

# **Tips & Tricks for Better Debugging**

## **with WinDbg**

Chris Alladoum

Security Software Engineer



hugsy

@\_hugsy\_

# ***Introduction***

# **Why**

# **WinDbg ?**

- The most advanced debugger for Windows
  - + Multi-architecture support (Intel, ARM, etc.)
  - + Multi-format support (PE - incl. dump, ELF - incl. dump)
  - + User & Kernel mode debugging
  - + Local & Remote debugging
  - + Thorough extension capabilities
    - Debug SDK defines API to build DLL
      - + From C++, even Rust!
    - JavaScript support
    - Plugin repositories (galleries)
  - + Comes with useful misc. tools (CDB, NTSD, DBGSRV, TTD, etc.)
- Since 2022, can be installed using `winget` (built-in since 22H2)

# **Why this workshop ?**

- WinDbg is fun to play with
  - + Debugging doesn't have to be a chore
  - + Many useful functions / commands have little to no documentation and/or exposure
  - + Several recent improvements change the way to debug entirely
- Started collecting tricks and sharing them on @windbgtips Twitter account
- This workshop comes in continuation of that
- Material is available online:
  - + Workshop repo on [GitHub](#) (hugsy/recon24\_windbg\_workshop)
  - + WinDbg cheatsheet can be found [here](#)

# **Workflow**

- Cover some known and (hopefully) lesser-known techniques
  - + Assume familiarity with WinDbg already
- We'll be operating mostly from KdNet session on Win11 (23H2)
  - + Need to quickly have a test environment? Use WindowsSandbox !
    - (opt. for network mgt) Enable-WindowsOptionalFeature -All -Online -LimitAccess - FeatureName Microsoft-Hyper-V
    - (opt) Enable-WindowsOptionalFeature -All -Online -FeatureName Containers-DisposableClientVM
    - CmDiag.exe DevelopmentMode -On
    - CmDiag.exe Debug -on -serial
    - OR CmDiag.exe Debug -on -net -hostip \$LocalHostIP -key 1.2.3.4 (faster)
    - windbgx \$output\_from\_previous\_cmd
  - + You can also use LKD on the same host/VM

# ***Tips & Tricks***

# ***for WinDbg***

# **Debugger Data Model & LINQ**

## **Tricks**

- Use `dx` - all the time, for everything 😊
  - Can replace `dt`, `x`, `?`, `bp/ba`, `.open`, `r` and more
  - Don't know where to start? Type `dx Debugger` (or even simply `d` - KdOnly)
  - `dx` also supports recursive display (-r), and grid display (-g)
  - Controls entirely TTD traces (`dx @\$curprocess.TTD` / `dx @\$cursession.TTD`)
- Can be used to query, map, filter sort data
  - + kd> dx @\$cursession.Processes.Where( p => ((char\*)p.KernelObject.ImageFileName).ToDisplayString("sb").StartsWith("S"))
- Allows to store variables and lambda functions
  - + dx @\$CurrentThreads = ( (x) => @\$cursession.Processes.Flatten( x => x.Threads ) )

# **Debugger Data Model & LINQ**

## **Tricks**

- Use cases - `dx`
  - + Dump all the GDT entries as a table
  - + Create data structure from a LIST\_ENTRY using `Debugger.Utility.Collections.FromListEntry`
    - ex. List Processes from `nt!KiProcessListHead` (which of type \_KPROCESS, though “ProcessListEntry” member)
  - + Represent raw pointer as an array of a specific type
    - ex. List Process Creation Callbacks from `nt!PspCreateProcessNotifyRoutine`, map into an object

# **Debugger Data Model & LINQ**

## **Tricks**

- Use cases - `dx`
  - + Dump all the GDT entries as a table
    - dx @gdtl ; dx -g (nt!\_KGDTENTRY64[\$n])@gdtr
  - + Create data structure from a LIST\_ENTRY using `Debugger.Utility.Collections.FromListEntry`
    - ex. List Processes from `nt!KiProcessListHead` (which of type \_KPROCESS, though “ProcessListEntry” member)
      - + dx -g Debugger.Utility.Collections.FromListEntry( \*(nt!\_LIST\_ENTRY\*)&(nt!KiProcessListHead), "nt!\_KPROCESS", "ProcessListEntry").Select( p => new {Process = (nt!\_EPROCESS\*)&p} )
  - + Represent raw pointer as an array of a specific type
    - ex. List Process Creation Callbacks from `nt!PspCreateProcessNotifyRoutine`, map into an object
      - + dx -g ((void\*[0x40])&nt!PspCreateProcessNotifyRoutine).Where( x => ((int)x) != 0).Select( x => x & ~0xf).Select(x => (void\*[3])x).Select( p => new { Address=p, Callback=p[1]})

