)

WinDbg "Preview"

- Newest and most active branch of WinDbg development
- New UI, AppContained (Metro-style) app – DbgX.Shell.exe
  - Installable via Microsoft Store, only Windows 10 Home and Pro, not 10 S
  - Yes – there is a dark mode (and since last June, also different color modes)
- Preview != Beta
- Backend engine stays the same (commands, syntax, etc.) as what was in WinDbg 10
- One game-changing builtin improvement (so far): Time Travel Debugging

FIRST COMMANDS: GENERIC

Action	Command	Examples
Help / Manual	`.hh <command>`	`.hh` `.hh !process`
Clear screen	`.cls`	

Dynamic evaluation	`?`	`? 40004141 - nt`
		`? 2 + 2`
		`? nt!ObTypeArrayIndex`

Comment	`$$`	`$$ this is a useful comment`
Print a string	`.echo` `.printf`	`.echo "Hello world"` `.printf "Hello %ma\n" , @$esp`
Command separator	`;`	`command1 ; command2`
Attach (Detach) to (from) process	`.attach` `.detach`	`.attach 0n<PidBase10>`
Display parameter value under different formats (hexadcimal, decimal, octal, ASCII, ...)	`.formats`	`.formats 0x42`
Change default base	`n`	`n 8` `n 10`
Quit the debugger (and kill the process if running)	`q`	
Restart application / Reboot	`.restart` `.reboot`	

FIRST COMMANDS: EXECUTION FLOW

Action	Command	Examples
Start or resume execution (go)	`g`	`g`

Dump register(s)	`r`	`r`
		`r eax`
		`r rax=42`

Step over / into	`p/ t`
	`(replace 'p' with 't')`
		`p`
		`pa 0xaddr (step over until 0xaddr is reached)`
		`pt (step over until return)`
		`pc (step over until next call)`

Execute until reaching current frame return address (go upper)	`gu`	

List module(s)	`lm`	`lm (UM: display all modules)`
		`lm (KM: display all drivers and sections)`
		`lm m *kern* (show module with pattern 'kern' )`

Get information about current debugging status	`.lastevent` `!analyze`	
Show stack call	`k`	

FIRST COMMANDS: MEMORY ACCESS

Action	Command	Examples
Read memory As	bytes: db word: dw dword: dd qword: dq pointer: dp	db @sp l30 dw 0xAddress lc0 dps @esp dyb @rip dW @rsp
Write memory As	bytes: eb word: ew dword: ed qword: eq ascii string: ea Unicode string: eu	eb @esp 41 41 41 41 ew @rip 0041 ea @pc "AAAA"
Dump memory to file	.writemem	.writemem C:\mem.raw @eip l1000
Load memory from file	.readmem	.readmem C:\mem.raw @rip l1000
Dump MZ/PE header info	!dh	!dh kernel32
Read / write physical memory	!db / !eb !dw / !ew !dd / !ed !dq / !eq	
Fill / Compare memory	f c	f @rsp l8 41 c @rsp l8 @rip
(KD x86/64) Display selector index	dg	dg 33

FIRST COMMANDS: MEMORY SEARCH

Action	Command	Examples
Search	`s [Type] [Range]`	

Search ASCII (Unicode)	`s –a <AddrStart> L<NbByte> "Pattern"`
	`s –a <AddrStart> <AddrEnd> "Pattern"`
	`(for Unicode – change `–a` with `–u`)`	

Search DWORD	`s –d <AddrStart> L<NbByte> 0xValue` `s –d <AddrStart> <AddrEnd> 0xValue`	
Search for pattern in command	`.shell -ci "<windbg command>" batch command`	`.shell -ci "!address" findstr PAGE_EXECUTE_READWRITE`

FIRST COMMANDS: BREAKPOINTS

Action	Command	Examples
Examine	`x`	`x nt!*CreateProcess*`
Display types	`dt`	`dt ntdll!_PEB @$peb` `dt ntdll!_TEB -r @$teb`

Set breakpoint (code)	`bp`	`bp 0xaddr / symbol`
		`bp /1 0xaddr (disable after 1 hit)`
		`bp 0xaddr 7 (disable after 6 hits)`

Disable breakpoint(s)
Delete breakpoint(s)
List breakpoints
	`bd`
	`bc`
	`bl`
		`bd *`

(Un)Set exception on event	`sx`	`sxe ld mydll.dll`
Set breakpoint (memory)	`ba`	`ba r 4 @esp`

Define breakpoint command	`bp … [Command]`
	`Where [Command] can be`
	`- an action: "r ; g"`
	`- a condition: ".if (@$rax == 1) {`
	`.printf \"hi\\\n\" }"`	

Enable breakpoint *after* n hit	`bp <address> N+1`	
Set "undefined" breakpoint	`bu <address>`	

FIRST COMMANDS: SYMBOLS AND STRING FORMATS

Action	Command	Examples
Examine	`x`	`x /t /v ntdll!*CreateProcess*`
Display types	`dt`	`dt ntdll!_PEB @$peb`
List nearest symbols	`ln`	`ln 0xaddr`
Set / update symbol path	`.sympath`	
Load module symbols	`ld`	`ld Module` `ld *`

Description	Formatter	Examples
ASCII C string (i.e. NULL terminated)	`%ma`	
Wide C string (i.e. NULL terminated)	`%mu`	
UNICODE_STRING** string	`%msu`	
Print the symbol pointed by address	`%y`	`.printf "%y\n", 8abc2010 // returns nt!PsLoadedModuleList`
Pointer	`%p`	`.printf "%p\n", nt!PsLoadedModuleList // returns 8abc2010`

FIRST COMMANDS: CONVENIENCE VARIABLES AND FUNCTIONS

Action	Command	Examples
Program entry point	`$exentry`	`bp $exentry`
Process Environment Block	`$peb`	`dt _PEB @$peb`
Thread Environment Block	`$teb`	`dt _TEB @$teb`
Return Address	`$ra`	`g @ra`
Instruction Pointer	`$ip`	
Size of Page	`$pagesize`	
Size of Pointer	`$ptrsize`	
Process ID	`$tpid`	
Thread ID	`$tid`	

Dereference memory	`poi(<AddrOrSymbol>) : dereference pointer size`
	`dwo(): dereference DWORD`
	`qwo(): dereference QWORD`
		`db poi( @$rax )`

Note: always use '@' when not query symbols (i.e. registers, variables, etc.)

FIRST COMMANDS: USEFUL EXTENSIONS

WinDbg extensions always start with '!' (bang). They are located in the `Debugger\%ARCH%\winext` directory .

Action	Command	Examples
Detailed information about loaded DLLs	`!dlls`	`!dlls -I (show load order)` `!dlls -c 0xaddr (show DLL containing 0xaddr)`
Get mapping information	`!address`	`!address -f:MEM_COMMIT`
Change verbosity of symbol loader	`!sym noisy` `!sym quiet`	
Dump PEB/TEB information	`!peb` `!teb`	
Analyze the reason of a crash	`!analyze`	`!analyze -v`
Convert an NTSTATUS code to text	`!error`	`!error c0000048` `!error @eax`
Perform heuristic checks to determine the exploitability of a bug (external)	`!exploitable`	
Encode/decode pointer encoded by KernelBase API EncodePointer()	`!encodeptr32 (or 64)` `!decodeptr32 (or 64)`	
Display the current exception handler	`!exchain`	
Dump heap information	`!heap`	

HANDS ON

- C:\Windows\System32\notepad.exe
  - Practice on running / continuing / attaching to the process
  - Dump structures (PEB, TEB, `dt` etc.)

