
DbgShell

A PowerShell front-end for the Windows debugger engine.

Ready to tab your way to glory? For a quicker intro, take a look at Getting Started.



Disclaimers

1. This project is not produced, endorsed, or monitored by the Windows debugger team. While the debugger team welcomes feedback about their API and front ends (windbg, kd, et al), they have no connection with this project. Do not file bugs or feedback to the debugger team concerning this project.
2. This is not a funded project: it has no official resources allocated to it, and is only worked on by volunteers. Do not take any production dependency on this project unless you are willing to support it completely yourself. Feel free to file Issues and submit Pull Requests, but understand that with the limited volunteer resources, it may be a while before your submissions are handled.
3. This is an experimental project: it is not fully baked, and you should expect breaking changes to be made *often*.

Corollary of above disclaimers: I would avoid attaching DbgShell to live targets of high value.

Binaries

<https://aka.ms/dbgshell-latest>

Motivation

Have you ever tried automating anything in the debugger? (cdb/ntsd/kd/windbg) How did that go for you?

The main impetus for DbgShell is that it's just waaaay too hard to automate anything in the debugger. There *are* facilities today to assist in automating the debugger, of course. But in my opinion they are not meeting people's needs. * Using the built-in scripting language is arcane, limited, difficult to get right, and difficult to get help with. * Writing a full-blown debugger extension DLL is very powerful, but it's a significant investment—way too expensive for solving quick, “one-off” problems as you debug random, real-world problems. Despite the cost, there are a large number of debugger extensions

in existence. I think there should not be nearly so many; I think the only reason there are so many is because there aren't viable alternatives. * Existing attempts at providing a better interface (such as PowerDbg) are based on "scraping" and text parsing, which is hugely limiting (not to mention ideologically annoying) and thus are not able to fulfill the promise of a truly better interface (they are only marginally better, at best). * Existing attempts to provide an easier way to write a debugger extension are merely a stop-gap addressing the pain of developing a debugger extension; they don't really solve the larger problem. (for instance, two major shortcomings are: they are still too low-level (you have to deal with the dbgeng COM API), and there's no REPL) * The debugger team has recently introduce Javascript scripting. Javascript is a much better (and more well-defined) language than the old windbg scripting language, but I think that PowerShell has some advantages, the largest of which is that nobody really uses a Javascript shell—PowerShell is much better as a combined shell *and* scripting language.

The goal of the DbgShell project is to bring the goodness of the object-based PowerShell world to the debugging world. When you do 'dt' to dump an 'object', you should get an *actual* object. Scripting should be as easy as writing a PowerShell script.

The DbgShell project provides a PowerShell front-end for dbgeng.dll, including: * a managed "object model" (usable from C# if you wished), which is higher-level than the dbgeng COM API, * a PowerShell "navigation provider", which exposes aspects of a debugging target as a hierarchical namespace (so you can "cd" to a particular thread, type "dir" to see the stack, "cd" into a frame, do another "dir" to see locals/registers/etc.), * cmdlets for manipulating the target, * a custom PowerShell host which allows better control of the debugger CLI experience, as well as providing features not available in the standard powershell.exe host (namely, support for text colorization using ANSI escape codes (a la ISO/IEC 6429))

The custom host is still a command-line (conhost.exe-based) program (analogous to ntsd/cdb/kd), but it can be invoked from windbg (![DbgShell](#)).

In addition to making automation much easier and more powerful, it will address other concerns as well, such as ease of use for people who don't have to use the debuggers so often. (one complaint I've heard is that "when I end up needing to use windbg, I spend all my time in the .CHM")

For seasoned windbg users, on the other hand, another goal is to make the transition as seamless as possible. So, for instance, the namespace provider is not the only way to access data; you can still use traditional commands like "~3 s", "k", etc.

What do you mean by "automation" and "scripting"?

I'm not *only* talking about the sort of thing where you open up a text editor and write some big script to do something complex—I'm *also* talking about being able to whip out relatively simple stuff directly

on the command line. There are many situations where you would like to be able to use a little bit of logic, but nothing so big or re-usable that you would even want to save it. It should be easy to just whip off “one-liners” like “break on CreateFile if the file being opened is on the user’s desktop and function Blah is on the stack.”

Why PowerShell?