# **Debugger Data Model & LINQ**

## **Tricks**

- Use DDM for conditional breakpoints
  - + `bp /w "DDM Boolean expression" \$Location`
    - Equivalent to
    - + `dx @\$d=Debugger.Utility.Control.SetBreakpointAtOffset("nt", "ZwOpenProcess"); dx @\$d.Condition = "@\$curprocess.Name == \"explorer.exe\""`
    - + Where the Boolean expression determines whether to break (if true) or not
      - ex. Break next time "explorer.exe" calls "nt!ZwCreateFile"
  - Can be used in conjunction with JS scripts
    - Exposes the JS `host` namespace - using `dx @\$host.<FunctionFromJsProvider.d.ts`
    - Exposes JS scripts function using `@\$scriptContent.<MyFunction>(\$func\_arg1, ...)`
      - + ex. List Process Creation Callbacks from before and show also symbol
        - Using `dx @\$scriptContent.host.getModuleContainingSymbolInformation( 0xaddress )`

# JavaScript scripting

## Tricks

- Prefer JS objects to plain logging
  - + Can be used together with WinDbg GUI (via DDM) to visualize data graphically
- Prefer Generators of objects (`\*function f() { .. yield}`) to `list/dict`
  - + Much faster
- Always use definitions from `JsProvider.d.ts`
  - + Enables signature completion in IDE

```
///<reference path="../extra/JSPrinter.d.ts" />
/// @ts-check
"use strict";
```



```
151 function invokeScript() {
152
153 let curproc = curprocess();
154
155
156
157 /**
```

# **JavaScript scripting**

## **Tricks**

- WinDbg embeds its own [Monaco IDE](#) (Scripting tab)
  - + Acts like a mini VSCode inside WinDbg
- `import` is not supported, so no library exists
  - + But template files can be used
    - [https://github.com/hugsy/windbg\\_js\\_scripts/blob/main/scripts/JsSkeleton.js](https://github.com/hugsy/windbg_js_scripts/blob/main/scripts/JsSkeleton.js)
- Demo !

# Galleries

- A Gallery is a repository of extensions (NatVis, JS, native) grouped together
  - + Uses an XML-declarative syntax to:
    - Declare the extensions part of the gallery
    - Define loading conditions (i.e. is kernel debugging?)
  - + Fairly unknown feature even though quite well documented on [GitHub](#)
  - + Load XML using `settings load path\to\config.xml`
  - + Save, to make WinDbg execute at restart
  - + Once in WinDbg, can be controlled via `dx`
    - dx -r1  
Debugger.State.ExtensionGallery.ExtensionRepositories
  - + Local gallery pre-declared:
    - \$env:AppLocalData\dbg\UserExtensions

```
Command X
kd> dx -r1 Debugger.State.ExtensionGallery.ExtensionRepositories
Debugger.State.ExtensionGallery.ExtensionRepositories
[0x0] : UserExtensions
[0x1] : hugsygallery
[0x2] : LocalInstalled

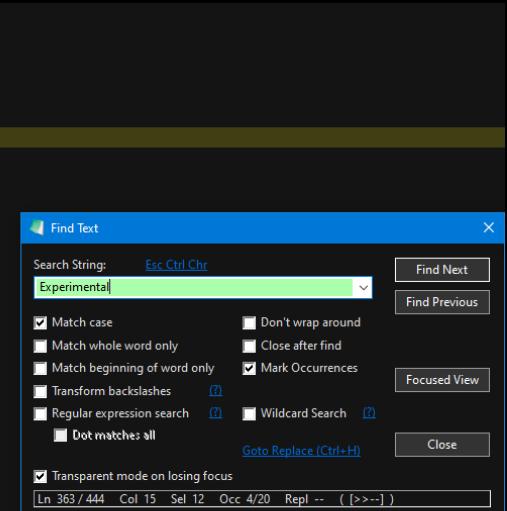
kd> dx -r1 Debugger.State.ExtensionGallery.ExtensionRepositories[1].Packages
Debugger.State.ExtensionGallery.ExtensionRepositories[1].Packages
[0x0] : EnumCallbacks
[0x1] : GetIdtGdt
[0x2] : BreakOnProcessCreate
[0x3] : DumpLookasides
[0x4] : GetSsdtTable
[0x5] : BigPool
[0x6] : VadExplorer
[0x7] : ObjectExplorer
[0x8] : RegistryExplorer
[0x9] : GetSiloMonitors
[0xa] : EnumApc
[0xb] : EnvVars
[0xc] : EnumImages
[0xd] : CallGraph
[0xe] : TraceFunctions
[0xf] : CyclicPattern
```