- \bin\Payroll\Payroll 2019.pdf.exe
  - Find and dump the shellcode encoded
    - Might want/need to disable your AV (might flag the encrypted meterpreter shellcode)

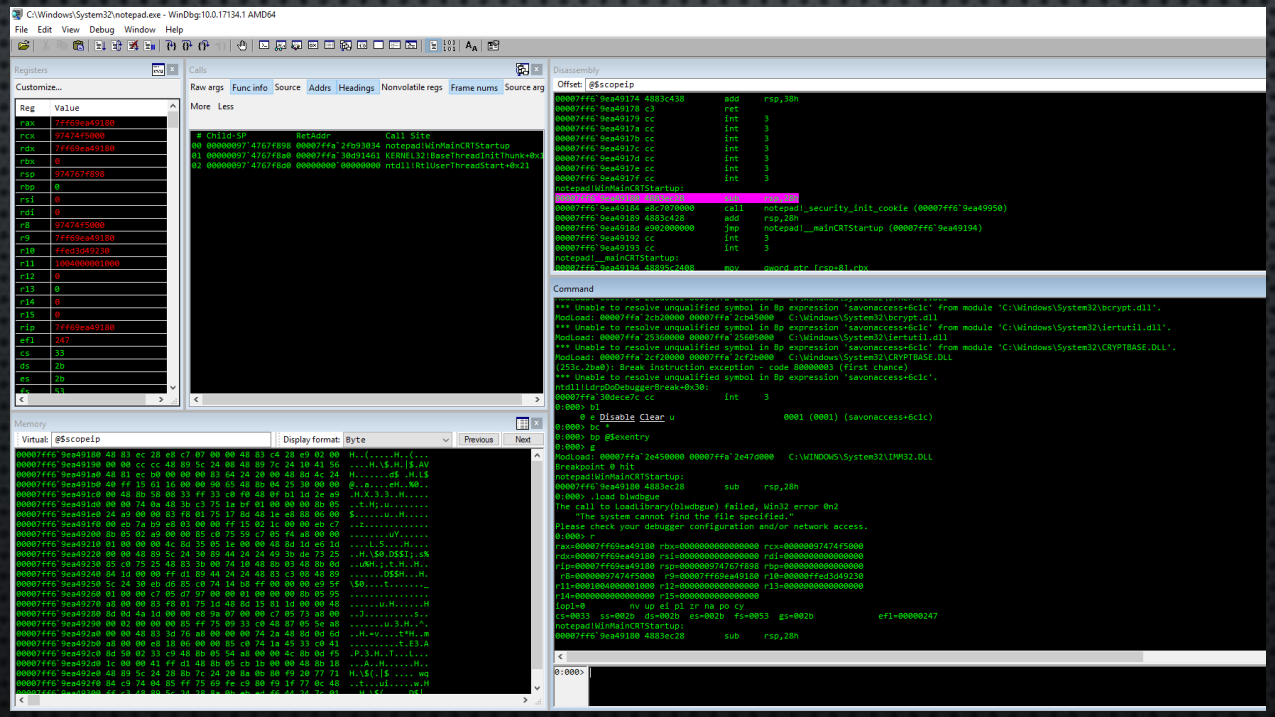
First commands: customization

- WinDbg works in workspaces
  - Can create custom debugging environment (custom setup, modified fonts, colors)
  - Can also be used to store breakpoints
    - Using `bu`
  - Use startup toggle `-W <WkSpc>` (or Ctrl-W) to select it

- WinDbg has a Debugger Markup Language (DML) to create dynamic content
  - Is toggled on/off with `.prefer_dml` set to 1 or 0
  - Can be used with `.printf` to create custom dynamic content (and scripts)

FIRST COMMANDS: CUSTOMIZATION

- SOME EXAMPLES IN \WORKSPACES
  - CHECK OUT THE DIFFERENT WORKSPACES
  - FEEL FREE TO CREATE YOUR OWN



WINDBG RELOADED

Better Debugging

- Introducing Windbg Preview
- Move on from WinDbg "Legacy"
  - And any other inferior debugger
- From now everything will assume WinDbg Prevew only
  - Meaning debugger host must run Windows 10

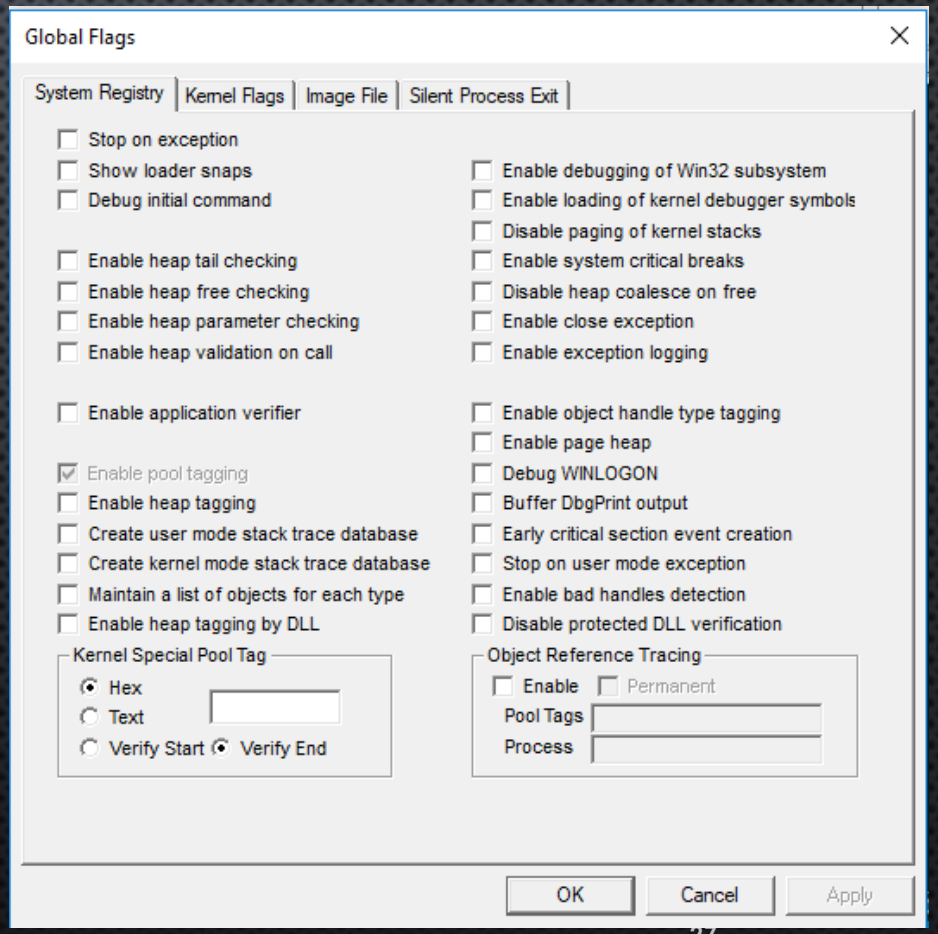
Better Debugging

- Application Verifier (appverifier.exe)
  - Helps detecting (among others):
    - Dead locks
    - Invalid handles
    - Memory Leaks (umdh.exe)

- Driver Verifier (verifier.exe)
  - Monitor one or many drivers' activity looking for illegal activity
    - Memory corruption
    - Pool leak / double free
    - Triggers interrupt (catchable by Kd) with useful analysis (use with !analyze, !verifier, !error)

