# 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/JSProvider.d.ts" />
///
/// @ts-check
///
"use strict";
```

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

Id
Modules
Name
Threads

(property) processInterface.Id: nu ×
mber

The process identifier (PID) for the process

# JavaScript scripting Tricks

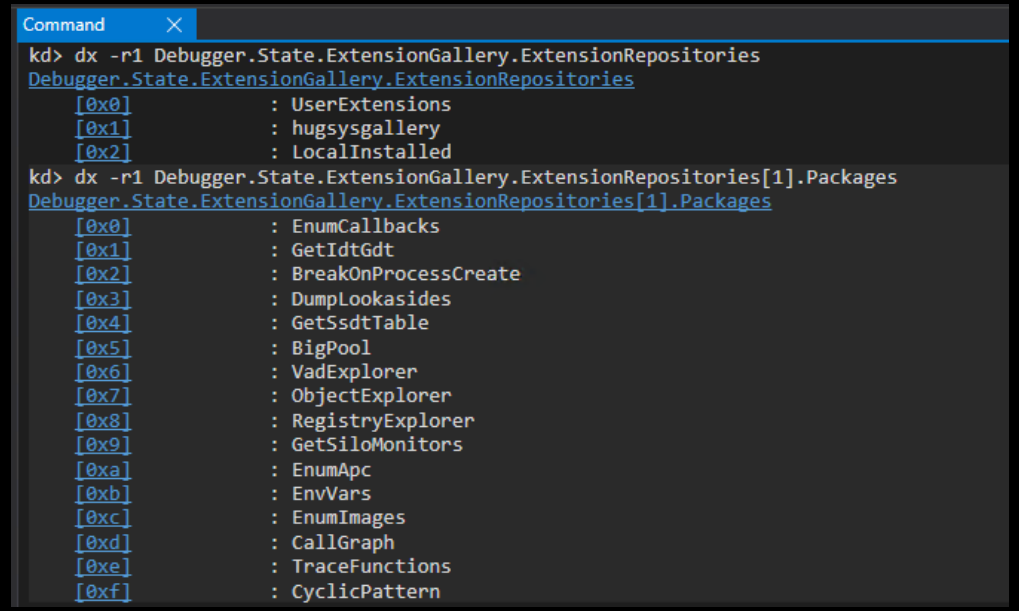
- WinDbg embeds its own Monaco IDE (Scripting tab)
  + Acts like a mini VSCode inside WinDbg

- `import` is not supported, so no library exists
  + But template files be can (re-)used
    - https://github.com/hugsy/windbg_js_scripts/blob/main/scripts/JsSkeleton.js

- Demo !

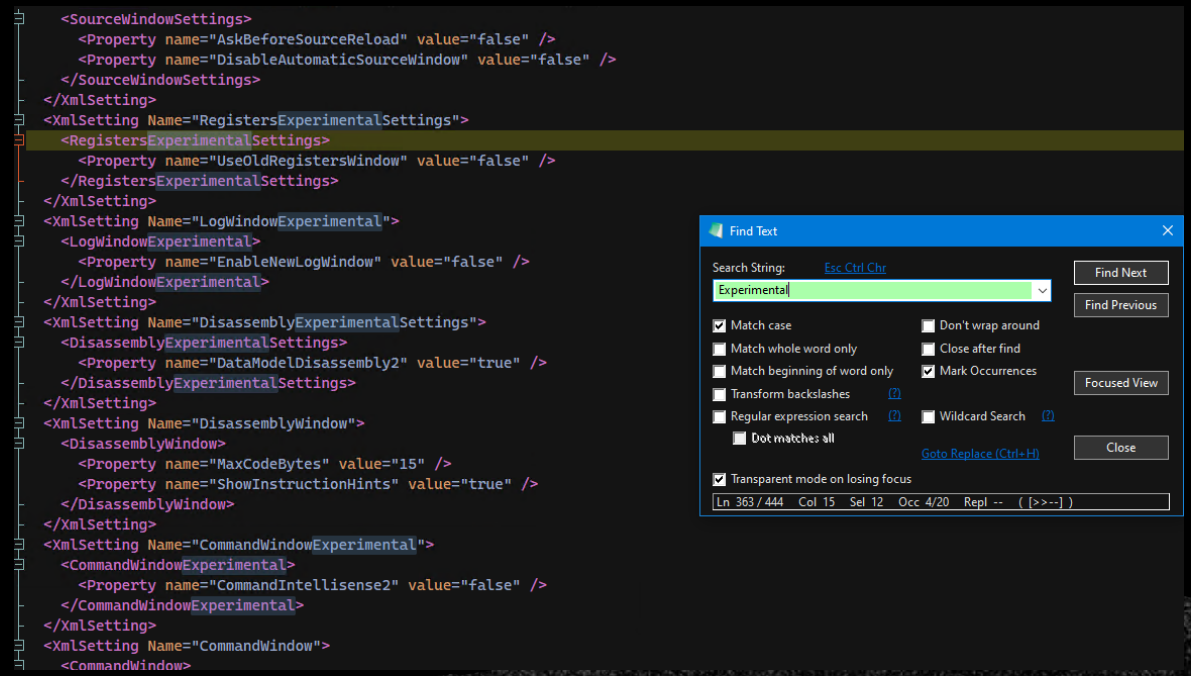
# Galleries

- 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



