PDB Files: The glue between binary file and source code

Vineel Kumar Reddy Kovvuri
http://vineelkumarreddy.com

Contents

1 Introduction

Have you ever wondered how a debugger magically gets you to the correct pdb and correct sources when debugging an application? This article talks exactly that in the context of Windbg.

As you might be aware of, PDB files(also called as symbol files) is the glue between your application binary and the source code. There are two key Environment variables which configures Windbg about where to look for symbols and sources. They are _NT_SYMBOL_PATH and _NT_SOURCE_PATH. The _NT_SYMBOL_PATH points to the directory containing your PDBs(also called as symbol files) or to a symbol server. _NT_SOURCE_PATH points to the directory of your sources or to a source server which indexes the soruce files. One important point to remember here is one or more source files make up one or more binary files. But each binary will have a single PDB unless the source code is modified. This is important because Windbg has to perform lot of book keeping to map binary symbols with their source locations.

In this article we would like to understand how Windbg brings the right symbols and sources from both Symbol Server and Source Server even when the binary changes across the debugging sessions. Below are the three topics that we are going to understand.

1. How do Windbg identify the correct symbol file?
2. How do Windbg identify the correct source file?
3. Windbg Symbols and Sources search heuristics

2 How do Windbg identify the correct symbol file?

Whenever an application is compiled the compiler will generate a pdb file associated with it. These pdbs comes in two variants one containing public symbols and other contianing private symbols. Public symbols does not contain all the information related to the binary and sources. Where as private symbols have every possible information (like line number/local variables/parameters info) related to the binary and sources. Every company who tries protect their intellectual property does not publish their private symbols because they contains way too much information which facilites reverse engineering the binaries much easier. Ofcourse, I should mention nothing really stop a skillful reverse engineer. That said, public symbols only gives the customer basic information about the components shipped by these companies. Since we are dealing with our own application we can assume we have access to private symbols which are much more helpful.

When application is being built the compiler embeds a GUID and the absolute path to the pdb in to binary. Also, it embeds the same GUID in to the generated PDB. This GUID acts as a hash for windbg to check whether the pdb located by the embedded path or via _NT_SYMBOL_PATH matches or not. In cases where we are using symbol server it queries the symbol server for the appropriate pdb based on the guid.

We can get this information either through **dumpbin /headers module.exe** or using **!lmi module** as shown below respectively

```
...
 Debug Directories

        Time Type        Size      RVA  Pointer
    -------- ------- -------- -------- --------
    5A6C0899 cv            53 0001A8B4     94B4    Format: RSDS,
↪   {BB6248C9-7748-4F74-9CBA-147BF261F206}, 1, C:\Programs\Sample.pdb
...
```

```
0:000> !lmi Sample
Loaded Module Info: [sample]
         Module: Sample
   Base Address: 00007ff6ff400000
```

```
    Image Name: Sample.exe
  Machine Type: 34404 (X64)
    Time Stamp: 5a6c0899 Fri Jan 26 21:05:29 2018
          Size: 24000
      CheckSum: 0
Characteristics: 22
Debug Data Dirs: Type  Size     VA  Pointer
             CODEVIEW    53, 1a8b4,    94b4 RSDS - GUID:
↪  {BB6248C9-7748-4F74-9CBA-147BF261F206}
               Age: 1, Pdb: C:\Programs\Sample.pdb
           VC\_FEATURE    14, 1a908,    9508 [Data not mapped]
   Symbol Type: DEFERRED - No error - symbol load deferred
   Load Report: no symbols loaded
```

If the GUID in pdb does not match with embedded GUID in binary it does not load the PDB file and throws following error ***** ERROR: Module load completed but symbols could not be loaded for Win32Sample.exe**.

```
0:000> .sympath E:\temp\Testing\x64\Release\ <-- Incorrect pdb
Symbol search path is: E:\temp\Testing\x64\Release\
Expanded Symbol search path is: e:\temp\testing\x64\release\

************* Path validation summary **************
Response                         Time (ms)     Location
OK                                             E:\temp\Testing\x64\Release\
*** WARNING: Unable to verify checksum for Win32Sample.exe
*** ERROR: Module load completed but symbols could not be loaded for
↪  Win32Sample.exe
```