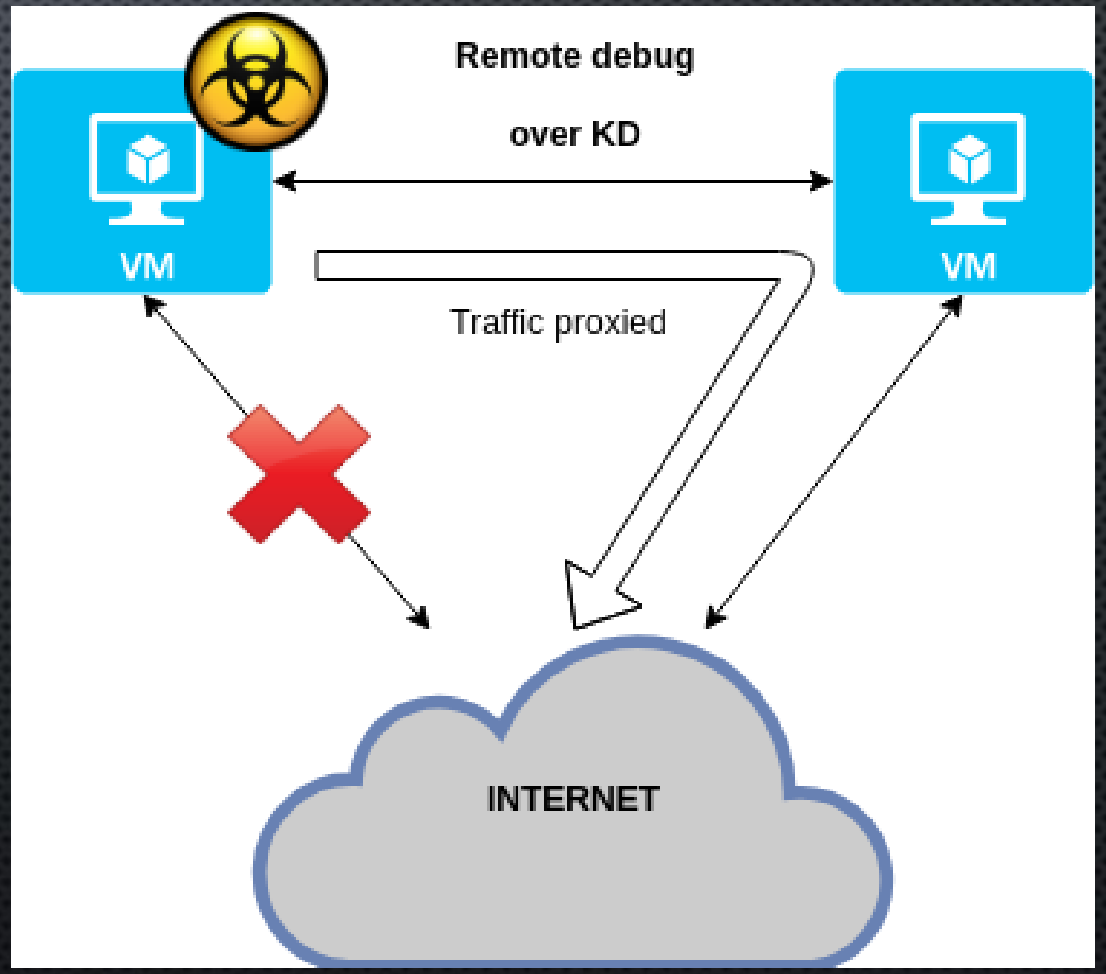
BETTER DEBUGGING

- GFLAGS (GFLAGS.EXE)
  - GLOBAL FLAGS (NEEDS ADMINISTRATOR PRIVILEGE)
  - GUI OR CONSOLE
  - CAN HAVE PROCESS OR SYSTEM-WIDE SETTINGS
  - CAN INTERACT WITH WINDBG (IF SETUP AS POST-MORTEM)
    - BREAK ON EXCEPTIONS
    - DEBUG ON LOAD
    - ENABLE PAGE HEAP
    - DETECT BAD HANDLES
    - HEAP TAG BY DLL



REMOTE DEBUGGING USING KD

- KD ALLOWS TO CREATE A SETUP LIKE THIS:
  - EVERYTHING (KERNEL, SYSTEM, NETWORK) CAN BE CAPTURED
  - KD SUPPORTS MANY COMMUNICATION PROTOCOLS (SERIAL, USB, UDP/IP, FIREWIRE)



Remote Debugging using KD

- User Mode : connecting to a **Remote Process**
  - Relies on a "Connection String"
    - Built from [Transport]:[Options]
      - Transport can be net(work), serial (com), USB, named pipe, ssl)
  - Debuggee initiates the debug session over the network
    - For example, the debuggee has IP 192.168.57.4, debugs NotePad on TCP/50000

```
C:\> dbgsrv32.exe -tcp:port=50000 –c[s] C:\windows\System32\notepad.exe
```

  - Debugger connects using the defined "Connection String"

```
C:\> windbgx.exe -remote tcp:port=50000,server=192.168.57.4
```

  - WinDbg on Debugger side acts as proxy (nothing is done on Debugger host)
    - Safe environment for remote debug even of malicious software

REMOTE DEBUGGING USING KD

- USER MODE : CONNECTING TO A **PROCESS SERVER**
  - DEBUGGEE CREATES A DEBUG SERVER SESSION WITH **DBGSRV** (FOR EX. ON TCP/50000) - `-t` FLAG
    - PROCESS RUNS IN BACKGROUND

```
C:\> DBGSRV32.EXE -T TCP:PORT=50000
OR
C:\> DBGSRV64.EXE -T TCP:PORT=50000
```

  - DEBUGGER CAN CONNECT USING `-PREMOTE` FLAG AND THE "CONNECTION STRING"

```
C:\> WINDBGX.EXE -PREMOTE TCP:PORT=50000,SERVER=192.168.57.4
```

    - OR USING REMOTE CONNECTION OPTIONS FROM INSIDE WINDBG (CTRL-R)

  - THEN DEBUGGER CAN EITHER
    - CREATE A NEW PROCESS
    - OR ATTACH TO A EXISTING ONE

Remote Debugging using KD

- Kernel Mode
  - Local KD (LKD)
  - Hack first introduced by SysInternals LiveKD (still available, works on Windows 2000+)

  - Similar capabilities is now built-in from Windows 8+
    - Simply needs to enable Debug mode from `bcdedit`

```
C:\> BCDEDIT /SET {CURRENT} DEBUG ON
C:\> BCDEDIT /DBGSETTINGS LOCAL
```

  - **Must** be launched as Administrator

```
C:\> WINDBGX.EXE -KL
```

  - Convenient for quick access to memory structures, etc. but cannot be interacted with
    - No breakpoint

Remote debugging using KD

- Kernel Mode
  - KD Over (UDP/IP) Network : KDNet
  - Windows 8+ only (for XP / 7, prefer KDCom – preferably using VirtualKd)
  - Pretty fast and secure (AES encrypted)

  - Can be activated on debuggee via `bcdedit`

```
C:\> BCDEDIT /SET {CURRENT} DEBUG ON
C:\> BCDEDIT /DBGSETTINGS NET HOSTIP:IP.OF.THE.SERVER PORT:50000 KEY:WINDBG.KERNEL.DEBUG.TRAINING
```

  - And make the server wait for incoming connection

```
C:\> WINDBGX.EXE -K NET:PORT=50000,KEY=WINDBG.KERNEL.DEBUG.TRAINING
```

  - God mode on the debuggee
    - Stop execution, read/write anywhere in memory (phy/virt), reboot, etc.

REMOTE DEBUGGING USING KD

- Extra hints (KD):
  - Increase the kernel debug verbosity level (from **KdPrintEx**)
    - Temporarily during runtime from WinDbg (lost once session is closed)

```
kd> ed nt!Kd_Default_Mask 0xf
```

    - Permanently from registry hive (on Debuggee)

```
C:\> reg add "HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Debug Print Filter" /v DEFAULT
/t REG_DWORD /d 0xf
```

  - Enable the Driver Verifier (verifier.exe - from SDK) to assist in finding/tracking bugs

```
C:\> verifier.exe /standard /all
Or more specifically
C:\> verifier.exe /standard /driver MyDriver.sys
```

HANDS-ON – KDNET

- Use DbgSrv to create a Debugging Server
  - Connect to the process server / debug instance using WinDbg Preview
  - Start a new process

- Use LKD / Kdnet to kernel debug a Windows environment
  - Setup a kernel debugging environment
  - *Bonus*: load a driver (for example HEVD.sys) using .kdfiles
    - See https://twitter.com/windbgtips/status/1062060718818701313

VISUALIZATION & DATA MODEL

NATVIS

- Some memory representations can be a pain when debugging/reversing
  - C structures
  - C++ classes

- WinDbg 10+ and Preview introduces Natural Visualization
  - Idea: Provide the debugger a more "human friendly" format for classes / structures
  - XML format (`.natvis` extension by default)
  - Can be loaded automatically and/or at runtime using the `.nvload` command
  - Can be enumerated using `.nvlist`
  - Can be invoked via the `dx` (Display eXpression) command
    - `0:000> dx MyStruct`
    - `0:000> dx *((MyStruct*) 0xAddress)`

