Definitive Dossier of Devilish Debug Details — Part One: PDB Paths and Malware | Mandiant

By Mandiant

Published: 2019-08-29 · Archived: 2026-04-05 16:19:43 UTC

Have you ever wondered what goes through the mind of a malware author? How they build their tools? How they organize their development projects? What kind of computers and software they use? We took a stab and answering some of those questions by exploring malware debug information.

We find that malware developers give descriptive names to their folders and code projects, often describing the capabilities of the malware in development. These descriptive names thus show up in a *PDB path* when a malware project is compiled with symbol debugging information. Everyone loves an origin story, and debugging information gives us insight into the malware development environment, a small, but important keyhole into where and how a piece of malware was born. We can use our newfound insight to detect malicious activity based in part on PDB paths and other debug details.

Welcome to part one of a multi-part, tweet-inspired series about PDB paths, their relation to malware, and how they may be useful in both defensive and offensive operations.

Human-Computer Conventions

Digital storage systems have revolutionized our world but in order to make use of our stored data and retrieve it in an efficient manner, we must organize it sensibly. Users structure directories carefully and give files and folders unique and descriptive names. Often users name folders and files based on their content. Computers force users to label and annotate their data based on the data type, role, and purpose. This human-computer convention means that most digital content has some descriptive surface area, or descriptive "features" that are present in many files, including malware files.

FireEye approaches detection and hunting from many angles, but on FireEye's Advanced Practices team, we often like to flex on "weak signals." We like to search for features of malware that are not evil in isolation but uncommon or unique enough to be useful. We create conditional rules that when met are "weak signals" telling us that a subset of data, such as a file object or a process, has some odd or novel features. These features are often incidental outcomes of adversary methods, or modus operandi, that each represent deliberate choices made by malware developers or intrusion operators. Not all these features were meant to be in there, and they were certainly not intended for defenders to notice. This is especially true for PDB paths, which can be described as an outcome of the compilation process, a toolmark left in malware that describes the development environment.

PDBs

A program database (PDB) file, often referred to as a "symbol file," is generated upon compilation to store debugging information about an individual build of a program. A PDB may store symbols, addresses, names of functions and resources and other information that may assist with debugging the program to find the exact source of an exception or error.

Malware is software, and malware developers are software developers. Like any software developers, malware authors often have to debug their code and sometimes end up creating PDBs as part of their development process. If they do not spend time debugging their malware, they risk their malware not functioning correctly on victim hosts, or not being able to successfully communicate with their malware remotely.

How PDB Paths are Made (the birds and the PDBs?)

But how are PDBs created and connected to programs? Let's examine the formation of one PDB path through the eyes of a malware developer and blogger, the soon-to-be-infamous "smiller."

Smiller has a lot of programming projects and organizes them in an aptly labeled folder structure on his computer. This project is for a shellcode loader embedded in an HTML Application (HTA) file, and the developer stores it quite logically in the folder:

D:\smiller\projects\super_evil_stuff\shellcode\

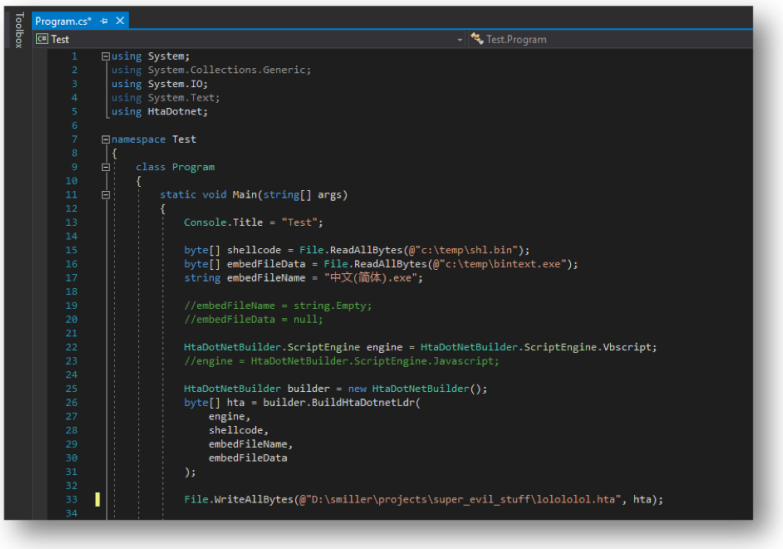

Figure 1: The simple “Test” project code file “Program.cs” which embeds a piece of shellcode and a launcher executable within an HTML Application (HTA) file

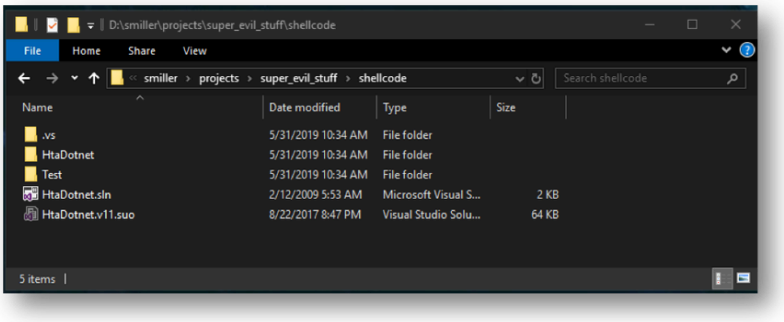

Figure 2: The malicious Visual Studio solution HtaDotnet and corresponding “Test” project folder as seen through Windows Explorer. The names of the folders and files are suggestive of their functionalities

The malware author then compiles their “Test” project Visual Studio in a default “Debug” configuration (Figure 3) and writes out Test.exe and Test.pdb to a subfolder (Figure 4).

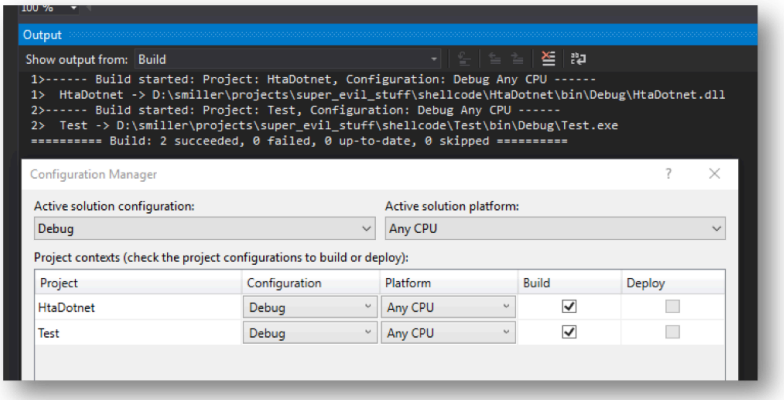

Figure 3: The Visual Studio output of a default compiling configuration

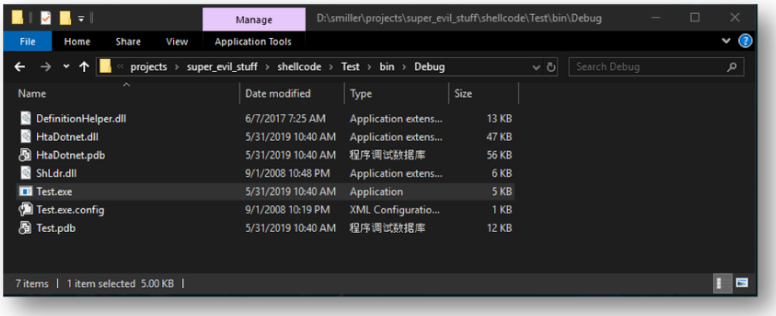

Figure 4: Test.exe and Test.pdb are written to a default subfolder of the code project folder

In the Test.pdb file (Figure 5) there are references to the original path for the source code files along with other binary information for use in debugging.