Let me be clear: it took me approximately 4 years to “warm up” to PowerShell. I feel it has sharp edges, aspects that are just plain difficult, and plenty of bugs, both in design and implementation. Sometimes it really irritates me. *However*, the benefits of PowerShell are compelling, and have convinced me that it’s the best thing to use for this project:

- It is both a scripting environment and a CLI environment. The fact that it has to do both leads to some negative things like a steeper learning curve, but in the end it is extremely handy, because you want to be able to both do stuff quickly in a command-line REPL, as well as write full-featured, robust scripts.
- It is very discoverable—things like `Get-Command`, tab completion, the ability to expose hierarchical data like a filesystem, the facilities for providing and synthesizing help, are very good.
- Tab completion. I know I mentioned it in the previous bullet, but it’s awesome enough to get its very own bullet.
- The object pipeline: the object-oriented nature of the PowerShell pipeline is so much more powerful and easy to use than the bad old days of string-parsing-based scripting that it’s not even funny. Imagine doing “`dt`” to “dump” an “object”, and actually getting an object. `DbgShell` does that.
- People know it: I estimate that the number of people who know PowerShell and/or C# is at least a handful of orders of magnitude larger than the people who know windbg scripting techniques. That means more people will be able to easily “pick up” a PowerShell-based debugger; and it also means that when people need help, the pool of potential helpers is much larger (for scripting-related issues, anyway).
- PowerShell is still a general-purpose shell: when using `DbgShell`, you have access to not just debugger commands, but you can “`cd`” over to the filesystem, registry, AD, etc.; you can execute `Send-MailMessage`, `Get-WmiObject`, `Invoke-WebRequest`, `Invoke-RestMethod`, run arbitrary programs, etc.

Current Status

`DbgShell` has been in “prototyping mode” for a long time. I have spent a lot of time figuring how something could or should be done, but not necessarily “finishing” everything. There are a huge number

of TODOs in the current code. So although it has started to become actually useful, the project is still pretty green. However, it can definitely demonstrate enough to give you a good taste of what it should be like.

Below are some screenshots. It's important to note that nothing you see is dbgeng text output. Although some stuff in the output will look familiar, that is only because I have used PowerShell's formatting and output features to customize how certain objects are displayed—all the output you see actually corresponds to real, full .NET objects. For instance, those ModLoad messages each correspond to a `MS.Dbg.ModuleLoadedEventArgs` object, which has more properties than what get displayed when sent to `Out-Default`. There is no string parsing of anything from dbgeng whatsoever. (Well... almost. I've made a few compromises where there is no other way to get information. For instance, disassembly stuff, or parsing the symbolic name of an adjustor thunk function to find the offset.)

This is a sort of “hello world” scenario: attaching to an instance of cmd.exe. I first use the PowerShell built-in command `Start-Process`, then pipe the output to the DbgShell command `Connect-Process`, and then poke around the namespace:

```

C:\Projects\DbgShell\DbgShell\bin\Debug\x86\DbgShell.exe
Microsoft Debugger DbgShell
Copyright (c) 2015

Welcome.

Note that script execution policy is 'Bypass' for this process.
Run 'Get-Help about_DbgShell_GettingStarted' to learn about DbgShell.

PS Dbg:\
> Start-Process cmd.exe -PassThru | Connect-Process
ModLoad: 76e80000 7700e000 ntdll
ModLoad: 76340000 76410000 KERNEL32
ModLoad: 75830000 759f2000 KERNELBASE
ModLoad: 76080000 7613d000 msvcrt
(3d2c.75d0): Break instruction exception - code 80000003 (first chance)
ntdll!LdrpDoDebuggerBreak+0x2b

eax=00000000 ebx=00c4a000 ecx=cafc0000 edx=00000000 esi=00f72758 edi=76e86bac
eip=76f2db8b esp=00eff258 ebp=00eff284 iopl=0   TBD: flags
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!LdrpDoDebuggerBreak+0x2b:
76f2db8b cc                int     3

PS Dbg:\cmd (15660)\Threads\0
> dir

Directory: \cmd (15660)\Threads\0

Container  Name
-----
[container] Stack
TebAddress : 0xc4d000
Teb        : _TEB: (InitialThread) LastError: 32 LastStatus: 80070032 OwnedLocks: 0
IsCurrentThread : True
IsEventThread  : True

PS Dbg:\cmd (15660)\Threads\0
> dir .\Stack\

Directory: \cmd (15660)\Threads\0\Stack

Container  Name
-----
[container] 0 ntdll!LdrpDoDebuggerBreak+0x2b
[container] 1 ntdll!LdrpInitializeProcess+0x1967
[container] 2 ntdll!LdrpInitialize+0x180
[container] 3 ntdll!LdrInitializeThunk+0x1c

PS Dbg:\cmd (15660)\Threads\0
> cd '.\Stack\2 ntdll!_LdrpInitialize+0x180' ; dir

Directory: \cmd (15660)\Threads\0\Stack\2 ntdll!_LdrpInitialize+0x180

Container  Name
-----
InstructionPointer : 0x76ed8a42
ReturnAddress     : 0x76ed886c
StackPointer      : 0xeff4cc
IsFrameInline     : False
SymbolName        : ntdll!_LdrpInitialize
Module            : ntdll 76e80000 7700e000
Function          : ntdll!_LdrpInitialize
Displacement      : 0x180
RegisterSet       : eax=00000000 ebx=ffffffff ecx=cafc0000 edx=00000000 esi=00000000 edi=000000...
Locals            : 0 items:

PS Dbg:\cmd (15660)\Threads\0\Stack\2 ntdll!_LdrpInitialize+0x180
>