- Check for support in WinDbg
  - Via `.scriptproviders`

```
0:000> .scriptproviders
Available Script Providers:
    NatVis (extension '.NatVis')
```

NATVIS

- Integrated with DML too
- Builtins and examples can be found in
  - "%ProgramFiles%\Windows Kits\10\Debuggers\x64\Visualizers"
- Very basic example (from NatVis builtins):

```
kd> dx (_DEVICE_OBJECT*) 0xFFFF9B0672BE32D0
(_DEVICE_OBJECT*) 0xFFFF9B0672BE32D0                 : 0xffff9b0672be32d0 : Device for "\Driver\NDIS" [Type: _DEVICE_OBJECT
    [<Raw View>]     [Type: _DEVICE_OBJECT]
    Flags            : 0x50
    UpperDevices     : None
    LowerDevices     : None
    Driver           : 0xffff9b0672be3e60 : Driver "\Driver\NDIS" [Type: _DRIVER_OBJECT *]
```

DEBUGGER DATA MODEL

- Structured hierarchy of objects
  - Functions, properties, settings
- WinDbg internal objects
  - Can be interacted with directly in the debugger (`dx`), but also exposed to JS API
- Leverages LINQ (Language-Integrated Query) as query/filtering language
  - From C#
  - SQL-ish CRM-ish -like syntax
- DDM allows also to create / modify / delete variables and methods
  - In a clearer way than MASM syntax (and with explicit types)
  - Has type completion from command prompt
  - Great amount of built-in objects
    - WinDbg Preview has a expandable menu for them
    - On WinDbg, just use `dx Debugger` (with DML)

DEBUGGER DATA MODEL

- Quick example – get process information with KD:
  - Traditional way:

```
KD> !process
$$$ Scroll for the process name
KD> !process 0 0 foobar.exe
$$$ Spot the EPROCESS address
KD> dt nt!_EPROCESS 0x<address>
          nt!_EPROCESS
             +0x000 Pcb                : _KPROCESS
             +0x2d8 ProcessLock        : _EX_PUSH_LOCK
             +0x2e0 UniqueProcessId    : 0x00000000`00001488 Void
             +0x2e8 ActiveProcessLinks : _LIST_ENTRY [ 0xffffff802`1a0523f0 - 0xffff9b06`73127668 ]

          […]
```

DEBUGGER DATA MODEL

- Quick example – get process information with KD:
  - The (hard) DDM way:

```
$$ get the process list

kd> dx @$ps = Debugger.Utility.Collections.FromListEntry( *(nt!_LIST_ENTRY*)&(nt!PsActiveProcessHead),
"nt!_EPROCESS", "ActiveProcessLinks")

$$ filter out objects whose name won't match

kd> dx @$ps.Where(p => ((char*)p.ImageFileName).ToDisplayString("sb") == "foobar.exe" )
```

DEBUGGER DATA MODEL

- Quick example – get process information with KD:
  - The (easy) DDM way:

```
$$ Same as before, but using predefined variables
kd> dx @$ps = @$cursession.Processes.Where(p=>p.Name == "foobar.exe")
```

- Can be invoked from command prompt, WinDbg scripts (WDS) or JavaScript

- Can define functions too:
  - **kd> dx @$f = ((x , y) => x + y)**
  - **kd> dx @$f(10, 20)**

DEBUGGER DATA MODEL

Variable description	Command	Example
Create a variable	`dx @$myVar = VALUE`	`dx @$ps = @$cursession.Processes`
Delete a varable	`dx @$vars.Remove("VarName")`	`dx @$vars.Remove("ps")`
List user defined variable	`dx @$vars` `dx @$Debugger.State.UserVariables`	

`` `dx Debugger.State.DebuggerVariables` `` for full details

Function description	Command	Example
Filter objects	`.Where( [FILTER PATTERN] )`	`.Where( x => x.Name == "System" && p.Id == 0 )`
Project object	`.Select( [PROJECTION KEYS] )`	`.Select( p => new { Item1 = p.Name, Item2 = p.Id } )`
Access n-th element of iterable	`[ INDEX ]`	`@$cursession.Processes[ 0 ]`
Get the number of objects in iterable	`.Count()`	

https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/using-linq-with-the-debugger-objects#supported-linq-syntax---query-methods

DEBUGGER DATA MODEL - SESSION ATTRIBUTES

```
0:000> dx  @$cursession.Attributes.Target
    IsUserTarget      : true
    IsKernelTarget    : false
    IsDumpTarget      : false
    IsTTDTarget       : true
    IsReptTarget      : false
    IsReplayableTarget : true
    IsLiveTarget      : true
    IsLocalKernelTarget : false
    IsExdiKernelTarget : false
    IsConnectedKernelTarget : false
    IsNTTarget        : true
    Details
```

```
0:000> dx @$cursession.Attributes.Machine
    FullName          : x64
    AbbrevName        : AMD64
    PageSize          : 0x1000
    PointerSize       : 0x8
    IsBigEndian       : false

0:000> dx  @$cursession.Attributes.Target.Details
    DumpFileName      :
C:\Users\over\Downloads\ping01.run
```

DEBUGGER DATA MODEL - FRAMES

```
0:000> dx @$curthread.Stack.Frames[0].Attributes
    InstructionOffset : 0x7ffba2c5c344
    ReturnOffset      : 0x7ffba00d9670
    FrameOffset       : 0xd538eccbd0
    StackOffset       : 0xd538eccbd8
    FuncTableEntry    : 0x0
    Virtual           : 1
    FrameNumber       : 0x0
```

DEBUGGER DATA MODEL - PEB

```
0:000> dx @$curprocess.Environment.EnvironmentBlock
    [+0x000] InheritedAddressSpace : 0x0 [Type: unsigned char]
    [+0x001] ReadImageFileExecOptions : 0x0 [Type: unsigned char]
    [+0x002] BeingDebugged    : 0x0 [Type: unsigned char]
    [+0x003] BitField         : 0x4 [Type: unsigned char]
…
```

DEBUGGER DATA MODEL - TEB

```
0:000> dx @$curthread.Environment.EnvironmentBlock
    [+0x000] NtTib            [Type: _NT_TIB]
    [+0x038] EnvironmentPointer : 0x0 [Type: void *]
    [+0x040] ClientId         [Type: _CLIENT_ID]
...
```

DEBUGGER DATA MODEL - I/O

```
0:000> dx 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 existance of a file at the specified path]
    OpenFile         [OpenFile(path) - Opens a file read/write at the specified path]
    TempDirectory    : C:\Users\over\AppData\Local\Temp
```

DEBUGGER DATA MODEL - DISASSEMBLER

```
0:000> dx @$d = Debugger.Utility.Code.CreateDisassembler()
@$d = Debugger.Utility.Code.CreateDisassembler()

0:000> dx -v @$d
    DisassembleBlocks [DisassembleBlocks(address) - Returns an iterable of basic blocks at the given address]
    DisassembleInstructions [DisassembleInstructions(address) - Returns an iterable of instructions disassembled at the
given address]
    DisassembleFunction [DisassembleFunction(address) - Returns a flow graph of all basic blocks within the function]
    GetRegister      [GetRegister(regId) - Returns a register object from a register ID]

0:000> dx @$d.DisassembleInstructions(@rip).First()
@$d.DisassembleInstructions(@rip).First()                 : ret
    Address          : 0x7ffba2c5c344
    Attributes
    CodeBytes
    Length           : 0x1
    LiveVariables
    Operands
    SourceInformation
    SourceDataFlow   : Invalid argument to method 'getModuleSymbol' [at CodeFlow (line 713 col 9)] [at CodeFlow (line 740
col 5)]
```