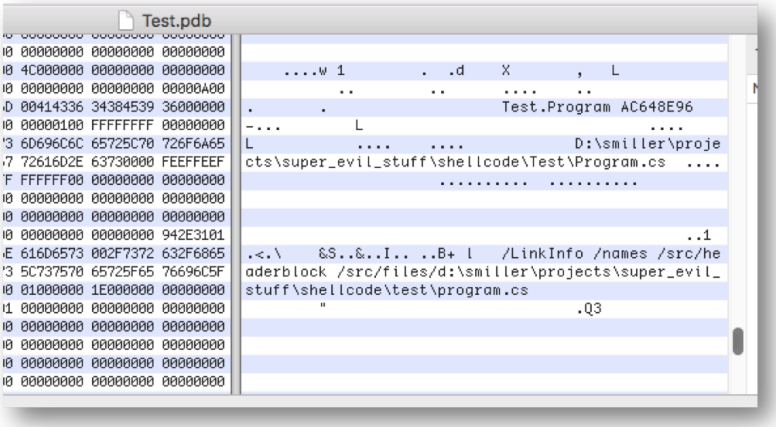

Figure 5: Test.pdb contains binary debug information and references to the original source code files for use in debugging

During the compilation, the *linker* program associates the PDB file with the built executable by adding an entry into the IMAGE_DEBUG_DIRECTORY specifying the type of the debug information. In this case, the debug type is CodeView and so the PDB path is embedded under IMAGE_DEBUG_TYPE_CODEVIEW portion of the file. This enables a debugger to locate the correct PDB file Test.pdb while debugging Test.exe.

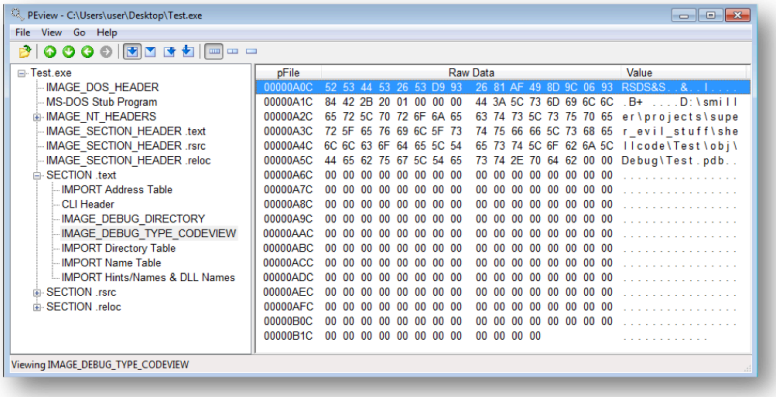

Figure 6: Test.exe as shown in the PEview utility, which easily parses out the PDB path from the IMAGE_DEBUG_TYPE_CODEVIEW section of the executable file

PDB Path in CodeView Debug Information

CodeView Structure

The exact format of the debug information may vary depending on compiler and linker and the modernity of one's software development tools. CodeView debug information is stored under IMAGE_DEBUG_TYPE_CODEVIEW in the following structure:

Type	Description
DWORD	"RSDS" header
GUID	16-byte Globally Unique Identifier
DWORD	"age" (incrementing # of revisions)
BYTE	PDB path, null terminated

Figure 7: Structure of CodeView debug directory information

Full Versus Partial PDB Path

There are generally two buckets of CodeView PDB paths, those that are *fully qualified* directory paths and those that are *partially qualified*, that specify the name of the PDB file only. In both cases, the name of the PDB file with the .pdb extension is included to ensure the debugger locates the correct PDB for the program.

A *partially qualified* PDB path would list only the PDB file name, such as:

Test.pdb

A *fully qualified* PDB path usually begins with a volume drive letter and a directory path to the PDB file name such as:

D:\smiller\projects\super_evil_stuff\shellcode\Test\obj\Debug\Test.pdb

Typically, native Windows executables use a *partially qualified* PDB path because many of the debug PDB files are publicly available on the Microsoft public symbol server, so the fully qualified path is unnecessary in the symbol path (the PDB path). For the purposes of this research, we will be mostly looking at fully qualified PDB paths.

Surveying PDB Paths in Malware

In Operation Shadowhammer, which has a myriad of connections to APT41, one sample had a simple, yet descriptive PDB path: "D:\C++\AsusShellCode\Release\AsusShellCode.pdb"

The naming makes perfect sense. The malware was intended to masquerade as Asus Corporation software, and the role of the malware was shellcode. The malware developer named the project after the function and role of the malware itself.

If we accept that the nature of digital data forces developers into these naming conventions, we figured that these *conventions* would hold true across other threat actors, malware families, and intrusion operations. FireEye's Advanced Practices team loves to take seemingly innocuous features of an intrusion set and determine what about these things is good, bad and ugly. What is normal, and what is abnormal? What is globally prevalent and what is rare? What are malware authors doing that is different from what non-malware developers are doing? What assumptions can we make and measure?

Letting our curiosity take the wheel, we adapted the CodeView debug information structure into a regular expression (Figure 8) and developed Yara rules (Figure 9) to survey our data sets. This helped us identify commonalities and enabled us to see which threat actors and malware families may be "detectable" based only on features within PDB path strings.

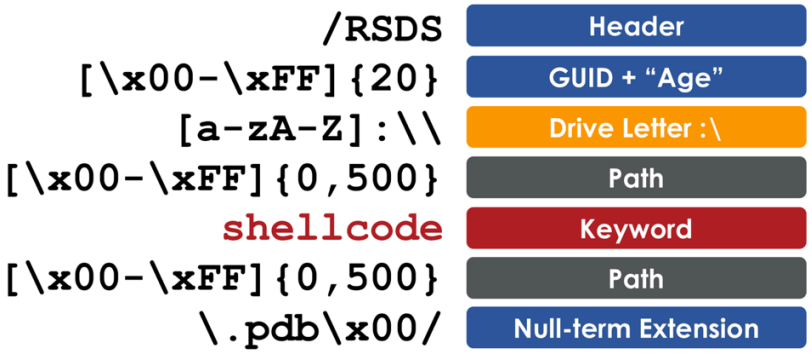

Figure 8: A Perl-compatible regular expression (PCRE) adaptation of the PDB7 debug information in an executable to include a specific keyword

```
rule ConventionEngine_Keyword_Shellcode
{
   meta:
      author = "@stvemillertime"
   strings:
        $anchor = "shellcode" nocase ascii wide
        $pcre = /RSDS[\x00-\xFF]{20}
         [a-zA-Z]:\\[\x00-\xFF]{0,500}
         shellcode[\x00-\xFF]{0,500}
         \.pdb\x00/ nocase ascii wide
   condition:
        (uint16(0) == 0x5A4D) and $anchor and $pcre
}
```

Figure 9: Template Yara rule to search for executables with PDB files matching a keyword

PDB Path Showcase: Malware Naming Conventions

We surveyed 10+ million samples in our incident response and malware corpus, and we found plenty of common PDB path keywords that seemed to transcend different sources, victims, affected regions, impacted industries, and actor motivations. To help articulate the broad reach of malware developer commonalities, we detail a handful of the stronger keywords along with example PDB paths, with represented malware families and threat groups where at least one sample has the applicable keyword.

Please note that the example paths and represented malware families and groups are a selection from the total data set, and not necessarily correlated, clustered or otherwise related to each other. This is intended to illustrate the wide presence of PDB paths with keywords and how malware developers, irrespective of origin, targets and motivations often end up using some of the same words in their naming. We believe that this commonality increases the surface area of malware and introduces new opportunities for detection and hunting.