- User config files host show some unexposed settings
  - + %LOCALAPPDATA%\DBG\DebugX.xml
  - + %LOCALAPPDATA%\DBG\config.xml
- Ctrl+F -> “Experimental” 😊
- `dbghelp` exposes some intrinsic functions, undocumented but useful.
  - + Those functions can be invoked via `dx`
  - + Some examples:
    - Filtering functions
      - + \_\_iserror , \_\_ignoreerror , \_\_isnovalue
    - Comparison functions
      - + wcsnicmp, \_wcsicmp, \_stricmp, memicmp

```

<SourceWindowSettings>
  <Property name="AskBeforeSourceReload" value="false" />
  <Property name="DisableAutomaticSourceWindow" value="false" />
</SourceWindowSettings>
</XmlSetting>
<XmlSetting Name="RegistersExperimentalSettings">
  <RegistersExperimentalSettings>
    <Property name="UseOldRegistersWindow" value="false" />
  </RegistersExperimentalSettings>
</XmlSetting>
<XmlSetting Name="LogWindowExperimental">
  <LogWindowExperimental>
    <Property name="EnableNewLogWindow" value="false" />
  </LogWindowExperimental>
</XmlSetting>
<XmlSetting Name="DisassemblyExperimentalSettings">
  <DisassemblyExperimentalSettings>
    <Property name="DataModelDisassembly2" value="true" />
  </DisassemblyExperimentalSettings>
</XmlSetting>
<XmlSetting Name="DisassemblyWindow">
  <DisassemblyWindow>
    <Property name="MaxCodeBytes" value="15" />
    <Property name="ShowInstructionHints" value="true" />
  </DisassemblyWindow>
</XmlSetting>
<XmlSetting Name="CommandWindowExperimental">
  <CommandWindowExperimental>
    <Property name="CommandIntellisense2" value="false" />
  </CommandWindowExperimental>
</XmlSetting>
<XmlSetting Name="CommandWindow">
  <CommandWindow>

```



# Undocumented WinDbg

## Tricks

# Customizing WinDbg

## Tricks

### • Workspaces

- + Pure XML file but undocumented 🤓
- + Suffix with ` `.debugTarget`
- + Describe the debugging session to have!
  - Making it perfect for being automatically generated (think fuzz crash analysis for instance)
- + Open with ` -loadSession`
  - `> windbgX -loadsession \path\to\workspace.debugTargets`
- + Perfect for reproducing crashes (for example, fuzzing cases)

The screenshot shows the WinDbg interface. At the top, there's a code editor window displaying an XML configuration file for a workspace:

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <TargetConfig Name="Attach Service" LastUsed="2031-01-01T05:23:58.2908827Z" AccentColor="#FFFF0000">
3      <EngineConfig />
4      <EngineOptions>
5          <Property name="Elevate" value="true" />
6          <Property name="Restartable" value="false" />
7      </EngineOptions>
8      <TargetOptions>
9  
```

The main WinDbg window below has several tabs: Disassembly, Registers, and Memory. The Command tab is active, showing the output of a command:

```
***** Waiting for Debugger Extensions Gallery to Initialize *****
>>>>>>>> Waiting for Debugger Extensions Gallery to Initialize completed, duration 0.016 seconds
----> Repository : UserExtensions, Enabled: true, Packages count: 0
----> Repository : hugsygallery, Enabled: true, Packages count: 0
----> Repository : LocalInstalled, Enabled: true, Packages count: 41