DEBUGGER DATA MODEL - SYNTHETIC TYPES

```
0:000> dx Debugger.Utility.Analysis.SyntheticTypes
Debugger.Utility.Analysis.SyntheticTypes
    ReadHeader       [ReadHeader(headerPath, module, [attributes])- Reads a header at the file and defines synthetic constructors for all structs and unions
in the header referencing existing types in the given module.  Attributes is a set of key value pairs.  Presently, it may contain Macros (another set of
key/value pairs), an optional ReturnEnumsAsObjects value (defaulting to false), and an optional RegisterSyntheticTypeModels value (defaulting to false).  If
synthetic type models are registered, they are named DataModel.SyntheticTypes.<HeaderName>.<TypeName> and can be extended similarly.]
    CreateInstance   [CreateInstance(typeName, addrObj) - Creates an instance of a synthetic type and returns it.  The type must have been read from a
header or defined through other APIs here]
    TypeTables
```

https://github.com/microsoft/WinDbg-Samples/tree/master/SyntheticTypes

DEBUGGER DATA MODEL - GALLERIES

```
0:000> dx -r1 Debugger.Settings.Extensions.ExtensionRepositories.overgallery
    Id                : 690F3350-53EE-4EC4-A95B-750068897ED4
    LocalCacheRootFolder : c:\work\codes\windbg-scripts\Manifest
    IsEnabled         : true
    URL               : https://github.com/0vercl0k/windbg-scripts

0:000> dx -r1 Debugger.Settings.Extensions.ExtensionRepositories.LocalInstalled
    IsEnabled         : true
```

Create yours: https://github.com/microsoft/WinDbg-Samples/tree/master/Manifest
Gallery example: https://github.com/0vercl0k/windbg-scripts#installing-the-script-gallery

DEBUGGER DATA MODEL

- Practical examples
  - How to get the name of the process with PID=1234
  - How to dump info about process with PID=1234
  - How to enumerate all the threads for the process with PID=1234

DEBUGGER DATA MODEL

- Practical examples
  - How to get the name of the process with PID=1234
    - `kd> dx @$cursession.Processes.Where(p => p.Id == 1234).Select( p => new { Pid = p.Id, Name = p.Name })`
  - How to dump info about process with PID=1234
    - `kd> dx @$cursession.Processes[1234]  $$ <<-- not hexa`
  - How to enumerate all the threads for the process with PID=1234
    - `kd> dx Debugger.Utility.Collections.FromListEntry(@$cursession.Processes[1234].KernelObject.ThreadListHead, "nt!_ETHREAD", "ThreadListEntry")`

DEBUGGER DATA MODEL

- Practical examples
  - How to list all processes being debugged ?
  - How to get the full path of all the processes with the exact same Token pointer to System's ?

DEBUGGER DATA MODEL

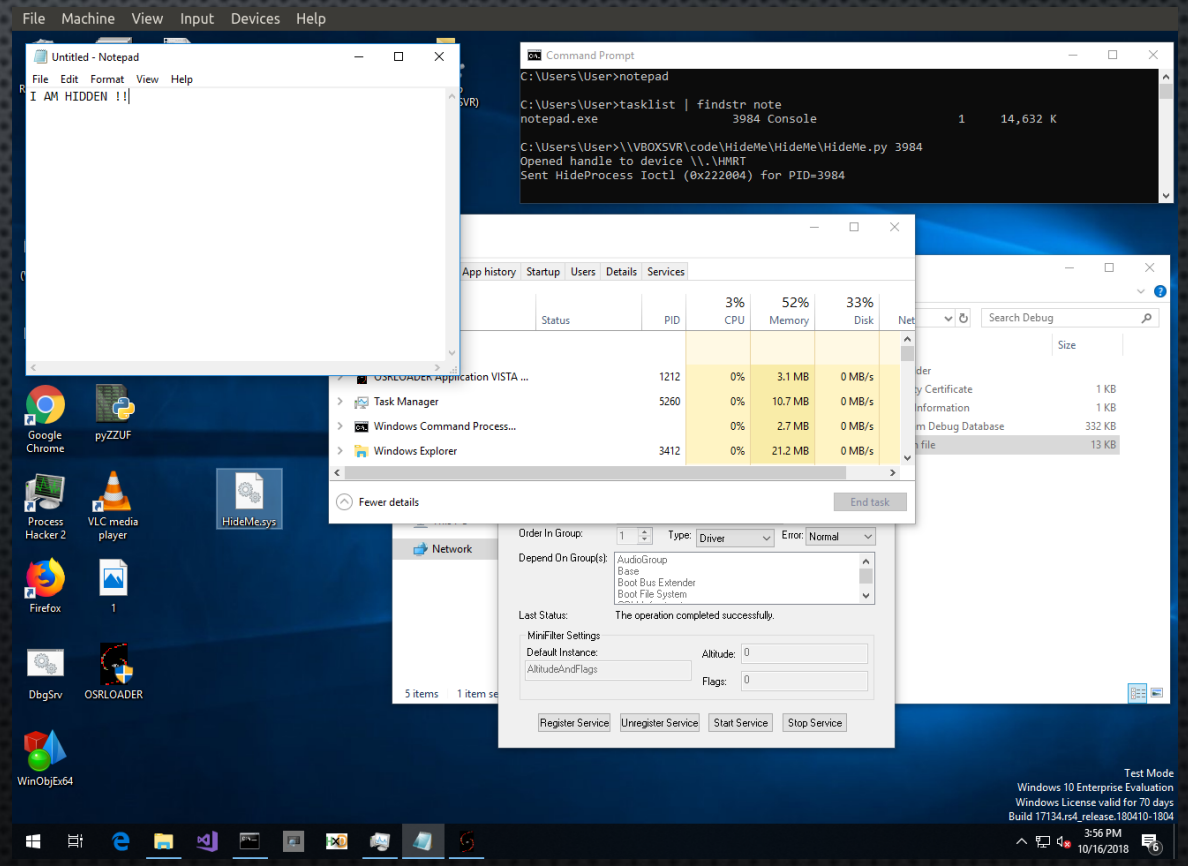
- Practical examples
  - How to list all processes being debugged ?
    - `kd> dx -g @$cursession.Processes.Where( p => p.KernelObject.Peb->BeingDebugged == 1)`
  - How to get the full path of all the processes with the exact same Token pointer to System's ?
    - `kd> dx -r0 @$system_token = @$cursession.Processes[4].KernelObject.Token`
    - `kd> dx @$cursession.Processes.Where( p => (void*)&(p.KernelObject.Token) == (void*)&(@$system_token) ).Select( p => new { Name = p.KernelObject->SeAuditProcessCreationInfo.ImageFileName } )`
      - * must be adjusted due to RefCnt

DEBUGGER DATA MODEL

- Practical examples
  - Open any debugging user/kernel session
    - Browse through the already built-in objects
      - Debugger
      - @$cursession
      - @$curprocess
    - Build your own queries
    - Build your own functions