PDB Path Keyword Prevalence

Keyword	Families and Groups Observed	Example PDB Path
anti	RUNBACK, HANDSTAMP, LOKIBOT, NETWIRE, DARKMOON, PHOTO, RAWHIDE, DUCKFAT, HIGHNOON, DEEPOCEAN, SOGU, CANNONFODDER APT10, APT24, APT41, UNC589, UNC824, UNC969, UNC765	H:\RbDoor\Anti_winmm\AppInit\AppInit\Release\AppInit.pdb
attack	MINIASP, SANNY, DIRTCHEAP, ORCUSRAT APT1, UNC776, UNC251. UNC1131	E:\C\Attack\mini_asp-0615\attack\MiniAsp3\Release\MiniAsp.pdb
backdoor	PACMAN, SOUNDWAVE, PHOTO, WINERACK, DUALGUN APT41, APT34, APT37, UNC52, UNC1131, APT40	Y:\Hack\backdoor\3-exe-attack\temp\UAC_Elevated\win32\UAC_Elevated.pdb

bind	SCREENBIND, SEEGAP, CABLECAR, UPDATESEE, SEEDOOR, TURNEDUP, CABROCK, YABROD, FOXHOLE UNC373, UNC510, UNC875, APT36, APT33, APT5, UNC822	C:\Documents and Settings\ss\桌面\tls\scr\bind\bind\Release\bind.pdb
bypass	POSHC2, FIRESHADOW, FLOWERPOT, RYUK, HAYMAKER, UPCONTROL, PHOTO, BEACON, SOGU APT10, APT34, APT21, UNC1289, UNC1450	C:\Documents and Settings\Administrator\桌面\BypassUAC.VS2010\Release\Go.pdb
downloader	SPICYBEAN, GOOSEDOWN, ANTFARM, BUGJUICE, ENFAL, SOURFACE, KASPER, ELMER, TWOBALL, KIBBLEBITS APT28, UNC1354, UNC1077, UNC27, UNC653, UNC1180, UNC1031	Z:\projects\vs 2012\Inst DWN and DWN XP\downloader_dll_http_mtfs\Release\downloader_dll_http_mtfs.pdb
dropper	CITADEL, FIDDLELOG, SWIFTKICK, KAYSLICE, FORMBOOK, EMOTET, SANNY, FIDDLEWOOD, DARKNEURON, URSNIF, RUNOFF UNC776, UNC1095, APT29, APT36, UNC964, UNC1437, UNC849	D:\Task\DDE Attack\Dropper_Original\Release\Dropper.pdb
exploit	TRICKBOT, RUNBACK, PUNCHOUT, QANAT, OZONERAT UNC1030, APT39, APT34, FIN6	w:\modules\exploits\littletools\agent_wrapper\release\12345678901234567890123456789012345678\wrapper3.pdb
fake	FIRESHADOW UNC1172, APT39, UNC822	D:\Work\Project\VS\house\Apple\Apple_20180115\Release\FakeRun.pdb

fuck	TRICKBOT, CEREAL, KRYPTONITE, SUPERMAN APT17, UNC208, UNC276	E:\CODE\工程文件 \20110505_LEVNOhard\CODE\AnyRat**Fuck**All'sUI\bin\FuckAll.pdb
hack	PHOTO, KILLDEVIL, NETWIRE, PACMAN, BADSIGN, TRESOCHO, BADGUEST, GH0ST, VIPSHELL UNC1152, APT40, UNC78, UNC874, UNC52, UNC502, APT33, APT8	C:\Users\Alienware.DESKTOP-MKL3QDN\Documents\Hacker\memorygrabber - ID\memorygrabber\obj\x86\Debug\vshost.pdb
hide	FRESHAIR, DIRTYWORD, GH0ST, DARKMOON, FIELDGOAL, RAWHIDE, DLLDOOR, TRICKBOT, 008S, JAMBOX, SOGU, CANDYSHELL APT26, APT40, UNC213, APT26, UNC44, UNC53, UNC282	c:\winddk\6001.18002\work**hide**port\i386\HidePort.pdb
hook	GEARSHIFT, METASTAGE, FASTPOS, HANDSTAMP, FON, CLASSFON, WATERFAIRY, RATVERMIN UNC842, UNC1197, UNC1040, UNC969	D:\รายงาน\C++ & D3D & Hook & VB.NET & PROJECT\Visual Studio 2010\CodeMaster OnlyTh\Inject_Win32_2\Inject Win32\Inject Win32\Release\OLT_PBFREE.pdb
inject	SKNET, KOADIC, ISMAGENT, FULLTRUNK, ZZINJECT, ENFAL, RANSACK, GEARSHIFT, LOCKLOAD, WHIPSNAP, BEACON, CABROCK, HIGHNOON, DETECT, THREESNEAK, FOXHOLE UNC606, APT10, APT34, APT41, UNC373, APT31, APT34, APT19, APT1, UNC82, UNC1168, UNC1149, UNC575	E:\0xFFDebug\My Source\HashDump\Release\injectLsa.pdb

install	FIRESHADOW, SCRAPMINT, BRIGHTCOMB, WINERACK, SLUDGENUDGE, ANCHOR, EXCHAIN, KIBBLEBITS, ENFAL, DANCEPARTY, SLIMEGRIME, DRABCUBE, EXCHAIN, DIMWIT, THREESNEAK, GOOGONE, STEW, LOWLIGHT, QUASIFOUR, CANNONFODDER, EASYCHAIR, ONETOFOUR, DEEPOCEAN, BRIGHTCREST, LUMBERJACK, EVILTOSS, BRIGHTCYAN, PEKINGDUCK, SIDEVIEW, BOSSNAIL UNC869, UNC385, UNC228, APT5, UNC229, APT26, APT37, UNC432, APT18, UNC27, APT6, UNC1172, UNC593, UNC451, UNC875, UNC53	i:\LIE_SHOU\URL_CURUN-A\installer\Release\jet.pdb
keylog	LIMITLESS, ZZDROP, WAVEKEY, FIDDLEKEYS, SKIDHOOK, HAWKEYE, BEACON, DIZZYLOG, SOUNDWAVE APT37, UNC82, UNC1095, APT1, APT40	D:\TASK\ProgamsByMe(2015.1~)\MyWork\Relative Backdoor\KeyLogger_ScreenCap_Manager\Release\SoundRec.pdb
payload	POSHC2, SHAKTI, LIMITLESS, RANSACK, CATRUNNER, BREAKDANCE, DARKMOON, METERPRETER, DHARMA, GAMEFISH, RAWHIDE, LIGHTPOKE UNC915, UNC632, UNC1149, APT28, UNC878	C:\Users\WIN-2-ViHKwdGJ574H\Desktop\NSA\Payloads\windows service cpp\Release\CppWindowsService.pdb
shell	SOGU, RANSACK, CARBANAK, BLACKCOFFEE,	E:\windows\dropperNew\Debug\testShellcode.pdb