- User config files host show some unexposed settings
  - + %LOCALAPPDATA%\DBG\DbgX.xml
  - + %LOCALAPPDATA%\DBG\config.xml
- Ctrl+F -> “Experimental” 😄

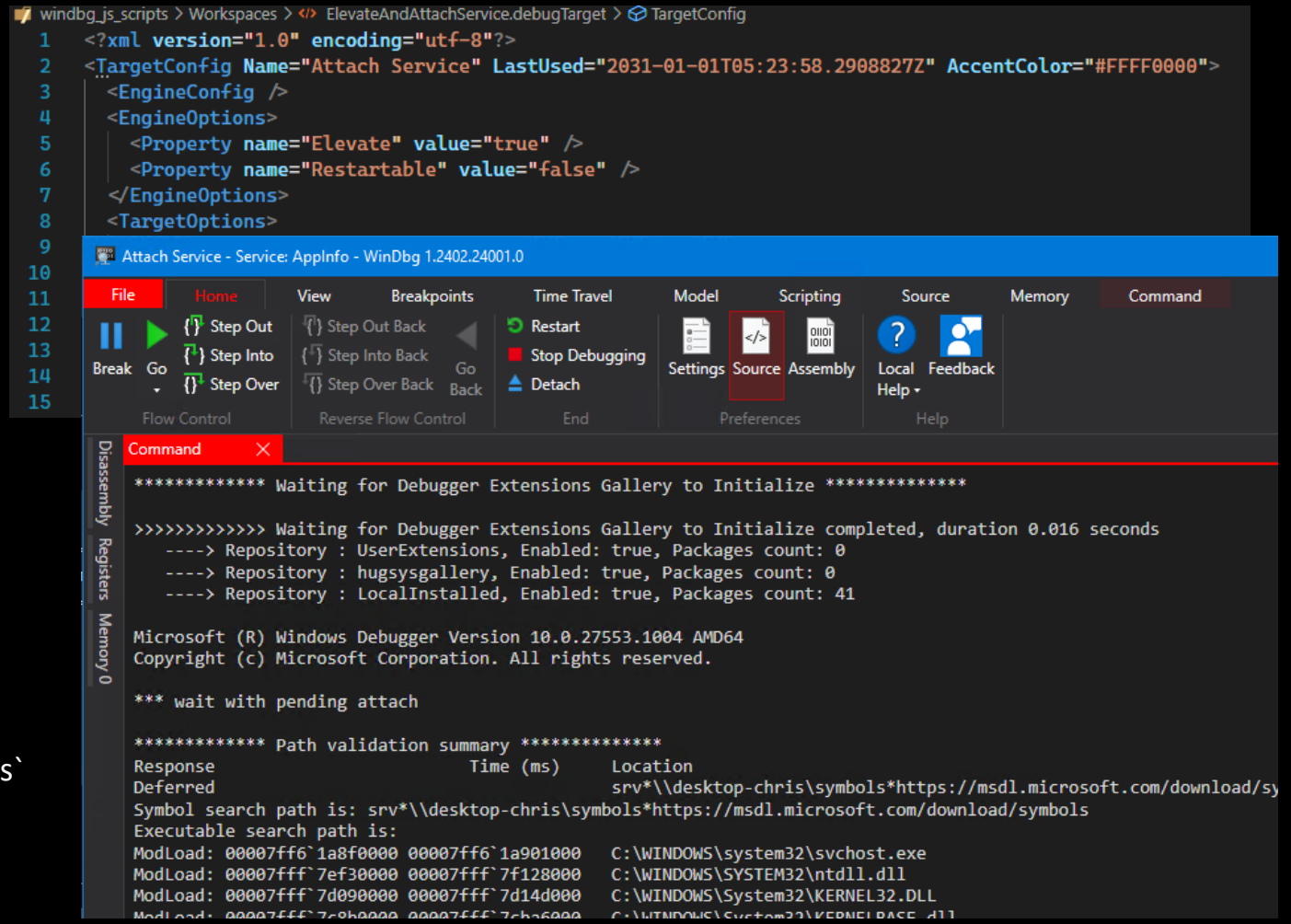
- `dbghelp` exposes some intrinsic functions, undocumented but useful.
  - + Those functions can be invoked via `dx`
  - + Some examples:
    - Filtering functions
      - + __iserror , __ignoreerror , __isnovalue
    - Comparison functions
      - + wcsnicmp, _wcsicmp, _stricmp, memicmp




# Undocumented WinDbg Tricks

# Customizing WinDbg Tricks

- **Workspaces**
  - + Pure XML file but undocumented 😳
  - + Suffixed 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)



# 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