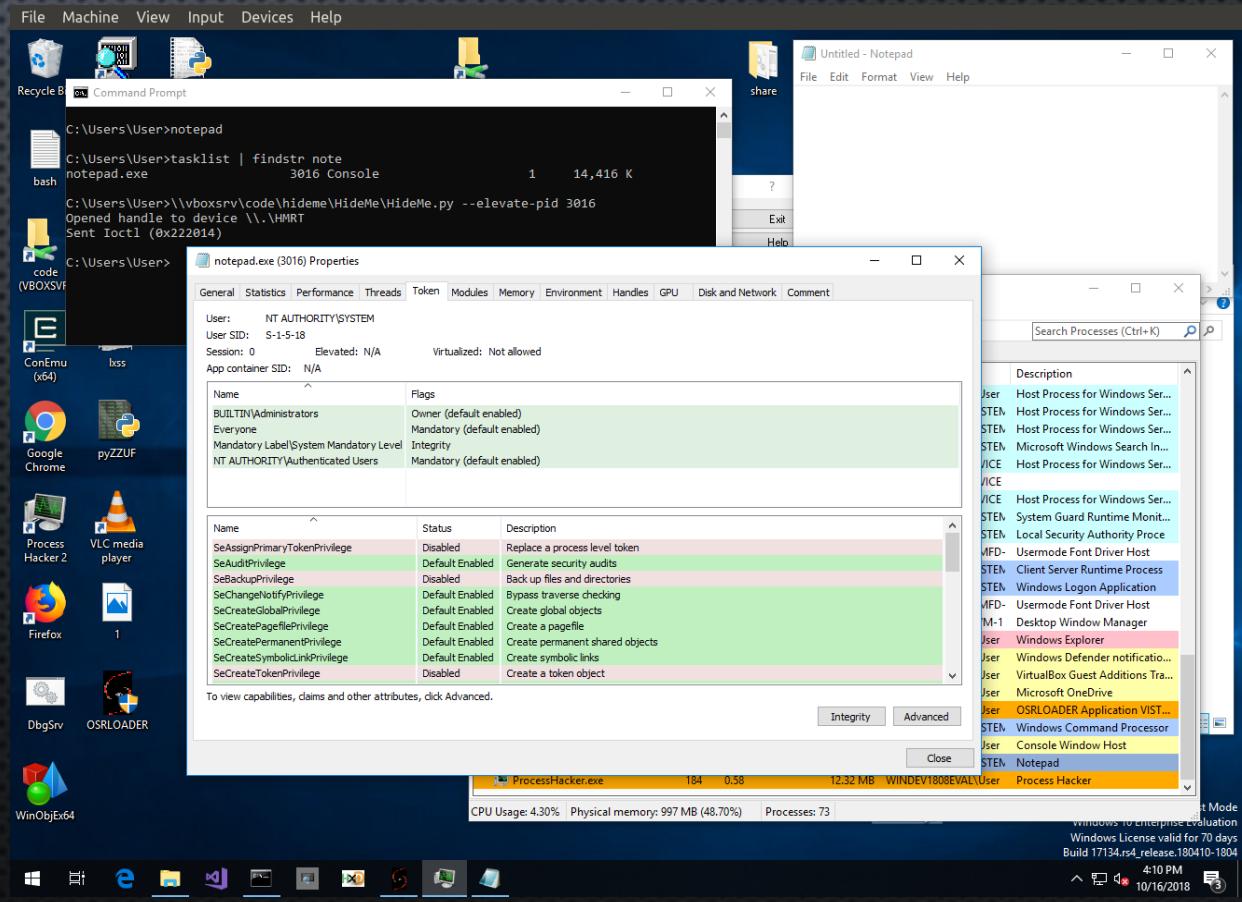
DEBUGGER DATA MODEL

- PRACTICAL CASE
  - HIDEME ROOTKIT: PROCESS HIDING
    - LOAD THE DRIVER AND START THE SERVICE
    - PICK ANY PROCESS AND RUN
      - C:\> PYTHON \HIDEME\HIDEME.PY --HIDE-PID [PID]
    - USING WINDBG `DX`, FIND THAT PROCESS



DEBUGGER DATA MODEL

- PRACTICAL CASE
  - HIDEME ROOTKIT: TOKEN ELEVATION
    - SAME AS BEFORE, LOAD THE DRIVER AND START THE SERVICE
    - PICK ANY PROCESS AND RUN
      - C:\> PYTHON \HIDEME\HIDEME.PY --ELEVATE-PID [PID]
    - USING WINDBG `DX`, FIND THAT PROCESS



(JAVA)SCRIPTING

WinDbg Scripting

- MASM Syntax
  - Allows to mix C/C++ syntax in WinDbg for parsing
    - ?? ((nt!_EPROCESS*)0x4141414142424242)->ImageFileName
- WinDbg Scripting language (.wds)
  - Succession of WinDbg native commands (like GDB scripting)
  - Allows to iterate through loops (.for), define function-like blocks (.block)
  - Invoked via "the angry rich man" : $$>< "Path\To\Script.wds"
- PyKd (not officially supported, nor built-in)
  - Powerful plugin that allows LibPython27.dll to run Python scripts inside WinDbg
  - https://githomelab.ru/pykd/pykd

WINDBG + JAVASCRIPT

- In 2017, WinDbg 10+ and Preview received a new provider
  - JsProvider.dll (along with ChakraCore.dll)
- Slow adoption by dev/infosec people (because it's JavaScript?)
- But actually pretty good
  - A "real" scripting language (!= WDS)
  - Straight built-in - No more hacks like PyKd
  - Backed by Chakra (actively maintained)

```
> typeof NaN
< "number"
> 9999999999999999
< 10000000000000000
> 0.5+0.1==0.6
< true
> 0.1+0.2==0.3
< false
> Math.max()
< -Infinity
> Math.min()
< Infinity
> []+[]
< ""
> []+{}
< "[object Object]"
> {}+[]
< 0
> true+true+true===3
< true
> true-true
< 0
> true==1
< true
> true===1
< false
> (!+[]+[]+![]).length
< 9
> 9+"1"
< "91"
> 91-"1"
< 90
> []==0
< true
```

WINDBG + JAVASCRIPT

- ChakraCore implements ECMAScript6 (2015)
  - Variable parameter number
  - Generators
  - Promises (async)
  - Block-scoped variables (let)
  - [...]
- Ensure that it is available
  - `.scriptproviders` should show JavaScript (.js)
- A "Hello / Goodbye World" would be as simple as this

```
"use strict";

const log = x => host.diagnostics.debugLog(`${x}\n`);

function invokeScript()
{
    log("Hello world from `invokeScript`");
}

function initializeScript()
{
    log("Hello world from `initializeScript`");
}

function uninitializeScript()
{
    log("Goodbye world from `uninitializeScript`");
}
```

- Because PrintHelloWorld() is exposed via DDM, it can also be called from WinDbg directly using `dx`

```
0:000> dx Debugger.State.<MyScriptName>.Contents.PrintHelloWorld()
```

WINDBG + JAVASCRIPT

- Inside the script, can define 3 "special" functions
  - initializeScript()
    - Always called when script is being loaded (`.scriptrun` or `.scriptload`)
  - invokeScript()
    - Called when invoking script with `.scriptrun`
  - uninitializeScript()
    - Called when unload the script with `.scriptunload`

```
1   var log = host.diagnostics.debugLog;
2
3   function invokeScript()
4   {
5       log("Hello world from `invokeScript`\n");
6   }
7
8   function initializeScript()
9   {
10      log("Hello world from `initializeScript`\n");
11  }
12
13  function uninitializeScript()
14  {
15      log("Goodbye world from `uninitializeScript`\n");
16  }
17
```

```
kd> .scriptload h:\windbg\scripts\javascript\hellogoodbyeworld.js
Hello world from `initializeScript`
JavaScript script successfully loaded from 'h:\windbg\scripts\javascript\hellogoodbyeworld.js'
kd> .scriptunload h:\windbg\scripts\javascript\hellogoodbyeworld.js
Goodbye world from `uninitializeScript`
JavaScript script unloaded from 'h:\windbg\scripts\javascript\hellogoodbyeworld.js'
kd> .scriptrun h:\windbg\scripts\javascript\hellogoodbyeworld.js
Hello world from `initializeScript`
JavaScript script successfully loaded from 'h:\windbg\scripts\javascript\hellogoodbyeworld.js'
Hello world from `invokeScript`
```

WINDBG + JAVASCRIPT

- `host` is the root namespace of all the WinDbg API exposed to JS
  - All defined in JsProvider.d.ts
    - %PROGRAMFILES%\Windows Kits 10\Debuggers\x64\winext\JsProvider.d.ts
  - Can be used for
    - Control WinDbg
      - host.namespace.Debugger.Utility.Control.ExecuteCommand(<Cmd>)
      - host.namespace.Debugger.Utility.Control.SetBreakpointAtOffset(<FuncName>, <lineNum>, [<ModName>])
    - Memory/Register access
      - host.currentThread.Registers.User (*.Rax, Rbx, …)
      - host.memory.readMemoryValues(<address>, <numElemt>, [<eltSz>, <isSigned>])
- DDM exposed directly
  - `dx Debugger` <-> `host.namespace.Debugger`
  - `dx @$scriptContents.host`
  - Get help ("man" style)
    - Ex. `dx Debugger.Utility.Control`
- And all can also be used from remote debugging