	SIDEWINDER, PHOTO, SHIMSHINE, PILLOWMINT, POSHC2, PI, METASTAGE, GH0ST, VIPSHELL, GAUSS, DRABCUBE, FINDLOCK, NEDDYSHELL, MONOPOD, FIREPIPE, URSNIF, KAYSLICE, DEEPOCEAN, EIGHTONE, DAYJOB, EXCALIBUR, NICECATCH UNC48, UNC1225, APT17, UNC1149, APT35, UNC251, UNC521, UNC8, UNC849, UNC1428, UNC1374, UNC53, UNC1215, UNC964, UNC1217, APT3, UNC671, UNC757, UNC753, APT10, APT34, UNC229, APT18, APT9, UNC124, UNC1559	
sleep	URSNIF, CARBANAK, PILLOWMINT, SHIMSHINE, ICEDID FIN7	O:\misc_src\release_priv_aut_v2.2_sleep_DATE\my\src\sdb_test_dll\x64\Release\sdb_test.pdb
spy	DUSTYSKY, OFFTRACK, SCRAPMINT, FINSPY, LOCKLOAD, WINDOLLAR FIN7, UNC583, UNC822, UNC1120	G:\development\Winspy\ntsvc32-93-01-05\x64\Release\ntsvcst32.pdb
trojan	ENFAL, IMMINENTMONITOR, MSRIP, GH0ST, LITRECOLA, DIMWIT UNC1373, UNC366, APT19, UNC1352, UNC27, APT1, UNC981, UNC581, UNC1559	e:\work\projects\trojan\client\dll\i386\Client.pdb

Figure 10: A selection of common keywords in PDB paths with groups and malware families observed and examples

PDB Path Showcase: Suspicious Developer Environment Terms

The keywords that are typically used to describe malware are strong enough to raise red flags, but there are other common terms or features in PDB paths that may signal that an executable is compiled in a non-enterprise setting. For example, any PDB path containing “Users” directory can tell you that the executable was likely compiled on Windows Vista/7/10 and likely does not represent an “official” or “commercial” development environment. The term “Users” is much weaker or

lower in fidelity than “shellcode” but as we demonstrate below, these terms are indeed present in lots of malware and can be used for weak detection signals.

PDB Path Term Prevalence

Term	**Families and Groups Observed**	**Example PDB Path**
Users	ABBEYROAD, AGENTTESLA, ANTFARM, AURORA, BEACON, BLACKDOG, BLACKREMOTE, BLACKSHADESRAT, BREAKDANCE, BROKEYOLK, BUSYFIB, CAMUBOT, CARDCAM, CATNAP, CHILDSPLAY, CITADEL, CROSSWALK, CURVEBALL, DARKCOMET, DARKMOON, DESERTFALCON, DESERTKATZ, DISPKILL, DIZZYLOG, EMOTET, FIDDLEWOOD, FIVERINGS, FLATTOP, FLUXXY, FOOTMOUSE, FORMBOOK, GOLDENCAT, GROK, GZIPDE, HAWKEYE, HIDDENTEAR, HIGHNOTE, HKDOOR, ICEDID, ICEFOG, ISMAGENT, KASPER, KOADIC, LUKEWARM, LUXNET, MOONRAT, NANOCORE, NETGRAIL, NJRAT, NUTSHELL, ONETOFOUR, ORCUSRAT, POISONIVY, POSHC2, QUASARRAT, QUICKHOARD, RADMIN, RANSACK, RAWHIDE, REMCOS, REVENGERAT, RYUK, SANDPIPE, SANDTRAP, SCREENTIME, SEEDOOR, SHADOWTECH, SILENTBYTES, SKIDHOOK, SLIMCAT, SLOWROLL, SOGU,	C:\Users\Yousef\Desktop\MergeFiles\Loader v0\Loader\obj\Release\Loader.pdb

	SOREGUT, SOURCANDLE, TREASUREHUNT, TRENDCLOUD, TRESOCHO, TRICKBOT, TRIK, TROCHILUS, TURNEDUP, TWINSERVE, UPCONTROL, UPDATESEE, URSNIF, WATERFAIRY, XHUNTER, XRAT, ZEUS APT5, APT10, APT17, APT33, APT34, APT35, APT36, APT37, APT39, APT40, APT41, FIN6, UNC284, UNC347, UNC373, UNC432, UNC632, UNC718, UNC757, UNC791, UNC824, UNC875, UNC1065, UNC1124, UNC1149, UNC1152, UNC1197, UNC1289, UNC1295, UNC1340, UNC1352, UNC1354, UNC1374, UNC1406, UNC1450, UNC1486, UNC1507, UNC1516, UNC1534, UNC1545, UNC1562	
ConsoleApplication WindowsApplication WindowsFormsApplication (Visual Studio default project names)	CROSSWALK, DESERTKATZ, DIZZYLOG, FIREPIPE, HIGHPRIEST, HOUDINI, HTRAN, KICKBACK, LUKEWARM, MOONRAT, NIGHTOWL, NJRAT, ORCUSRAT, REDZONE, REVENGERAT, RYUK, SEEDOOR, SLOAD, SOGU, TRICKBOT, TRICKSHOW APT1, APT34, APT36, FIN6, UNC251, UNC729, UNC1078, UNC1147, UNC1172, UNC1267, UNC1277, UNC1289, UNC1295, UNC1340, UNC1470, UNC1507	D:\Projects\ByPassAV\ConsoleApplication1\Release\ConsoleApplication1.pdb
New Folder	HOMEUNIX, KASPER, MOONRAT,	c:\Users\USA\Documents\Visual Studio 2008\Projects\New folder (2)\kasper\Release\kasper.pdb

	NANOCORE, NETWIRE, OZONERAT, POISONIVY, REMCOS, SKIDHOOK, TRICKBOT, TURNEDUP, URLZONE APT18, APT33, APT36, UNC53, UNC74, UNC672, UNC718, UNC1030, UNC1289, UNC1340, UNC1559	
Copy	DESERTFALCON, KASPER, NJRAT, RYUK, SOGU UNC124, UNC718, UNC757, UNC1065, UNC1215, UNC1225, UNC1289	D:\dll_Mc2.1mc\2.4\2.4.2 xor\zhu\dll_Mc - Copy\Release\shellcode.pdb
Desktop	AGENTTESLA, AVEO, BEACON, BUSYFIB, CHILDSPLAY, COATHOOK, DESERTKATZ, FIVERINGS, FLATTOP, FORMBOOK, GH0ST, GOLDENCAT, HIGHNOTE, HTRAN, IMMINENTMONITOR, KASPER, KOADIC, LUXNET, MOONRAT, NANOCORE, NETWIRE, NUTSHELL, ORCUSRAT, RANSACK, RUNBACK, SEEDOOR, SKIDHOOK, SLIMCAT, SLOWROLL, SOGU, TIERNULL, TINYNUKE, TRICKBOT, TRIK, TROCHILUS, TURNEDUP, UPDATESEE, WASHBOARD, WATERFAIRY, XRAT APT5, APT17, APT26, APT33, APT34, APT35, APT36, APT41, UNC53, UNC276, UNC308, UNC373, UNC534, UNC551,	C:\Users\Develop_MM\Desktop\sc_loader\Release\sc_loader.pdb

	UNC572, UNC672, UNC718, UNC757, UNC791, UNC824, UNC875, UNC1124, UNC1149, UNC1197, UNC1352	

Figure 11: A selection of common terms in PDB paths with groups and malware families observed and examples

PDB Path Showcase: Exploring Anomalies

Outside of keywords and terms, we discovered on a few uncommon (to us) features that may be interesting for future research and detection opportunities.