Microsoft (R) Windows Debugger Version 10.0.27553.1004 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

*** wait with pending attach

***** Path validation summary *****
Response                      Time (ms)    Location
Deferred                         0           srv*\desktop-chris\symbols*https://msdl.microsoft.com/download/sy
Symbol search path is: srv*\desktop-chris\symbols*https://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 00007fff`1a8f0000 00007fff`1a901000  C:\WINDOWS\system32\svchost.exe
ModLoad: 00007fff`7ef30000 00007fff`7f128000  C:\WINDOWS\SYSTEM32\tdll.dll
ModLoad: 00007fff`7d090000 00007fff`7d14d000  C:\WINDOWS\System32\KERNEL32.DLL
ModLoad: 00007fff`7e0b0000 00007fff`7e0b6000  C:\WINDOWS\System32\USER32.DLL
```

# WinDbg SDK

## Tricks

- Simpler SDK using Modern C++ (+ WIL/CMake)
  - + Using [Microsoft::WIL](#) considerably simplifies the lifetime management of COM objects (with >= C++20)
  - + Using [CMake](#) considerably simplify the integration of Microsoft::WIL
  - + Result: Build safe native `DbgEng` extension in minutes
    - Including [complete DDM integration](#)

```
20 extern "C" void CALLBACK my_ext(PDEBUG_CLIENT Client, PCSTR args)
21 {
22     HRESULT hr = S_OK;
23     IDebugClient* debugClient = nullptr;
24     IDebugControl* debugControl = nullptr;
25
26     if ((hr = Client->QueryInterface(__uuidof(IDebugClient), (void**)&debugClient)) != S_OK)
27     {
28         return;
29     }
30
31     if ((hr = debugClient->QueryInterface(__uuidof(IDebugControl), (void**)&debugControl)) != S_OK)
32     {
33         debugClient->Release();
34         return;
35     }
36
37     debugControl->Output(DEBUG_OUTPUT_NORMAL, "hello from my_ext\n");
38
39     debugControl->Release();
40     debugClient->Release();
41 }
```



```
31 extern "C" void CALLBACK my_ext(PDEBUG_CLIENT Client, PCSTR args)
32 {
33     auto wil::com_ptr<...> cli = wil::com_ptr(Client);
34     if (!cli)
35         return;
36
37     auto wil::com_ptr<...> dbg = cli.try_query<IDebugClient>();
38     if (!dbg)
39         return;
40
41     auto wil::com_ptr<...> ctl = dbg.try_query<IDebugControl>();
42     if (!ctl)
43         return;
44
45     ctl->Output(Mask: DEBUG_OUTPUT_NORMAL, Format: "hello from my_ext\n");
46 }
```

Traditional COM  
Complex, verbose, memory-leak prone

Modern C++ style using WIL/COM  
Clean, safe

# WinDbg SDK

## Tricks

- Simpler SDK using Rust
  - + Crate `[dbgeng-rs](#)` provides a Rust implementation for building COM Client from `IDebugClient`
    - Can build a DLL library quickly
      - + cargo new --lib my\_windbg\_ext
      - + cargo add dbgeng@0.1
      - + cargo add windows @0.52 --features Win32\_Foundation,Win32\_System,Win32\_System\_Diagnostics,Win32\_System\_Diagnostics\_Debug,Win32\_System\_SystemInformation
  - Still a WIP