WINDBG + JAVASCRIPT

- The "Slight" problem 64b number representation in JavaScript
  - JS represents numbers on 53 bits
    - See standard IEEE 754 double-precision floating point
  - Native structures for 64b integer: class `host.int64`
    - Implemented a native host.Int64 (also available in 32b) class to manipulate addresses
      - `var a = host.parseInt("0x42", 16)`
      - `var b = host.int64(66)`
      - `if (a.compareTo(b) == 0) { …`
  - Always ensure you're using a correct class (int64 vs number) to avoid many inconsistences

WINDBG + JAVASCRIPT

- CAN EXPOSE JS FUNCTIONS TO DDM

```
62  /**
63   * Initialize the function alias.
64   */
65  function initializeScript()
66  {
67      return [
68          new host.functionAlias(greet, "greet"),
69          new host.functionAlias(GetEnvironmentVariables, "env")
70      ];
71  }
72
```

```
Command
0:001> dx -g @$env().Where(e => e.Contains("PRO") )
=================================================================================
=                                                                  = (+) Length =
=================================================================================
= [0x0] : ALLUSERSPROFILE=C:\ProgramData                           - 0x1e       =
= [0x1] : FPS_BROWSER_APP_PROFILE_STRING=Internet Explorer         - 0x30       =
= [0x2] : FPS_BROWSER_USER_PROFILE_STRING=Default                  - 0x27       =
= [0x3] : NUMBER_OF_PROCESSORS=1                                   - 0x16       =
= [0x4] : PROCESSOR_ARCHITECTURE=AMD64                             - 0x1c       =
= [0x5] : PROCESSOR_IDENTIFIER=Intel64 Family 6 Model 158 S...     - 0x48       =
= [0x6] : PROCESSOR_LEVEL=6                                        - 0x11       =
= [0x7] : PROCESSOR_REVISION=9e09                                  - 0x17       =
= [0x8] : PROMPT=$P$G                                              - 0xb        =
= [0x9] : USERDOMAIN_ROAMINGPROFILE=WINDEV1808EVAL                 - 0x28       =
= [0xa] : USERPROFILE=C:\Users\User                                - 0x19       =
=================================================================================
```

WINDBG + JAVASCRIPT

Action	Function	Examples
Print message	`host.diagnostics.debugLog`	
Read data from memory	`host.memory.readMemoryValues`	
Read string from memory	`host.memory.readString` `host.memory.readWideString`	
Evaluate expression	`host.evaluateExpression`	
Resolve symbol	`host.getModuleSymbolAddress`	`See DumpKnownTypes.js`
Dereference a pointer as an object	`host.createPointerObject(...).dereference()`	`See DumpKnownTypes.js`
Dereference memory	`host.evaluateExpression('(int*)0xADDRESS').dereference()`	
Convert string to Int64	`host.parseInt64('value')` `host.parseInt64('value', 16 )`	`host.parseInt64('42');` `host.parseInt64('0x1337', 16);`
Get access to the Pseudo-Registers	`host.namespace.Debugger.State.PseudoRegisters`	`host.namespace.Debugger.State.PseudoRegisters.General.exentry.address`


```
Using dx :
   @scriptContents.host <-> Debugger.State.Scripts.test.Contents.host
```

WINDBG + JAVASCRIPT

Action	Function
Execute WinDbg command	`host.namespace.Debugger.Utility.Control.ExecuteCommand`

Set Breakpoint	`host.namespace.Debugger.Utility.Control.SetBreakpointAtSourceLocation`
	`host.namespace.Debugger.Utility.Control.SetBreakpointAtOffset`
	`host.namespace.Debugger.Utility.Control.SetBreakpointForReadWrite`

Iterate through LIST_ENTRYs	`host.namespace.Debugger.Utility.Collections.FromListEntry`

- There is no native way (yet) to set callback on breakpoints correctly (`bp.Command` doesn't work as expected)
- @0vercl0k described a hack to circumvent that

```
function MyBreakpointHandlerInJS() {
    host.diagnostics.debugLog('Hello from JS callback!');
    […]
}

var BpAddr = 0x414141414141; // Or wherever you choose to break on
var Cmd = 'bp /w "@$scriptContents.MyBreakpointHandlerInJS()" ' + BpAddr.toString(16);
host.namespace.Debugger.Utility.Control.ExecuteCommand(Cmd);
```

WINDBG + JAVASCRIPT

- Some useful links
  - Official documentation
    - https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/native-objects-in-javascript-extensions
    - https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/javascript-debugger-example-scripts
  - Some JS Script Repo
    - https://github.com/microsoft/WinDbg-Samples
    - https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/javascript-debugger-example-scripts
    - https://github.com/0vercl0k/stuffz/tree/master/windbg-scripts
    - https://github.com/hugsy/windbg_js_scripts
  - Walk-through of a JS script development
    - https://www.osr.com/blog/2017/05/18/windbg-debugger-objects-javascript-oh/
    - https://doar-e.github.io/blog/2017/12/01/debugger-data-model/#x64-exception-handling-vs-javascript

HANDS ON

- Get familiar with JS API
  - Ideas :
    - Easy
      - Write a Hello World script
      - Make it a new command (Hint: google for "host.functionalias")
      - Add a small documentation via meta-data (Hint: google for "host.metadata.definemetadata")
    - Medium
      - Read / write values from registers / memory
      - Use DDM integration to play with the @$cursession
    - Advanced
      - Add JS breakpoint(s) to your script
        - Then add JS callback using @0ver's technique
      - Fiddle with the Code / FileSystem disassembly interface
      - Make it a Gallery from the hello-world script created before

HANDS ON

- Advanced practical cases:
  - Make the SSDT table a convenience variable in WinDbg
    - See \scripts\javascript\GetSsdtTable.js
  - Enumerate the root objects
    - See \scripts\javascript\ObjectExplorer.js
  - Trace functions called
    - See \scripts\javascript\TraceFunctions.js

TIME-TRAVEL DEBUGGING

TIME-TRAVEL DEBUGGING

- Entirely part of WinDbg Preview
  - Publicly released in October 2017 (< 2 years ago)
- Allows to rewind to given time of a process execution
- Works in 2 steps
  1. Running WinDbg Preview as Administrator to spawn or attach to a process
     - Records a trace of *everything* happening during the execution (instruction, memory access, I/O, etc.)
  2. When done, WinDbg opens the trace allowing to go back and forth through execution
     - Positive consequence: when replaying the trace, access is Read-Only
     - Negative consequence: program runs as Administrator
     - Useful for malware analysis
       - Sharing the trace is totally safe (no execution)
       - Helps for mutualizing access ("Jump to index XX:YY")
- Trace execution is also scriptable via C++ API

TIME-TRAVEL DEBUGGING

- Integrated with the Debugger Data Model (and JavaScript!)
  - "Man" page
    - `0:000> .prefer_dml 1 $$ optional - clicky clicky`
    - `0:000> dx @$curprocess.TTD`
    - `0:000> dx @$cursession.TTD`
  - Catch all the exceptions raised
    - `0:000> dx @$curprocess.TTD.Events.Where(t => t.Type == "Exception").Select(e => e.Exception)`
  - List all the modules loaded
    - `0:000> dx -g @$curprocess.TTD.Events.Where(t => t.Type == "ModuleLoaded").Select(m => m.Module.Name)`
  - Enumerate threads (over-)writing the PEB (process hollowing, process doppelganging, etc.)
    - `0:000> dx -g @$cursession.TTD.Memory( 0x<AddrPeb>, 0x<AddrPeb>+sizeof(_PEB), "w")`

TIME-TRAVEL DEBUGGING

- Practice Case: get unpacked code from a packed binary (traditional packer)
  1. The packer allocates RWE region
     - Get the return values of all the calls to VirtualAlloc() where permission is PAGE_EXECUTE_READWRITE
     - `0:000> dx -g @$cursession.TTD.Calls("Kernel*!VirtualAlloc*").Where( c => c.Parameters[3] == 0x40 ).Select( c => new { Address = c.ReturnValue , Size = c.Parameters[1]} )`
  2. Then jumps in it
     - `0:000> dx -g @$cursession.TTD.Memory( <AddressFromAbove>, <AddressFromAbove> + <SizeFromAbove>, "e")[0].TimeStart.SeekTo()`
  3. We can now dump it to disk for static analysis
     - `0:000> .writemem C:\Temp\MyDump.bin <Address> <Size>`