Non-ASCII Characters

PDB paths with any non-ASCII characters have a high ratio of malware to non-malware in our datasets. The strength of this signal is only because of a data bias in our malware corpus and in our client base. However, if this data bias is consistent, we can use the presence of non-ASCII characters in a PDB path as a signal that an executable merits further scrutiny. In organizations that operate primarily in the world of ASCII, we imagine this will be a strong signal. Below we express logic for this technique in Yara:

```
rule ConventionEngine_Anomaly_NonAscii{ meta: author = "@stvemillertime" strings: $pcre = /RSDS[\x00-\xFF]{20}[a-zA-Z]:\\[\x00-\xFF]{0,500}[^\x00-\x7F]{1,}[\x00-\xFF]{0,500}\.pdb\x00/ condition: (uint16(0) == 0x5A4D) and uint32(uint32(0x3C)) == 0x00004550 and $pcre}
```

Multiple Paths in a Single File

Each compiled program should only have one PDB path. The presence of multiple PDB paths in a single object indicates that the object has subfile executables, from which you may infer that the parent object has the capability to “drop” or “install” other files. While being a dropper or installer is not malicious on its own, having an alternative method of applying those classifications to file objects may be of assistance in surfacing malicious activity. In this example, we can also search for this capability using Yara:

```
rule ConventionEngine_Anomaly_MultiPDB_Triple{ meta: author = "@stvemillertime" strings: $anchor = "RSDS" $pcre = /RSDS[\x00-\xFF]{20}[a-zA-Z]:\\[\x00-\xFF]{0,200}\.pdb\x00/ condition: (uint16(0) == 0x5A4D) and uint32(uint32(0x3C)) == 0x00004550 and #anchor == 3 and #pcre == 3}
```

Outside of a Debug Section

When a file is compiled the entry for the debug information is in the IMAGE_DEBUG_DIRECTORY. Similar to seeing multiple PDB paths in a single file, when we see debug information inside an executable that does not have a debug directory, we can infer that the file has subfile executables, and is likely has dropper or installer functionality. In this rule, we use Yara’s convenient PE module to check the relative virtual address (RVA) of the IMAGE_DIRECTORY_ENTRY_DEBUG entry, and if it is zero we can presume that there is no debug entry and thus the presence of a CodeView PDB path indicates that there is a subfile.

```
rule ConventionEngine_Anomaly_OutsideOfDebug{ meta: author = "@stvemillertime" description = "Searching for PE files with PDB path keywords, terms or anomalies." strings: $anchor = "RSDS" $pcre = /RSDS[\x00-\xFF]{20}[a-zA-Z]:\\[\x00-\xFF]{0,200}\.pdb\x00/ condition: (uint16(0) == 0x5A4D) and uint32(uint32(0x3C)) == 0x00004550 and $anchor and $pcre and pe.data_directories[pe.IMAGE_DIRECTORY_ENTRY_DEBUG].virtual_address == 0}
```

Nulled Out PDB Paths

In the typical CodeView section, we would see the “RSDS” header, the 16-byte GUID, a 4-byte “age” and then a PDB path string. However, we’ve identified a significant number of malware samples where the embedded PDB path area is nulled out. In this example, we can easily see the CodeView debug structure, complete with header, GUID and age, followed by nulls to the end of the segment.

```
00147880: 52 53 44 53 18 c8 03 4e 8c 0c 4f 46 be b2 ed 9e : RSDS...N..OF....00147890: c1 9f a3 f4 01 00 00 00 00 00 00 00 00 00 00 00 : ................001478a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 : ................001478b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 : ................001478c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 : ................
```

There are a few possibilities of how and why a CodeView PDB path may be nulled out, but in the case of intentional tampering, for the purposes of removing toolmarks, the easiest way would be to manually overwrite the PDB path with

\x00s. The risk of manual editing and overwriting via hex editor is that doing so is laborious and may introduce other static anomalies such as checksum errors.

The next easiest way is to use a utility designed to wipe out debug artifacts from executables. One stellar example of this is "peupdate" which is designed not only to strip or fabricate the PDB path information, but can also recalculate the checksum, and eliminate Rich headers. Below we demonstrate use of peupdate to clear the PDB path.

```
C:\Users\user\Desktop>peupdate.exe tamper.exe -c
processing "tamper.exe"
    PE32 signature found
    image checksum is 0x00000000 / calc 0x00001CCA (CHECKSUM OK)
    debug directory found at rva 0x27f0 (size=0x1c) in .text section
    CvPDB 7.0 sig={93D95326-8126-49AF-8D9C-069384422B20} age=0x1
    PDB path currently set to "D:\smiller\projects\super_evil_stuff\shellcode\Te
st\obj\Debug\Test.pdb"
cleared PDB path buffer of 260 bytes

C:\Users\user\Desktop>peupdate.exe tamper.exe
processing "tamper.exe"
    PE32 signature found
    image checksum is 0x00000000 / calc 0x0000C408 (CHECKSUM OK)
    debug directory found at rva 0x27f0 (size=0x1c) in .text section
    CvPDB 7.0 sig={93D95326-8126-49AF-8D9C-069384422B20} age=0x1
PDB path currently set to ""
```

Figure 12: Using peupdate to clear the PDB path information from a sample of malware

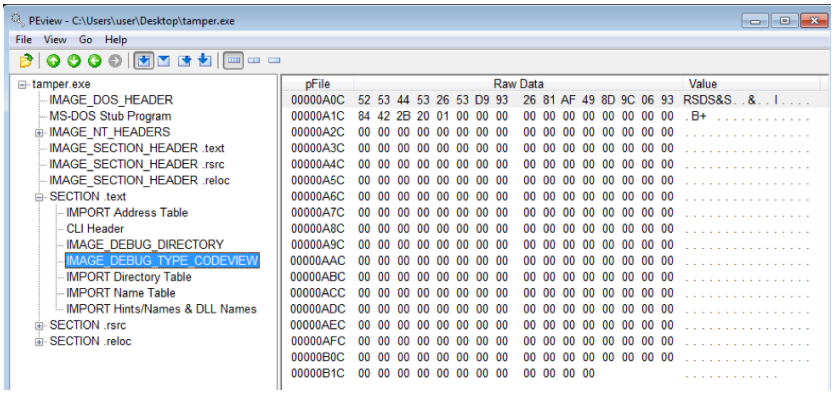

Figure 13: The peupdate tampered malware as shown in the PEview utility. We see the CodeView section is still present but the PDB path value has been cleared out

PDB Path Anomaly Prevalence

Anomaly	**Families and Groups Observed**	**Examples**
Non-Ascii Characters	008S, AGENTTESLA, BADSIGN, BAGELBYTE, BIRDSEED, BLACKCOFFEE, CANNONFODDER, CARDDROP, CEREAL, CHILDSPLAY, COATHOOK, CURVEBALL, DANCEPARTY, DIMWIT, DIZZYLOG, EARTHWORM, EIGHTONE, ELISE, ELKNOT, ENFAL, EXCHAIN, FANNYPACK, FLOWERPOT, FREELOAD, GH0ST, GINGERYUM, GLASSFLAW, GLOOXMAIL,	I:\RControl\小工具\123\判断加载着\Release\判断加载着.pdb

	GOLDENCAT, GOOGHARD, GOOGONE, HANDSTAMP, HELLWOOD, HIGHNOON, ICEFOG, ISHELLYAHOO, JAMBOX, JIMA, KRYPTONITE, LIGHTSERVER, LOCKLOAD, LOKIBOT, LOWLIGHT, METASTAGE, NETWIRE, PACMAN, PARITE, POISONIVY, PIEDPIPER, PINKTRIP, PLAYNICE, QUASARRAT, REDZONE, SCREENBIND, SHADOWMASK, SHORTLEASH, SIDEWINDER, SLIMEGRIME, SOGU, SUPERMAN, SWEETBASIL, TEMPFUN, TRAVELNET, TROCHILUS, URSNIF, VIPER, VIPSHELLAPT1, APT2, APT3, APT5, APT6, APT9, APT10, APT14, APT17, APT18, APT20, APT21, APT23, APT24, APT24, APT24, APT26, APT31, APT33, APT41, UNC20, UNC27, UNC39, UNC53, UNC74, UNC78, UNC1040, UNC1078, UNC1172, UNC1486, UNC156, UNC208, UNC229, UNC237, UNC276, UNC293, UNC366, UNC373, UNC451, UNC454, UNC521, UNC542, UNC551, UNC556, UNC565, UNC584, UNC629, UNC753, UNC794, UNC798, UNC969	
Multi Path in Single File	AGENTTESLA, BANKSHOT, BEACON, BIRDSEED, BLACKBELT, BRIGHTCOMB, BUGJUICE, CAMUBOT, CARDDROP, CETTRA,	Single Sample of TRICKBOT:D:\MyProjects\spreader\Release\spreader_x86.pdbD:\MyProjects\spreader\Release\ssExecutor_