```cpp
extern "C" void CALLBACK my_ext(PDEBUG_CLIENT Client, PCSTR args)
{
    HRESULT hr = S_OK;
    IDebugClient* debugClient = nullptr;
    IDebugControl* debugControl = nullptr;

    if ((hr = Client->QueryInterface(__uuidof(IDebugClient), (void**)&debugClient)) != S_OK)
    {
        return;
    }

    if ((hr = debugClient->QueryInterface(__uuidof(IDebugControl), (void**)&debugControl)) != S_OK)
    {
        debugClient->Release();
        return;
    }

    debugControl->Output(DEBUG_OUTPUT_NORMAL, "hello from my_ext\n");

    debugControl->Release();
    debugClient->Release();
}
```

Traditional COM
Complex, verbose, memory-leak prone

```cpp
extern "C" void CALLBACK my_ext(PDEBUG_CLIENT Client, PCSTR args)
{
    auto wil::com_ptr_ ... ption_policy> cli = wil::com_ptr(Client);
    if ( !cli )
        return;

    auto wil::com_ptr_ ... ption_policy> dbg = cli.try_query<IDebugClient>();
    if ( !dbg )
        return;

    auto wil::com_ptr_ ... ption_policy> ctl = dbg.try_query<IDebugControl>();
    if ( !ctl )
        return;