TIME-TRAVEL DEBUGGING

The context commands are the same, just need to add '-' at the end of the command

Action	Command	Example
Run execution back	g-	
Step Over back	p-	
Step Into back	t-	
Regenerate the index	!tt.index	
Jump to position XX:YY in execution	!ttexd.tt XX:YY (native) *.<Position>.SeekTo() (DDM)	!tt.position D:0 dx -r1 @$curprocess.TTD.Lifetime.MinPosition

Example:

Dump the arguments of the last call to memset() of the execution

```
0:000> bp ntdll!memset ".printf \"rcx=%p rdx=%p r8=%p\\n\", @rcx, @rdx, @r8"
0:000> g-
rcx=0000000000a6c50 rdx=0000000000000000 r8=0000000000000048
Time Travel Position: 12:89
```

TIME-TRAVEL DEBUGGING

- Known problems:
  - If present, EMET might kill the debugger when recording the trace
  - Records *all* of the activity -> trace can quickly become huge for complex binaries
    - Think of Web browsers or Office suite programs – need to target better when to start the recording
  - Kernel mode tracing is not possible
  - Cannot trace Protected Processes
  - Trace is immutable (by design)
    - Can't be used for fuzzing

TIME-TRAVEL DEBUGGING - DDM - MISC

```
0:000> dx -r1 @$curprocess.TTD
    Threads
    Events
    Lifetime          : [10:0, 1B6F:0]
    SetPosition       [Sets the debugger to point to the given position on this process.]

0:000> dx -r1 @$curprocess.TTD.Lifetime
    MinPosition       : 10:0 [Time Travel]
    MaxPosition       : 1B6F:0 [Time Travel]

0:000> dx -r1 @$curprocess.TTD.Events
    [0x0]             : Module ping.exe Loaded at position: 2:0
    [0x1]             : Module TTDRecordCPU.dll Loaded at position: 3:0
    [0x2]             : Module apphelp.dll Loaded at position: 4:0
    [0x3]             : Module IPHLPAPI.DLL Loaded at position: 5:0
    [0x4]             : Module KERNELBASE.dll Loaded at position: 6:0
          …
```

TIME-TRAVEL DEBUGGING - DDM - CALLS

```
0:000> dx @$cursession.TTD.Calls("msvcrt!write").First()
    EventType         : 0x0
    ThreadId          : 0x1e04
    UniqueThreadId    : 0x2
    TimeStart         : 1842:2455 [Time Travel]
    TimeEnd           : 18E9:14 [Time Travel]
    Function          : UnknownOrMissingSymbols
    FunctionAddress   : 0x7ffba0d5fb30
    ReturnAddress     : 0x7ffba0d873ce
    ReturnValue       : 0x9a
    Parameters
```

TIME-TRAVEL DEBUGGING - DDM - MEMORY

```
0:000> dx @$cursession.TTD.Memory(0x000000d538ecd000, 0x00000d538ed0000, "r").First()
    EventType         : 0x1
    ThreadId          : 0x1e04
    UniqueThreadId    : 0x2
    TimeStart         : 11:32 [Time Travel]
    TimeEnd           : 11:32 [Time Travel]
    AccessType        : Read
    IP                : 0x7ffba2c2e48d
    Address           : 0xd538ecf2d8
    Size              : 0x8
    Value             : 0x7ffba2be2d38
```

TIME-TRAVEL DEBUGGING - DDM - RESOURCES

- HeapMemory:

```
0:000> dx -g @$cursession.TTD.Resources.HeapMemory
=================================================================================================================================================
=          = (+) ResourceId  = (+) ResourceIdNew = (+) Size = (+) Function             = (+) ThreadId = (+) UniqueThreadId = (+) Position =
=================================================================================================================================================
= [0x0]     - 0x1fb7c577ae0    - 0x0               - 0x1000    - ntdll!RtlAllocateHeap    - 0x1e04       - 0x2                - 12A8:C       =
= [0x1]     - 0x1fb7c577000    - 0x0               - 0x38      - ntdll!RtlAllocateHeap    - 0x1e04       - 0x2                - 128E:318     =
= [0x2]     - 0x1fb7c56dde0    - 0x0               - 0x40      - ntdll!RtlAllocateHeap    - 0x1e04       - 0x2                - 1297:3E3     =
= [0x3]     - 0x1fb7c577cb0    - 0x0               - 0x38      - ntdll!RtlAllocateHeap    - 0x1e04       - 0x2                - 12BA:250     =
...
```

TIME-TRAVEL DEBUGGING - DDM - UTILITY

- GetHeapAddress:

```
0:000> dx -g @$cursession.TTD.Utility.GetHeapAddress(0x1fb7c9259c0).OrderBy(p => p.TimeStart)
=================================================================================================================================
=                          = Action  = Heap             = Address          = Size    = Flags = (+) TimeStart = (+) TimeEnd = Result =
=================================================================================================================================
= [0x20c] : [object Object] - Alloc    - 0x1fb7c920000    - 0x1fb7c9259c0    - 0xf06    - 0x0    - 4D:1E1A       - 4F:3F       -        =
= [0x205] : [object Object] - Free     - 0x1fb7c920000    - 0x1fb7c9259c0    -          - 0x0    - 57:464        - 59:35       - 0x1    =
=================================================================================================================================
```

HANDS ON

- C:\Windows\System32\notepad.exe
  - Record a session of running notepad
  - Try to leak the keys stroke from WinDbg
    - Hint : monitor messages received by `user32!GetMessageW`, and check `wintypes!MSG.wParam`

- \bin\Ransomware\MiniRansomware.run.7z
  - Dump the encryption password using TTD

- \bin\ProcessHollower\ProcessHollower.exe
  - Run
    - \bin\ProcessHollower\ProcessHollower.exe C:\Windows\System32\notepad.exe \bin\ProcessHollower\MessageBox-x64.exe
  - Detect the hollowing
    - PEB monitoring
    - CreateProcess(..., SUSPENDED, ...)

FINAL WORDS

KEY TAKE-AWAYS FROM THIS WORKSHOP

- STOP USING WINDOWS 7 (OR GOD FORBIDS, WINDOWS XP)
  - SOMEONE SAID CVE-2019-0708?
- STOP USING WINDBG LEGACY
  - AND/OR OTHER INFERIOR DEBUGGERS FOR THAT MATTER
- LINQ + DDM ARE REALLY GOOD
  - LET'S USE THEM!
- A LITTLE BIT OF JAVASCRIPT CAN GO A LONG WAY
  - ESPECIALLY WHEN DEBUGGING COMPLEX (STRIPPED) CODE

SOME MORE USEFUL RESOURCES

- Free Windows 10 VM with VS
  - https://developer.microsoft.com/en-us/windows/downloads/virtual-machines
- Extra Resources for WinDbg
  - What's new in Windbg preview? https://www.youtube.com/watch?v=sTJZPJVnVVU
  - Windbg Playlist - https://www.youtube.com/playlist?list=PLJAuO31RG973XOVDi5RXWlrC-XlPZelGn
  - https://doar-e.github.io/blog/2017/12/01/debugger-data-model/
  - https://blahcat.github.io/2018/11/02/some-time-travel-musings/
  - https://www.usenix.org/legacy/events/vee06/full_papers/p154-bhansali.pdf
  - https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/data-model-cpp-overview
  - https://blogs.msdn.microsoft.com/windbg/2016/10/03/writing-linq-queries-in-windbg/
  - https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/keyboard-shortcuts
- Community
  - https://github.com/microsoftfeedback/WinDbg-Feedback/
  - https://twitter.com/aluhrs13

THE END

@_hugsy_

@0vercl0k