	CHIPSHOT, COOKIECLOG, CURVEBALL, DARKMOON, DESERTFALCON, DIMWIT, ELISE, EXTRAMAYO, FIDDLELOG, FIDDLEWOOD, FLUXXY, FON, GEARSHIFT, GH0ST, HANDSTAMP, HAWKEYE, HIGHNOON, HIKIT, ICEFOG, IMMINENTMONITOR, ISMAGENT, KASPER, KAZYBOT, LIMITLESS, LOKIBOT, LUMBERJACK, MOONRAT, ORCUSRAT, PLANEDOWN, PLANEPATCH, POSEIDON, POSHC2, PUBNUBRAT, PUPYRAT, QUASARRAT, RABBITHOLE, RATVERMIN, RAWHIDE, REDTAPE, RYUK, SAKABOTA, SAMAS, SAMAS, SEEGAP, SEEKEYS, SKIDHOOK, SOGU, SWEETCANDLE, SWEETTEA, TRAVELNET, TRICKBOT, TROCHILUS, UPCONTROL, UPDATESEE, UROBUROS, WASHBOARD, WHITEWALK, WINERACK, XTREMERAT, ZXSHELL	
	APT1, APT2, APT17, APT5, APT20, APT21, APT26, APT34, APT36, APT37, APT40, APT41, UNC27, UNC53, UNC218, UNC251, UNC432, UNC521, UNC718, UNC776, UNC875, UNC878, UNC969, UNC1031, UNC1040, UNC1065, UNC1092, UNC1095, UNC1166, UNC1183, UNC1289, UNC1374,	

	UNC1443, UNC1450, UNC1495	
Outside of Debug Section	ABBEYROAD, AGENTTESLA, BEACON, BLACKSHADESRAT, CHIMNEYDIP, CITADEL, COOKIECLOG, COREBOT, CRACKSHOT, DAYJOB, DIRTCHEAP, DIZZYLOG, DUSTYSKY, EARTHWORM, EIGHTONE, ELISE, EXTRAMAYO, FRONTWHEEL, GELCAPSULE, GH0ST, HAWKEYE, HIGHNOON, KAYSLICE, LEADPENCIL, LOKIBOT, METASTAGE, METERPRETER, MURKYTOP, NUTSHELL, ORCUSRAT, OUTLOOKDUMP, PACMAN, POISONIVY, PLANEPATCH, PONY, PUPYRAT, RATVERMIN, SAKABOTA, SANDTRAP, SEADADDY, SEEDOOR, SHORTLEASH, SOGU, SOULBOT, TERA, TIXKEYS, UPCONTROL, WHIPSNAP, WHITEWALK, XDOOR, XTUNNEL APT5, APT6, APT9, APT10, APT17, APT22, APT24, APT26, APT27, APT29, APT30, APT34, APT35, APT36, APT37, APT40, APT41, UNC20, UNC27, UNC39, UNC53, UNC69, UNC74, UNC105, UNC124, UNC125, UNC147, UNC213, UNC215, UNC218, UNC227, UNC251, UNC276,	

	UNC282, UNC307, UNC308, UNC347, UNC407, UNC565, UNC583, UNC587, UNC589, UNC631, UNC707, UNC718, UNC775, UNC776, UNC779, UNC842, UNC869, UNC875, UNC875, UNC924, UNC1040, UNC1080, UNC1148, UNC1152, UNC1225, UNC1251, UNC1428, UNC1450, UNC1486, UNC1575	
Nulled Out PDB Paths	HIGHNOON, SANNY, PHOTO, TERA, SOYSAUCE, VIPER, FIDDLEWOOD, BLACKDOG, FLUSHSHOW, NJRAT, LONGCUT APT41, UNC776, UNC229, UNC177, UNC1267, UNC878, UNC1511	

Figure 14: A selection of anomalies in PDB paths with groups and malware families observed and examples

PDB Path Showcase: Outliers, Oddities, Exceptions and Other Shenanigans

The internet is a weird place, and at a big enough scale, you end up seeing things that you never thought you would. Things that deviate from the norms, things that shirk the standards, things that utterly defy explanation. We expect PDB paths to look a certain way, but we've run across several samples that did not, and we're not always sure why. Many of these samples below may be results of errors, corruption, obfuscation, or various forms of intentional manipulation. We're demonstrating them here to show that if you are attempting PDB path parsing or detection, you need to understand the variety of paths in the wild and prepare for shenanigans galore. Each of these examples are from confirmed malware samples.

Shenanigan	**Example PDB Paths**
Unicode error	Text Path: C^\Users\DELL\Desktop\interne.2.pdb Raw Path: 435E5C55 73657273 5C44454C 4C5C4465 736B746F 705C696E 7465726E 6598322E 706462 Text Path: Cj\Users\hacker messan\Deskto \Server111.pdb Raw Path: 436A5C55 73657273 5C686163 6B657220 6D657373 616E5C44 65736B74 6FA05C53 65727665 72313131 2E706462
Nothing but space	Text Path: Full Raw: 52534453 7A7F54BF BAC9DE45 89DC995F F09D2327 0A000000 20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202000
Spaced out	Text Path: D:\ .pdb Full Raw: 52534453 A7FBBBFE 5C41A545 896EF92F 71CD1F08 01000000 443A5C20 20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020 2E706462 00

Nothin' but null	Text Path: <null bytes only> Full Raw: 52534453 97272434 3BACFA42 B2DAEE99 FAB00902 01000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
Random characters	Text Path: Lmd9knkjasdLmd9knkjasLmd9knkAaGc.pdb
Random path	Text Path: G:\givgLxNzKzUt\TcyaxiavDCiu\bGGiYrco\QNfWgtSs\auaXaWyjgmPqd.pdb
Word soup	Text Path: c:\Busy\molecule\Blue\Valley\Steel\King\enemy\Himyard.pdb
Mixed doubles	Text Path: C::\\QQQQQQQQ\VVVVVVVVVVVVVVVVV.pdb
Short	Text Path: 1.pdb
No .pdb	Text Path: a Full Raw: 52534453 ED86CA3D 6C677946 822E668F F48B0F9D 01000000 6100
Long and weird with repeated character	Text Path: ªªªªªªªªªªªªªªªªªªªªtinjs\aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaae.pdb Full Raw: 52534453 DD947C2F 6B32544C 8C3ACB2E C7C39F45 01000000 AAAAAAAA AAAAAAAA AAAAAAAA AAAAAAAA AAAAAAAA 74696E6A 735C6161 61616161 61616161 61616161 61616161 61616161 61616161 61616161 61616161 61616161 61616161 61616161 61616161 61616161 61616161 61616161 61616161 61616161 61616161 61616161 652E7064 6200
No idea	Text Path: n:.Lí..×ÖòÒ. Full Raw: 52534453 5A2D831D CB4DCF1E 4A05F51B 94992AA0 B7CFEE32 6E3AAD4C ED1A1DD7 D6F2D29E 00
Forward slashes and no drive letter	Text Path: /Users/user/Documents/GitHub/SharpWMI/SharpWMI/obj/Debug/SharpWMI.pdb
Network share	Text path: \\vmware-host\shared folders\Decrypter\Decrypter\obj\Release\Decrypter.pdb
Non-Latin drive letter	We haven't seen this yet, but it's only a matter of time until you can have an emoji as a drive letter.