3 How do Windbg identify the correct source file?

PDB files contain not only information about symbols like functions/structures/classes etc but also about the artifacts(like .obj) involved in generating your application binary. Unfortunately examining this information from PDB is little complicated because the PDB format is not documented by Microsoft. But the good news is Microsoft has provided an API to query the information about any given PDB. This API is called Debug Interface Access SDK. Luckly, every installation of Visual Studio ships with a sample project aptly named as **Dia2Dump** at `C:\Program Files (x86)\Microsoft Visual Studio\2017\Enterprise\DIA SDK\Samples\DIA2Dump`. When you build this project in Visual Studio we get Dia2Dump.exe, Using this we can solve the second puzzle.

```
usage: Dia2Dump.exe [ options ] <filename>
  -?                : print this help
  -all              : print all the debug info
  -m                : print all the mods
  -p                : print all the publics
  -g                : print all the globals
  -t                : print all the types
  -f                : print all the files
  -s                : print symbols
  -l [RVA [bytes]]  : print line number info at RVA address in the bytes range
  -c                : print section contribution info
  -dbg              : dump debug streams
  -injsrc [file]    : dump injected source
  -sf               : dump all source files
  -oem              : dump all OEM specific types
```

```
  -fpo [RVA]        : dump frame pointer omission information for a func addr
  -fpo [symbolname] : dump frame pointer omission information for a func symbol
  -compiland [name] : dump symbols for this compiland
  -lines <funcname> : dump line numbers for this function
  -lines <RVA>      : dump line numbers for this address
  -type <symbolname>: dump this type in detail
  -label <RVA>      : dump label at RVA
  -sym <symbolname> [childname] : dump child information of this symbol
  -sym <RVA> [childname]        : dump child information of symbol at this addr
  -lsrc  <file> [line]          : dump line numbers for this source file
  -ps <RVA> [-n <number>]       : dump symbols after this address, default 16
  -psr <RVA> [-n <number>]      : dump symbols before this address, default 16
  -annotations <RVA>: dump annotation symbol for this RVA
  -maptosrc <RVA>   : dump src RVA for this image RVA
  -mapfromsrc <RVA> : dump image RVA for src RVA
```

The most important of all these flags is **-sf** which will dump out all the source files used to create an obj(object file). A sample output form this command **Dia2Dump.exe -sf ¡path of pdb file¿** with a PDB will be as shown below

```
....
Compiland = C:\..<snipped>..\Win32Sample\x64\Release\Win32Sample.obj

        c:\..<snipped>..\10.0.15063.0\shared\basetsd.h (MD5:
↪  464E631AE358F42C09701CE07F35F8BF)
        c:\..<snipped>..\10.0.15063.0\shared\guiddef.h (MD5:
↪  CA7D066706A198EA5999B084AAB0CE58)
        c:\..<snipped>..\10.0.15063.0\shared\stralign.h (MD5:
↪  D27BD3C9FFF58FF4798B1F17B38C5B06)
        c:\..<snipped>..\10.0.15063.0\shared\winerror.h (MD5:
↪  7AD19053F0A83DDC031CDE4638299080)
        c:\..<snipped>..\10.0.15063.0\ucrt\corecrt_memory.h (MD5:
↪  33686D742EF373658431918E1A52326C)
        c:\..<snipped>..\10.0.15063.0\ucrt\ctype.h (MD5:
↪  1AC17C8CFC2358BD87784AB186BBAFCC)
        c:\..<snipped>..\10.0.15063.0\ucrt\stdlib.h (MD5:
↪  49CF59C87D23BB42C2D25CDF0D089509)
        c:\..<snipped>..\10.0.15063.0\ucrt\string.h (MD5:
↪  1DD6630B6C5E4B83DE098670242950A2)
        c:\..<snipped>..\10.0.15063.0\um\memoryapi.h (MD5:
↪  6F6D38BE202064596573E9449CBAAC58)
        c:\..<snipped>..\10.0.15063.0\um\oleauto.h (MD5:
↪  9048E2C1FD07AD42EA6E7F51EF63D42B)
        c:\..<snipped>..\10.0.15063.0\um\processthreadsapi.h (MD5:
↪  65891E84D54E51FA2017AB4D17BF9958)
        c:\..<snipped>..\10.0.15063.0\um\propidl.h (MD5:
↪  B19A6DCE51821A635FE051DBC1CE6E7E)
        c:\..<snipped>..\10.0.15063.0\um\winbase.h (MD5:
↪  86C4964B16E8566D1E8F05482D9FFA49)
        c:\..<snipped>..\10.0.15063.0\um\winnt.h (MD5:
↪  02092055CFA70103E984B6855200845C)
        c:\..<snipped>..\10.0.15063.0\um\winuser.h (MD5:
↪  8A3479DAEAB702729FFBA6C669F54438)
        c:\..<snipped>..\win32sample\stdafx.h (MD5:
↪  AA9C091299F07AD95BB49E6EE4BFF136)
        c:\..<snipped>..\win32sample\win32sample.cpp (MD5:
↪  BBB7EE64784A7C2A96B1439310EEF84A)  <---
        c:\..<snipped>..\win32sample\x64\release\win32sample.pch
```

```
....
```