```

Here I have attached to a test program, and looked at the stack, switched to a particular stack frame, dumped locals, inspected the value of a local `std : map`, and inspected some type information for a local enum value. Note the display of the enumeration value: not only does DbgShell handle looking

up the symbolic name for single enumerands, but also when multiple enumerands are OR'ed together. You can't tell this from the screenshot, but there is tab completion for all of this stuff.

```
C:\Projects\DbgShell\DbgShell\bin\Debug\x86\DbgShell.exe
),std::vector<std::basic_string<wchar_t,std::char_traits<wchar_t>,std::allocator<wchar_t> >,std::allocator<std::basic_string<wchar_t,std::char_traits<wchar_t>,std::allocator<wchar_t> > > > > >+0x1c
06 TestNativeConsoleApp!std::_Invoker_ret<int,0>::__Call<int (__cdecl*)(std::vector<std::basic_string<wchar_t,std::char_traits<wchar_t>,std::allocator<wchar_t> >,std::allocator<std::basic_string<wchar_t,std::char_traits<wchar_t>,std::allocator<wchar_t> > > > > > > > > >+0x1c
07 TestNativeConsoleApp!std::_Func_impl_no_alloc<int (__cdecl*)(std::vector<std::basic_string<wchar_t,std::char_traits<wchar_t>,std::allocator<wchar_t> >,std::allocator<std::basic_string<wchar_t,std::char_traits<wchar_t>,std::allocator<wchar_t> > > > > > > > > >+0x1c
08 TestNativeConsoleApp!std::_Func_class<int,std::vector<std::basic_string<wchar_t,std::char_traits<wchar_t>,std::allocator<wchar_t> >,std::allocator<std::basic_string<wchar_t,std::char_traits<wchar_t>,std::allocator<wchar_t> > > > > > > > > >+0x24
09 TestNativeConsoleApp!wmain+0x3ed
0a TestNativeConsoleApp!invoke_main+0x1e
0b TestNativeConsoleApp!__scrt_common_main_seh+0x150
0c TestNativeConsoleApp!__scrt_common_main+0xd
0d TestNativeConsoleApp!wmainCRTStartup+0x8
0e KERNEL32!BaseThreadInitThunk+0x24
0f ntdll!__RtlUserThreadStart+0x2f
10 ntdll!__RtlUserThreadStart+0x1b

PS Dbg:\testApp\Threads\0
> .frame 9
PS Dbg:\testApp\Threads\0\Stack\9 TestNativeConsoleApp!wmain+0x3ed
> dv

Index Name Type Location Size Value
-----
0 numArgs Int4B 002cf9bc 0x4 7
1 args WCHAR** 002cf9c0 0x4 00622b48 -> "C:\Projects\DbgShell\DbgShe...
2 blah Int4B 002cf994 0x4 0n42
3 f Foo<int &> 002cf98b 0x1 Foo<int &>
4 i Int4B 002cf860 0x4 6
5 iter std::_Tree_iterator<st... 002cf810 0xc std::_Tree_iterator<std::_Tree_val<std::...
6 rc Int4B 002cf9a0 0x4 0
7 rm std::map<std::basic_st... 002cf974 0xc std::_Tree_val<std::_Tree_simple_types<s...
8 routine std::function<int __cd... 002cf868 0x28 std::function<int __cdecl(std::vector<st...
9 routineArgs std::vector<std::basic... 002cf898 0x10 2736
10 routineName std::basic_string<wcha... 002cf840 0x1c "WaitEvent"
11 se SomeEnum 002cf99c 0x4 0x12 (SecondBit | FifthBit) SomeEnum

PS Dbg:\testApp\Threads\0\Stack\9 TestNativeConsoleApp!wmain+0x3ed
> dt rm

Key Value
----
"callFFE0" std::function<int __cdecl(std::vector<std::basic_string<...
"callFoo" std::function<int __cdecl(std::vector<std::basic_string<...
"nothing" std::function<int __cdecl(std::vector<std::basic_string<...
"setEvent" std::function<int __cdecl(std::vector<std::basic_string<...
"sleep" std::function<int __cdecl(std::vector<std::basic_string<...
"twoThreadGuTest" std::function<int __cdecl(std::vector<std::basic_string<...
"waitEvent" std::function<int __cdecl(std::vector<std::basic_string<...

PS Dbg:\testApp\Threads\0\Stack\9 TestNativeConsoleApp!wmain+0x3ed
> (dt se).DbgGetSymbol().Type.Enumerands

Index Name Value
-----
0 None 0x00
1 FirstBit 0x01
2 SecondBit 0x02
3 ThirdBit 0x04
4 FourthBit 0x08
5 FifthBit 0x10

PS Dbg:\testApp\Threads\0\Stack\9 TestNativeConsoleApp!wmain+0x3ed
>
```