    ctl->Output(Mask: DEBUG_OUTPUT_NORMAL, Format: "hello from my_ext\n");
}
```

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

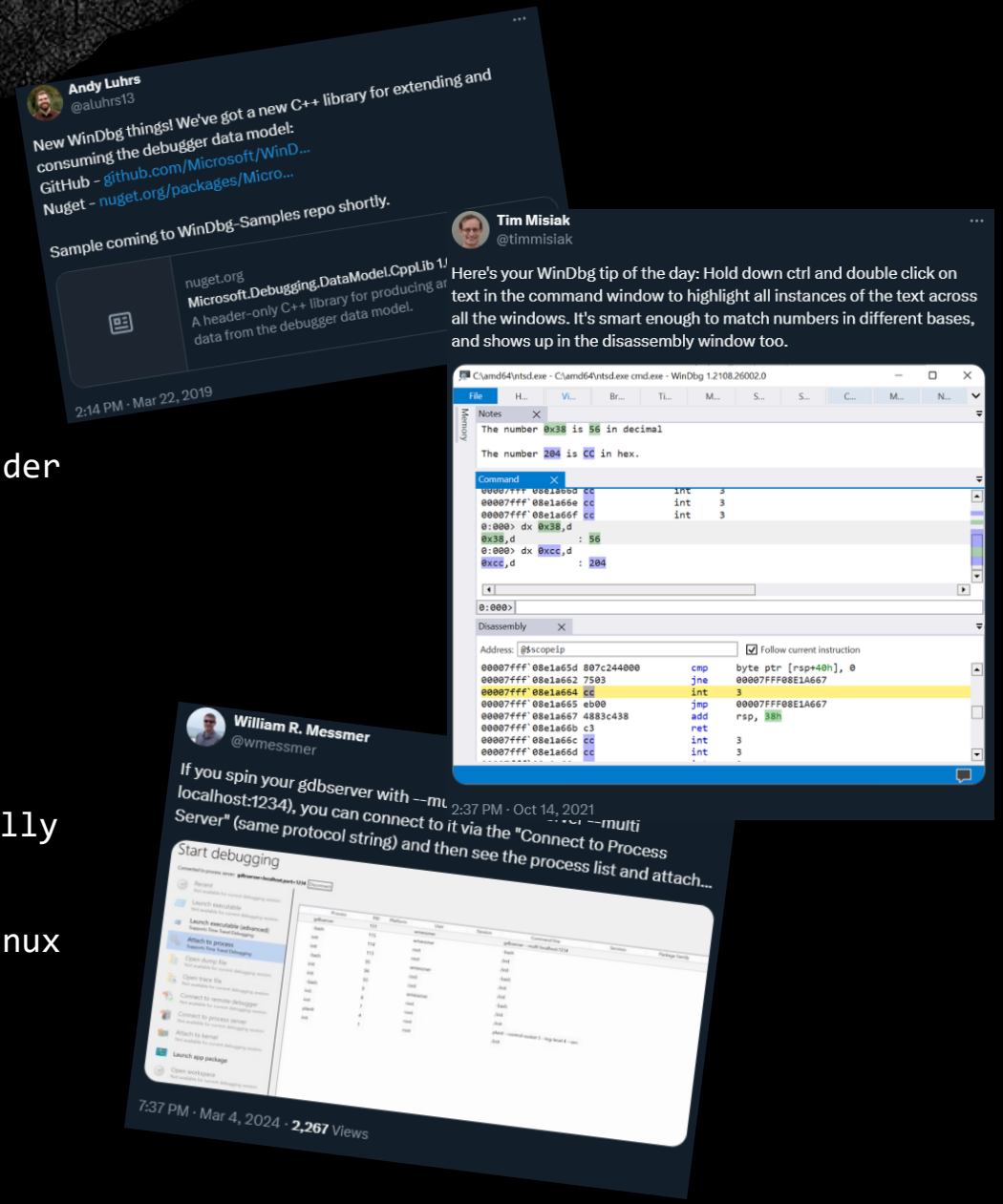
# WinDbg SDK Tricks

- 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

```rust
#[no_mangle]
extern "C" fn DebugExtensionInitialize(_version: *mut u32, _flags: *mut u32) -> HRESULT {
    S_OK
}

#[no_mangle]
extern "C" fn DebugExtensionUninitialize() {}

#[no_mangle]
extern "C" fn my_ext(raw_client: RawIUnknown, _args: PCSTR) -> HRESULT {
    let Some(client: &IUnknown) = (unsafe { IUnknown::from_raw_borrowed(&raw_client) }) else {
        return E_ABORT;
    };

    let Ok(dbg: DebugClient) = DebugClient::new(client) else {
        return E_ABORT;
    };

    let _ = dlogln!(dbg, "running `my_ext`");

    S_OK
}
```

# 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



# 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`