The above information clearly suggests that PDB also contains hash of all the files needed to create a obj. This hash could be MD5 or SHA256(which is often denoted with 0x3), This can be verified using simple get-filehash commandlet on our source file(Win32Sample.cpp) as shown below. Similar to GUID which binds a binary with PDB file this file checksum will bind the binary with its appropriate source file. But the checking of source files against its checksum is somewhat relax(more on this later).

```
PS> get-filehash -Algorithm MD5 "c:\<snipped>\win32sample\win32sample.cpp"

Algorithm Hash                             Path
--------- ----                             ----
MD5       BBB7EE64784A7C2A96B1439310EEF84A
↪  C:\<snipped>\win32sample\win32sample.cpp
```

This confirms the PDB is indeed storing the MD5 Hash of the file content.

Because each PDB contains symbol and line number information it can open the appropriate source file and at the correct line number.

Whenever windbg tries to open a source file associated with a symbol it tries to check for this checksum of the source file.

4 Windbg Symbols and Sources search heuristics

If the absolute paths embeded in the PDB file is all we have then debugging in Windbg would not be any interesting and fun. Practically we cannot have the pdbs and sources available at the embedded paths(for example a program being debugged at the client machine). So how does Windbg figures out the right PDB even when the symbol paths set via .sympath+ or _NT_SYMBOL_PATH and the source path set via .srcpath+ or _NT_SOURCE_PATH are different from the actual embeded paths in the PDB?

To understand this we need to enable **!sym noisy** and **.srcnoisy 3** when debugging. As an example I have my original source code built from `C:\users\vineelko\documents\visualstudio2017\projects\win32sample` which generated below files

1. Sources: `C:\<snipped>\Win32Sample\Win32Sample.c`
2. Binary: `C:\<snipped>\Win32Sample\x64\Release\Win32Sample.exe`
3. PDB: `C:\<snipped>\Win32Sample\x64\Release\Win32Sample.pdb`

Lets say we moved the folder from `C:\Users\vineelko\Documents\VisualStudio2017\Projects\` to `E:\Temp\` like below

1. Sources: `E:\Temp\Win32Sample\Win32Sample.c`
2. Binary: `E:\Temp\Win32Sample\x64\Release\Win32Sample.exe`
3. PDB: `E:\Temp\Win32Sample\x64\Release\Win32Sample.pdb`

Now when we start debugging session **windbg.exe** `E:\Temp\Win32Sample\x64\Release\Win32Sample.exe` Windbg tries to find the .c and .pdb from `C:\Users\vineelko\Documents\VisualStudio2017\Projects\Win32Sample` but will not find them as they are moved. This is where we have to make use of **.sympath+** `E:\Temp\Win32Sample\x64\Release` and **.srcpath+** `E:\Temp\Win32Sample`. Also running **!sym noisy** and **.srcnoisy 3** commands will enable tracing of debugger when it is searching for the symbol files and source files respectively.

Below is the output of !lmi which dumps the GUID and the PDB location.

```
0:000> !lmi Win32Sample
Loaded Module Info: [win32sample]
         Module: Win32Sample
   Base Address: 00007ff62d2f0000
     Image Name: Win32Sample.exe
   Machine Type: 34404 (X64)
     Time Stamp: 5a6c354c Sat Jan 27 00:16:12 2018
           Size: 7000
       CheckSum: 0
Characteristics: 22
Debug Data Dirs: Type  Size     VA  Pointer
             CODEVIEW    78,  2368,    1568 RSDS - GUID:
↪  {7FAA9AFE-D714-4E38-B2CE-A99C41BF8CD4}
               Age: 1, Pdb: C:\Users\vineelko\Documents\Visual Studio
↪  2017\Projects\Win32Sample\x64\Release\Win32Sample.pdb
           VC_FEATURE    14,  23e0,    15e0 [Data not mapped]
                 POGO   26c,  23f4,    15f4 [Data not mapped]
    Symbol Type: DEFERRED - No error - symbol load deferred
    Load Report: no symbols loaded
```