Notable Features

- Color: support for text colorization using ANSI escape codes (a la ISO/IEC 6429)
- Custom formatting engine: Don't like .ps1xml stuff? Me neither. In addition to standard table, list, and custom views, you can define "single-line" views which are very handy for customizing symbol value displays.
- Custom symbol value conversion: For most variables, the default conversion and display are good. But sometimes, you'd like the debugger to do a little more work for you. The symbol value conversion feature allows, for instance, STL collection objects to be transformed into .NET collection objects that are much easier to deal with.
- Derived type detection: For when your variable is an IFoo, but the actual object is a FooImpl.
- Rich type information: exposed for your programmatic pleasure.
- **Q:** Does it work in WinDbg? I will only use WinDbg. **A:** Yes—load up the DbgShellExt.dll extension DLL, and then run "`!dbgshell`" to pop open a DbgShell console.

Current Deficiencies

- The biggest deficiency currently is that it does not support kernel mode well (if you are already in the proper context, you can display values, but you can't change context from within DbgShell, and the namespace is not wired up).
- Although you can load and execute traditional debugger extensions in the usual way, there are still many windbg commands missing.
- Remotes are not supported: the dbgeng API supports connecting to a remote debugger. Unfortunately, the symbol and type information exposed by the dbgeng API is critically insufficient for DbgShell's needs, so DbgShell uses the dbghelp API. Unfortunately, there is no such thing as remote dbghelp. We will need to work with the debugger team to solve this problem.

License

Licensed under the MIT License.

Contributing

This project welcomes contributions and suggestions. Most contributions require you to agree to a Contributor License Agreement (CLA) declaring that you have the right to, and actually do, grant us the rights to use your contribution. For details, visit <https://cla.microsoft.com>.

When you submit a pull request, a CLA-bot will automatically determine whether you need to provide a CLA and decorate the PR appropriately (e.g., label, comment). Simply follow the instructions provided by the bot. You will only need to do this once across all repos using our CLA.

See Contributing for more information about contributing to the project.

Code of Conduct

This project has adopted the Microsoft Open Source Code of Conduct.

For more information see the Code of Conduct FAQ or contact opencode@microsoft.com with any additional questions or comments.

Other topics

- Getting Started with DbgShell
- Color
- Custom formatting engine
- Custom symbol value conversion
- Derived type detection
- Rich type information
- Hacking on DbgShell
- DbgEngWrapper

You can find a short (3 minute) video introduction here: <https://youtu.be/ynbg2zZ1lgc>