WinDbg Intro

Contents

Intro 1

Moving Parts 2

Process 2

Module 2

Debug Symbols 2

Crash Dumps 2

Debugging the Dumps 3

Installation 4

WinDbg GUI 4

Binaries 4

Debug Symbols 4

Crash Dump Analysis 4

Generic Analysis 4

Advanced WinDbg Commands 5

Final Words 6

Bonus Chapter: Analyzing Kernel Dumps 6

Intro

Windows OS has awesome support for debugging stuff. The infrastructure for that is included in the kernel, and in many user-mode components as well.

WinDbg discussed in this article is merely a GUI wrapper around the corresponding OS subsystem.

Unfortunately, good UX was not a priority for MS. The tool is relatively hard to use, that's why this document is required.

The crash dumps workflow discussed in this article is a last resort, not a replacement for QA processes. Like it or not, your app gonna crash occasionally on customer's PCs, once your install base becomes large enough. Even if your code is 100% perfect, other components are not. Windows DLLs, drivers, and even hardware have bugs in them. Given vast diversity of PC hardware, you better be prepared for the worst.

Moving Parts

Process

From Wikipedia: a **process** is the instance of a computer program that is being executed by one or many threads. It contains the program code and its activity. A process is made up of multiple threads of execution that execute instructions concurrently.

A process consists of the following resources:

An image of the executable machine code associated with a program. In Windows, that code comes from modules.

Memory, specifically multiple regions of virtual memory; includes executable code, data from the modules, a call stack of every thread of the process, and a heap.

Operating system descriptors of resources that are allocated to the process, such as file and socket handles.

Thread state (context), such as the content of registers and physical memory addressing. The state is stored in CPU registers while a thread is executing, in memory otherwise.

Module

Most Windows processes have exactly one entry point module, the .exe file, and multiple DLL modules. Some DLLs are parts of the OS, others are installed along with the .exe, some generated in runtime and aren't backed by a file on disk. Dll files don't need to have the .dll extension, that's just a convention Windows kernel doesn't care too much about. In practice, *.ocx, *.mui, *.sys Windows files are actually DLLs, some third-party software uses custom extensions for their DLLs.

Debug Symbols

When compilers are producing Windows binaries, they put there just enough stuff to run the code.[1] The metadata to correlate machine code with source code is called "debug symbols", compilers put that into .pdb files they produce. Debug symbols contain names of functions, variables and structures, mappings between binary locations (instruction and data addresses) and source code locations, definitions of structures, and more. Typically, they're considerably larger than binaries, and aren't included in installers of software.

For better debugging experience, Microsoft is distributing public symbols of most Windows DLLs. WinDbg can download them as needed from Microsoft symbol servers.

For custom built binaries, it's developer's responsibility to produce and keep debug symbols. Debugging without symbols is doable but extremely hard to do. Builds made by VC++ are not reproducible, they have quite a few timestamps in various headers. Even if you use exactly same compiler and Windows SDK version, binaries and symbols will be slightly different each time you build your code.

Crash Dumps

Ideally, when debugging stuff, you want to have everything: live process, binaries, debug symbols, and source code. When you press F5 in visual studio, that's what normally happens. You have the source

[1] Things are different on Linux. There, binaries include symbols by default, and a separate tool exists to strip them.

code you've just compiled. You have the binaries produced by compiler. The binaries include absolute paths to the debug symbols (VC++ projects produce them by default). Your application is running, so you have live process the debugger can interact with in many ways.

When your software runs after it's shipped, or worse, when it crashes on a customer's PC, that's not the case, but you still want to debug it.

One way to do it is by analyzing crash dumps.

From Wikipedia: a crash dump consists of the recorded state of the working memory of a computer program at a specific time, generally when the program has crashed or otherwise terminated abnormally.

See this article how to setup Windows to save crash dumps of specific processes. In the software I ship, I usually set up these registry entries in the installer. As you see from the article linked above, crash dumps come in 2 types: full dumps, and mini dumps.

Protip: if your software gonna include code to upload the collected dumps, be sure to compress them, it helps a lot. For example, a full dump of Windows 10 calculator takes 224MB uncompressed, 73.7MB compressed into ZIP by Windows explorer, and just 46.9MB compressed with 7zip using "Ultra" compression level.

Full Dumps

They include the complete address space of the process. Most useful for debugging, but given the amount of RAM found in modern PCs, they take a lot of space for complex apps, and a lot of bandwidth to upload. Full dumps contain all loaded binaries, the complete heap, stacks of all threads, and more.

Mini Dumps

They can be customized to select what's included. By default, they contain following data.

- Basic system information: Windows version, uptime, limited CPU ID.
- Paths, versions and checksums of all modules loaded in the process.
- List of all threads of the process, their complete native stacks, and thread states.

By default, they won't contain the heaps, nor the binaries. Because of that, their size is much more reasonable, a dozen of MB when compressed, it's perfectly OK to upload them somewhere (after asking user's permission, of course).

The rest of this document assumes you have a minidump, unless noted otherwise.

One more thing, you don't have to crash an app to produce a dump. Find the process in Windows task manager on Details tab, right click, and select "Create dump file" from the context menu. Windows will pause the process, write the full dump to %TEMP%, and resume the process once finished. This can be useful if the application hangs as opposed to crashing, or consumes 100% CPU time for no apparent reason, and you want to fix that thing.

Debugging the Dumps

Assuming you've got a dump of your process, you then want to make sense of it — to find out what was wrong, and ideally fix the software.