Turn on verbose symbol logging

```
0:000> !sym noisy
noisy mode - symbol prompts on
```

Try setting the symbol path to `E:\temp\Testing`

```
0:000> .sympath E:\temp\Testing\
Symbol search path is: E:\temp\Testing\
Expanded Symbol search path is: e:\temp\testing\
************* Path validation summary **************
Response                         Time (ms)     Location
OK                                             E:\temp\Testing\
```

Try reloading the binary PDB

```
0:000> .reload /f win32sample.exe
DBGHELP: e:\temp\testing\Win32Sample.pdb - file not found
DBGHELP: e:\temp\testing\exe\Win32Sample.pdb - file not found
DBGHELP: e:\temp\testing\symbols\exe\Win32Sample.pdb - file not found
DBGHELP: C:\Users\vineelko\Documents\Visual Studio
↪  2017\Projects\Win32Sample\x64\Release\Win32Sample.pdb - file not found
*** WARNING: Unable to verify checksum for Win32Sample.exe
*** ERROR: Module load completed but symbols could not be loaded for
↪  Win32Sample.exe
DBGHELP: Win32Sample - no symbols loaded

0:000> .sympath E:\temp\Testing\x64\Release
0:000> .reload /f win32sample.exe
*** WARNING: Unable to verify checksum for Win32Sample.exe
DBGHELP: Win32Sample - private symbols & lines
        e:\temp\testing\x64\release\Win32Sample.pdb
0:000> lm
start             end                 module name
00007ff6`2d2f0000 00007ff6`2d2f7000   Win32Sample C (private pdb symbols)
↪  e:\temp\testing\x64\release\Win32Sample.pdb
```

```
00007ff9`0c5e0000 00007ff9`0c5f6000   VCRUNTIME140   (deferred)
00007ff9`2c110000 00007ff9`2c206000   ucrtbase   (deferred)
00007ff9`2c430000 00007ff9`2c696000   KERNELBASE   (deferred)
00007ff9`2f160000 00007ff9`2f20e000   KERNEL32   (deferred)
00007ff9`2f400000 00007ff9`2f5e0000   ntdll      (export symbols)
↪  C:\WINDOWS\SYSTEM32\ntdll.dll
```

Looking at the .reload command output we can say Windbg expects symbol files to be present inside the same directory as the application or the folder named 'exe' inside .sympath directory or the folder named 'dll' inside the .sympath directory or at the actual embedded path. Running lm confirms that symbols are recognized for Win32Sample.exe