Figure 15: A selection of PDB paths shenanigans with examples

Betwixt Nerf Herders and Elite Operators

There are many differences between apex threat actors and the rest, even if all successfully perform intrusion operations. Groups that exercise good OPSEC in some campaigns may have bad OPSEC in others. APT36 has hundreds of leaked PDB paths, whereas APT30 has a minimal PDB path footprint, while APT38 is a ghost.

When PDB paths are present, the types of keywords, terms, and other string items present in PDB paths are all on a spectrum of professionalism and sophistication. On one end we're seeing "njRAT-FUD 0.3" and "1337 h4ckbot" and on the other end we're seeing "minidionis" and "msrstd".

The trendy critique of string-based detection goes something like "advanced adversaries *would never* act so carelessly; they'll obfuscate and evade your naïve and brittle signatures." In the tables above for PDB path keywords, terms and anomalies, we think we've shown that bona fide APT/FIN groups, state-sponsored adversaries, and the best-of-the-best attackers do sometimes slip up and give us an opportunity for detection.

Let's call out some specific examples from boutique malware from some of the more advanced threat groups.

Equation Group

Some Equation Group samples show full PDB paths that indicate that some of the malware was compiled in debug mode on workstations or virtual machines used for development.

- c:\users\rmgree5\co\standalonegrok_2.1.1.1\gk_driver\gk_sa_driver\objfre_wnet_amd64\amd64\SaGk.pdb

Other Equation Group samples have partially qualified PDB paths that represent something less obvious. These standalone PDB names may reflect a more *tailored*, multi-developer environment, where it wouldn't make sense to specify a fully qualified PDB path for a single developer system. Instead, the linker is instructed to write only the PDB file name in the built executable. Still, these PDB paths are unique to their malware samples:

- tdip.pdb
- volrec.pdb
- msrstd.pdb

Regin

Deeming a piece of malware a "backdoor" is increasingly passé. Calling a piece of malware an "implant" is the new hotness, and the general public may be adopting this nouveau nomenclature long after purported Western governments. In this component of the Regin platform, we see a developer that was way ahead of the curve:

- C:\dev\k1svn\dsd\Implants\WarriorPride\production2.0\package\E_Wzowski\Release\E_Qwerty.pdb

APT29

Let's not forget APT29, whose brazen worldwide intrusion sprees often involve pieces of creative, elaborate, and stealthy malware. APT29 is amongst the better groups at staying quiet, but in thousands of pieces of malware, these normally disciplined operators did leak a few PDB paths such as:

- c:\Users\developer\Desktop\unmodified_netimplant\minidionis\minidionis\obj\Debug\minidionis.pdb
- C:\Projects\nemesis-gemina\nemesis\bin\carriers\ezlzma_x86_exe.pdb

Even when the premier outfits don't use the glaring keywords, there may still be some string terms, anomalies and *unique values* present in PDB paths that each represent an opportunity for detection.

ConventionEngine

We extract and index all PDB paths from all executables so we can easily search and spelunk through our data. But not everyone has it that easy, so we cranked out a quick collection of nearly 100 Yara rules for PDB path keywords, terms and anomalies that we believe researchers and analysts can use to detect evil. We named this collection of rules "ConventionEngine" after the industry jokes that security vendors like to talk about their elite detection "engines," but behind the green curtain they're all just a code spaghetti mess of scripts and signatures, which this absolutely started as.

Instead of tight production "signatures," you can think of these as "weak signals" or "discovery rules" that are meant to build haystacks of varying size and fidelity for analysts to hunt through. Those rules with a low signal-to-noise ratio (SNR) could be fed to automated systems for logging or contextualization of file objects, whereas rules with a higher SNR could be fed directly to analysts for review or investigation.

Our adversaries are human. They err. And when they do, we can catch them. We are pleased to release ConventionEngine rules for anyone to use in that effort. Together these rules cover samples from over 300 named malware families, hundreds of unnamed malware families, 39 different APT and FIN threat groups, and over 200 UNC (uncategorized) groups of activity.

We hope you can use these rules as templates, or as starting points for further PDB path detection ideas. There's plenty of room for additional keywords, terms, and anomalies. Be advised, whether for detection or hunting or merely for context, you will need to tune and add additional logic to each of these rules to make the size of the resulting haystacks appropriate for your purposes, your operations and the technology within your organization. When judiciously implemented, we believe these rules can enrich analysis and detect things that are missed elsewhere.

PDB Paths for Intelligence Teams

Gettin' Lucky with APT31

During an incident response investigation, we found an APT31 account on Github being used for staging malware files and for malware communications. The intrusion operators using this account weren't shy of putting full code packages right into the repositories and we were able to recover actual PDB files associated with multiple malware ecosystems. Using the actual PDB files, we were able to see the full directory paths of the raw malware source code, representing a considerable intelligence gain about the malware original development environment. We used what we found in the PDB itself to search for other files related to this malware author.

Finding Malware Source Code Using PDBs

Malware PDBs themselves are easier to find than one may think. Sure, sometimes the authors are kind enough to leave everything up on Github. But there are some other occasions too: sometimes malware source code will get inadvertently flagged by antivirus or endpoint detection and response (EDR) agents; sometimes malware source code will be left in open directories; and sometimes malware source code will get uploaded to the big malware repositories.

You can find malware source code by looking for things like Visual Studio solution files, or simply with Yara rules looking for PDB files in archives that have some non-zero detection rate or other metadata that raises the likelihood that some component in the archive is indeed malicious.

```
 rule PDB_Header_V2
{
    meta:
        author="@stvemillertime"
        description = "This looks for PDB files based on headers.
    strings:
        //$string = "Microsoft C/C++ program database 2.00"
        $hex = {4D696372 6F736F66 7420432F 432B2B20 70726F67 72616D20 64617461 62617365 20322E30 300D0A}
    condition:
        $hex at 0
rule PDB_Header_V7
{
    meta:
        author="@stvemillertime"
        description = "This looks for PDB files based on headers.
    strings:
        //$string = "Microsoft C/C++ MSF 7.00"
        $hex = {4D696372 6F736F66 7420432F 432B2B20 4D534620 372E3030}
    condition:
        $hex at 0
}
```

PDB Paths for Offensive Teams

FireEye has confirmed individual attribution to bona fide threat actors and red teamers based in part on leaked PDB paths in malware samples. The broader analyst community often uses PDB paths for clustering and pivoting to related malware families and while building a case for attribution, tracking, or pursuit of malware developers. Naturally, red team and offensive operators should be aware of the artifacts that are left behind during the compilation process and abstain from compiling with symbol generation enabled – basically, remember to practice good OPSEC on your implants. That said, there is an opportunity for creating artificial PDB paths should one wish to intentionally introduce this artifact.

Making PDB Paths Appear More "Legitimate"

One notable differentiator between malware and non-malware is that malware is typically not developed in an "enterprise" or "commercial" software development setting. The difference here is that in large development settings, software engineers are working on big projects together through productivity tools, and the software is constantly updated and rebuilt through automated "continuous integration" (CI) or "continuous delivery" (CD) suites such as Jenkins and TeamCity. This means that when PDB paths are present in legitimate enterprise software packages, they often have toolmarks showing their compile path on a CI/CD build server.