Installation

WinDbg GUI

Microsoft has rather long article on the subject. Hopefully, you already have visual studio and Windows SDK installed on your development PC. If that's the case, the simple way to get WinDbg is by using add/remove programs screen. Click settings, programs and features, type "Kit" in the search box. Right click the latest version of "Windows Software Development Kit", and select "Change" from the context menu.

On the next screen, select "Change" and press Next.

On the next screen, check the "Debugging Tools for Windows" box and press Change.

Once finished, you'll get 2 new shortcuts in the start menu, "WinDbg (X64)" and "WinDbg (X86)". Run one of them.

Binaries

If you have a full dump, you may skip this section.

Minidumps don't contain binaries, they only contain their paths, versions and checksums.

One way to proceed, install same version of your software to the same location where it was on the customer's PC. If you'll do that, WinDbg will find the binaries by absolute paths.

Another option, press File / "Image File Path…" or Control+I, and enter the path to a directory in your development PC where you have binaries of the matching version, matching the binaries which were running at the time the dump was created.

The rest of this document assumes that path is `c:\Dev\Binaries`

Debug Symbols

Press File / "Symbol File Path…" or Control+S, copy-paste following:

```
C:\Dev\Symbols;srv*C:\Temp\SymbolCache*https://msdl.microsoft.com/download/symbols
```

This assumes you have debug symbols of your binaries in C:\Dev\Symbols, and you want to cache the downloaded ones into C:\Temp\SymbolCache.

Remember, MS symbol server only has public debug symbols for Windows DLLs, not for your software. It's your responsibility to supply debug symbols of your modules, matching the binaries which were running at the time of crash.

Crash Dump Analysis

You're now ready to analyze dump files.

Generic Analysis

Press File / "Open Crash Dump…" or Control+D, open the *.dmp file that you have.

And now comes the trickiest part, using the WinDbg command-line shell. It looks like a GUI app but the main component of it is an interactive shell.

If you don't know the exact version of your software that was running while capturing the dump, run the following command:

```
> lm vm YourApp
```

where YourApp is the name of some module of your software. You should see following information, along with more info you can ignore for now:

```
start             end                 module name
000001e2`25e70000 000001e2`25ff8000   YourApp    (deferred)
    Mapped memory image file: c:\Dev\Binaries\YourApp.exe
    Image path: C:\Program Files\YourApp\YourApp.exe
    Image name: YourApp.exe
    File version:     2.1.22.0
    Product version:  2.1.22.0
```

Mapped memory image file is what you have on your development PC, can be missing if you don't have.

Image path is what your customer had at the time of crash. The critically important part is version information. This should allow you to find the correct binary and matching debug symbols.

If your software includes parts written in .NET, you'll want a debugger extension DLL. Here's how to install the correct version of it:

```
> .cordll -ve -u -l
```

If you have set up the symbol server correctly, and if you're using reasonably new version of .NET framework in your software (4.5 or newer), this step will download and run the correct version of that extension.

To **analyze the dump**, run following command:

```
> !analyze -v
```

This may take considerable time, minutes, especially the first time you're doing it, because WinDbg will be downloading debug symbols for all Windows DLLs which were loaded at the time of crash. And they gonna take sizeable bandwidth and disk space — if you do that often, you'll soon note your symbols cache folder takes many gigabytes.

If you're lucky, the output will tell you exactly what's wrong, including source paths and line numbers.

Otherwise, you need more of these commands.

Advanced WinDbg Commands

Identifying the CPU

`!cpuid` command prints details on the CPU which was running the code. The manufacturer is not available. But at least you usually have frequency, hardware threads count, and the F/M/S numbers mean Family/Model/Stepping. WinDbg prints them in decimal, so if it printed 16,4,3, this translates to 10/4/3 in hex, which means your software was running on a prehistoric AMD K10 CPU like Phenom II X4. Search the web for details.

Random Commands

Clear screen	`.cls`	
List all threads	`~`	
List threads including their stacks	`~* K`	
Print native stack trace	`K`	
Print .NET stack trace	`!clrstack`	
View all .NET exception objects	`!dumpheap -type Exception`	Requires a full dump, unfortunately — .NET objects are on the CLR heap.
List modules	`lm`	
List modules with full details	`lm v`	
List modules with their paths on customer's PC	`lm o f sm`	
Locate all stacks that contain a symbol or module	`!findstack Symbol`	
Disassemble a function	`uf dxgi!CDXGISwapChain::Present`	Note the format. In this example, dxgi is module name, because the target function is from dxgi.dll. The function name to disassemble is to the right of the exclamation mark.
Display a symbol at or near the address	`ln 000001e225f08000`	

See this article for more complete list of these commands. Or this page for complete reference, but it has pages and pages of them.

Final Words

WinDbg is a complex beast, this article only scratched the surface of what it can do. It can also debug live processes not just crash dumps. It has kernel mode debugger not just user mode. It supports kernel debugging of another PC connected with a special cable. It includes infrastructure for remote debugging over the network.

It has tons of extensions, both first-party made by MS, and third-party ones. It's very old and stable technology, the first version mentioned on the Internets is 4.0.18 from 2001.

I'm not an expert in the technology, but for many of my use cases it still helps a lot.

Bonus Chapter: Analyzing Kernel Dumps

If you ever have blue screen of death caused by critically failing driver or hardware, by default, windows saves kernel minidumps in C:\Windows\minidump folder. Repeat the above steps to make sense of them: setup symbols (unless you working for MS you don't have private symbols for Windows kernel, just the public ones), load the dump with Control+D, and run that `!analyze -v` command. If you'll see a third-party module on the call stack (nvlddmkm: nVidia GPU, igdkmd64: Intel GPU, etc.), you can be almost sure you've found the buggy kernel mode driver. Or you might find some other hints on what might be wrong with the PC.