Turn on verbose source logging

```
0:000> !srcnoisy 3
Noisy source output: on
Noisy source server output: on
Filter out everything but source server output: off

0:000> .srcpath e:\temp\Testing
Source search path is: e:\temp\Testing
************* Path validation summary **************
Response                         Time (ms)     Location
OK                                             e:\temp\Testing

0:000> x win32Sample!main
00007ff6`2d2f1060 Win32Sample!main (void)
0:000> bu win32Sample!main
0:000> g
Breakpoint 0 hit
Win32Sample!main:
00007ff6`2d2f1060 4883ec48        sub     rsp,48h
DBGENG:  Scan paths for partial path match:
DBGENG:    prefix 'c:\users\vineelko\documents\visual studio
↪  2017\projects\win32sample'
DBGENG:    suffix 'win32sample.cpp'
DBGENG:      match 'e:\temp\Testing' against 'c:\users\vineelko\documents\visual
↪  studio 2017\projects\win32sample': 66 (match '')
DBGENG:  Scan paths for partial path match:
DBGENG:    prefix 'c:\users\vineelko\documents\visual studio 2017\projects'
DBGENG:    suffix 'win32sample\win32sample.cpp'
DBGENG:      match 'e:\temp\Testing' against 'c:\users\vineelko\documents\visual
↪  studio 2017\projects': 54 (match '')
DBGENG:  Scan paths for partial path match:
DBGENG:    prefix 'c:\users\vineelko\documents\visual studio 2017'
DBGENG:    suffix 'projects\win32sample\win32sample.cpp'
DBGENG:      match 'e:\temp\Testing' against 'c:\users\vineelko\documents\visual
↪  studio 2017': 45 (match '')
DBGENG:  Scan paths for partial path match:
DBGENG:    prefix 'c:\users\vineelko\documents'
DBGENG:    suffix 'visual studio 2017\projects\win32sample\win32sample.cpp'
DBGENG:      match 'e:\temp\Testing' against 'c:\users\vineelko\documents': 26
↪  (match '')
DBGENG:  Scan paths for partial path match:
DBGENG:    prefix 'c:\users\vineelko'
DBGENG:    suffix 'documents\visual studio
↪  2017\projects\win32sample\win32sample.cpp'
DBGENG:      match 'e:\temp\Testing' against 'c:\users\vineelko': 16 (match '')
DBGENG:  Scan paths for partial path match:
```

```
DBGENG:    prefix 'c:\users'
DBGENG:    suffix 'vineelko\documents\visual studio
↪  2017\projects\win32sample\win32sample.cpp'
DBGENG:      match 'e:\temp\Testing' against 'c:\users': 7 (match '')
DBGENG:  Scan paths for partial path match:
DBGENG:    prefix 'c:'
DBGENG:    suffix 'users\vineelko\documents\visual studio
↪  2017\projects\win32sample\win32sample.cpp'
DBGENG:      match 'e:\temp\Testing' against 'c:': 1 (match '')
DBGENG:  Scan all paths for:
DBGENG:    'c:\users\vineelko\documents\visual studio
↪  2017\projects\win32sample\win32sample.cpp'
DBGENG:      check 'e:\temp\Testing\c:\users\vineelko\documents\visual studio
↪  2017\projects\win32sample\win32sample.cpp'
DBGENG:  Scan all paths for:
DBGENG:    'users\vineelko\documents\visual studio
↪  2017\projects\win32sample\win32sample.cpp'
DBGENG:      check 'e:\temp\Testing\users\vineelko\documents\visual studio
↪  2017\projects\win32sample\win32sample.cpp'
DBGENG:  Scan all paths for:
DBGENG:    'vineelko\documents\visual studio
↪  2017\projects\win32sample\win32sample.cpp'
DBGENG:      check 'e:\temp\Testing\vineelko\documents\visual studio
↪  2017\projects\win32sample\win32sample.cpp'
DBGENG:  Scan all paths for:
DBGENG:    'documents\visual studio 2017\projects\win32sample\win32sample.cpp'
DBGENG:      check 'e:\temp\Testing\documents\visual studio
↪  2017\projects\win32sample\win32sample.cpp'
DBGENG:  Scan all paths for:
DBGENG:    'visual studio 2017\projects\win32sample\win32sample.cpp'
DBGENG:      check 'e:\temp\Testing\visual studio
↪  2017\projects\win32sample\win32sample.cpp'
DBGENG:  Scan all paths for:
DBGENG:    'projects\win32sample\win32sample.cpp'
DBGENG:      check 'e:\temp\Testing\projects\win32sample\win32sample.cpp'
DBGENG:  Scan all paths for:
DBGENG:    'win32sample\win32sample.cpp'
DBGENG:      check 'e:\temp\Testing\win32sample\win32sample.cpp'
DBGENG:  Scan all paths for:
DBGENG:    'win32sample.cpp'
DBGENG:      check 'e:\temp\Testing\win32sample.cpp'
DBGENG:      found file 'e:\temp\Testing\win32sample.cpp'
```

Source path heuristics to match a source file is much more involved, It is done by prefix and suffix matches by joining the file path and .srcpath directory. If the checksum of the file present in PDB does does not match with the file checksum then you should see a warning like below(Which is very important). Unlike symbol files, Eventhough Windbg throws a warning it opens the source file in code window. But we should be vigilient about it.

```
windbg> .open -a Win32Sample!main
WARNING: Unable to find source file with matching checksum.  Found
↪  'c:\<snipped>\win32sample\win32sample.cpp' with mismatch!
```

In any case, Understanding these details will help us solve unresolved source and symbol files issue with much more confidence!

5 References

1. PDB Files: What Every Developer Must Know
2. Debug Interface Access SDK