```
8 #[no_mangle]
9 extern "C" fn DebugExtensionInitialize(_version: *mut u32, _flags: *mut u32) → HRESULT {
10     S_OK
11 }
12
13 #[no_mangle]
14 extern "C" fn DebugExtensionUninitialize() {}
15
16 #[no_mangle]
17 extern "C" fn my_ext(raw_client: RawIUnknown, _args: PCSTR) → HRESULT {
18     let Some(client: &IUnknown) = (unsafe { IUnknown::from_raw_borrowed(&raw_client) }) else {
19         return E_ABORT;
20     };
21
22     let Ok(dbg: DebugClient) = DebugClient::new(client) else {
23         return E_ABORT;
24     };
25
26     let _ = dprintln!(dbg, "running `my_ext`");
27
28     S_OK
29 }
```

# And more...

- And more tricks there was no time to cover:
  - + TTD is a gold mine!
  - + Use [SyntheticTypes](#) to import custom structures (C header file) into WinDbg
  - + Automatically map drivers using ` `.kdfiles`
  - + DDM has many more useful features:
    - ` Debugger.Utility.Code` module (programmatically disassemble code)
    - ` Debugger.Utility.FileSystem` module (programmatically access files)
  - + WinDbgX now supports GDB protocol, allowing remote Linux debugging to host running gdbserver

The collage includes:

- A screenshot of a Twitter post by Andy Luhrs (@aluhrs13) about a new C++ library for extending and consuming the debugger data model. It shows a GitHub link and a NuGet package link.
- A screenshot of a Twitter post by Tim Misiak (@timmisiak) with a WinDbg tip: holding down Ctrl and double-clicking on text in the command window highlights all instances of that text across all windows.
- A screenshot of the WinDbg interface showing the Command window with assembly instructions and the Disassembly window below it.
- A screenshot of a GitHub post by William R. Messmer (@wmessmer) about using the GDB server with WinDbgX. It shows the WinDbgX interface with a list of processes and the GDB server configuration.

# *Final Words*

# Conclusion

- WinDbg is really fun!
  - + Easy to stick to the “old” style commands
  - + But made (a lot) more powerful through
    - DDM & LINQ
    - Customization
      - + NatVis / JS / Native extensions
      - + Workspaces & Galleries
    - Side tools (dbgsrv, kdnet, ttd are amazing)
  - + `dx` alone transforms the way we usually debug

***Thank you for attending!***

***Enjoy REcon!***

Feel free to contact me:

GH: hugsy

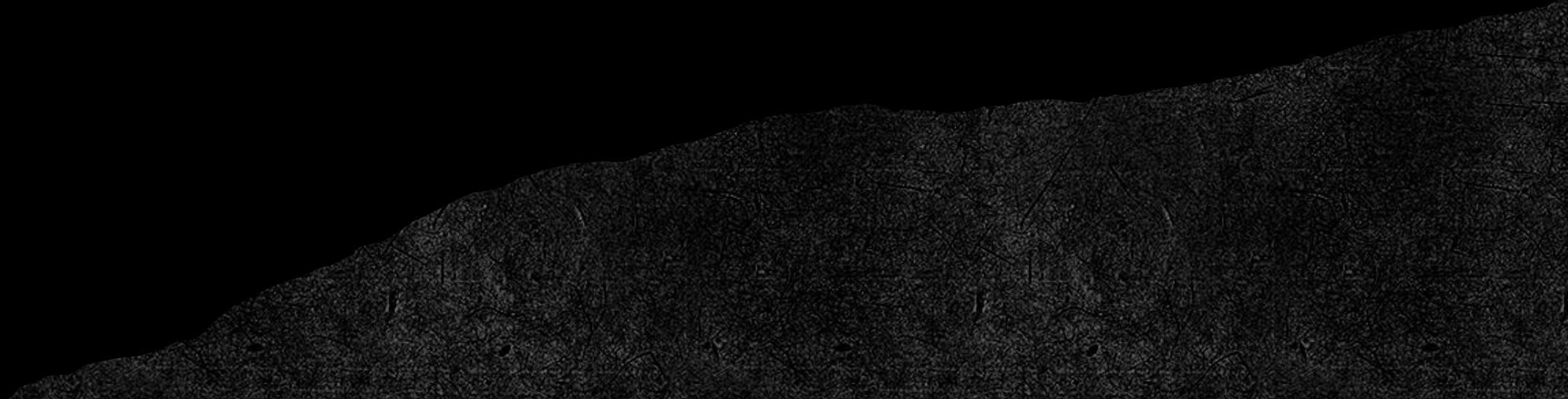
TW: @\_hugsy\_ (or @windbgtips to share tips 😊)



**Bonus**

# **Challenge**

Discover the message draw in the trace



# **Challenge**

Discover the message draw in the trace

Hint 1: Filter calls to `user32!GetMessageW`

# Challenge

Discover the message draw in the trace

Hint 1: Filter calls to `user32!GetMessageW`

Hint 2: Check for WM\_MOUSEMOVE in the output message as `wintypes!MSG`