Here are some examples of PDB paths of legitimate software executables built in a CI/CD environment:

- D:\Jenkins\workspace\QA_Build_5_19_ServerEx_win32_buildoutput\ServerEx\Win32\Release_symbols\keysvc.pdb
- D:\bamboo-agent-home\xml-data\build-dir\MC-MCSQ1-JOB1\src\MobilePrint\obj\x86\Release\MobilePrint.pdb
- C:\TeamCity\BuildAgent\work\714c88d7aeacd752\Build\Release\cs.pdb

We do not discount the fact that some malware developers are using CI/CD build environments. We know that some threat actors and malware authors are indeed adopting contemporary enterprise development processes, but malware PDBs like this example are extraordinarily rare:

- c:\users\builder\bamboo~1\xml-data\build-~1\trm-pa~1\agent\window~1\rootkit\Output\i386\KScan.pdb

Specifying Custom PDB Paths in Visual Studio

Specifying a custom path for a PDB file is not uncommon in the development world. An offensive or red team operator may wish to specify a fake PDB path and can do so easily using compiler linking options.

As our example malware author "smiller" learns and hones their tradecraft, they may adopt a stealthier approach and choose to include one of those more "legitimate" looking PDB paths in new malware compilations.

Take smiller's example malware project located at the path:

D:\smiller\projects\offensive_loaders\shellcode\hello\hellol\

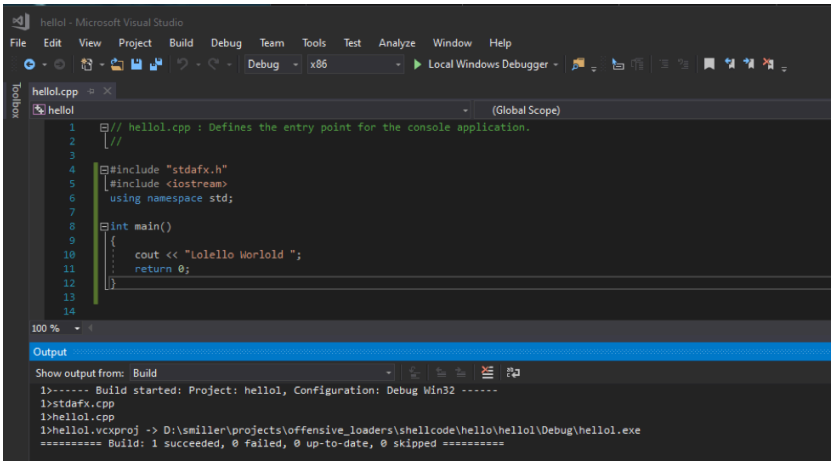

Figure 16: hellol.cpp code shown in Visual Studio with debug build information

This project compiled in Debug configuration by default places both the hellol.exe file and the hellol.pdb file under

D:\smiller\projects\offensive_loaders\shellcode\hello\hellol\Debug\

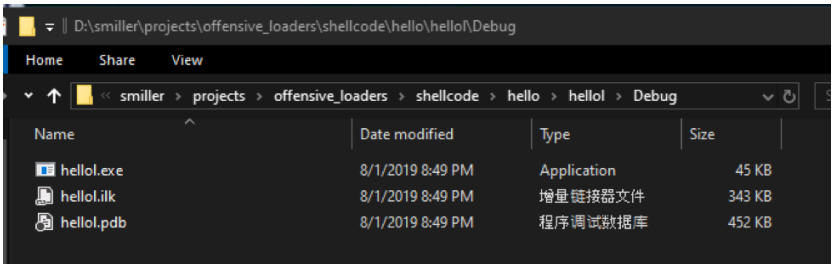

Figure 17: hellol.exe and hellol.pdb, compiled by debug configuration default into its resident folder

It's easy to change the properties of this project and manually specify the generation path of the PDB file. From the Visual Studio taskbar, select **Project** > **Properties**, then in the side pane select **Linker** > **Debugging** and fill the option box for "**Generate Program Database File.**" This option accepts Visual Studio macros so there is plenty of flexibility for scripting and creating custom build configurations for falsifying or randomizing PDB paths.

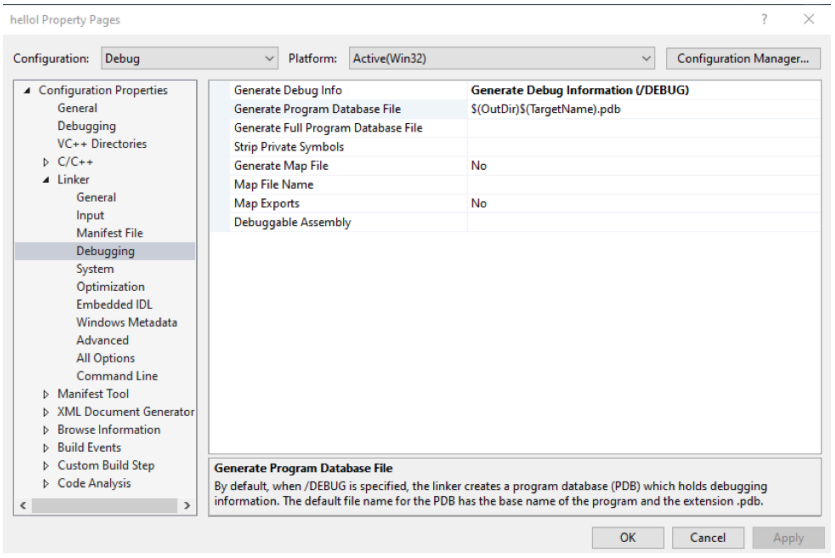

Figure 18: hellol project Properties showing defaults for the PDB path

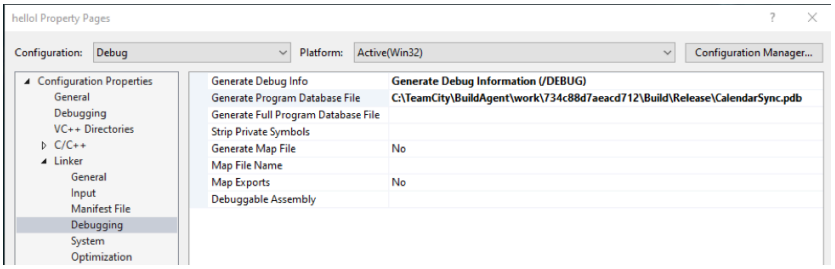

Figure 19: hellol project Properties now showing a manually specified path for the (fake) PDB path

When we examine the raw ConsoleApplication1.exe, we can see at the byte level that the linker has included debug information in the executable specifying our designated PDB path, which of course is not real. Or if built at the command line, you could specify /PDBALTPATH which can create a PDB file name that is does not rely on the file structure of the build computer.

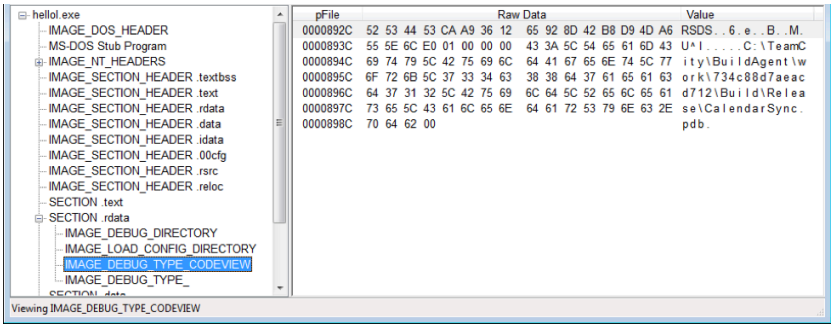

Figure 20: Rebuilt hellol.exe as seen through the PEview utility, which shows us the fake PDB path in the IMAGE_DEBUG_TYPE_CODEVIEW directory of the executable

An offensive or red team operator could intentionally include a PDB path in a piece of malware, making the executable appear to be compiled on a CI/CD server which could help the malware fly under the radar. Additionally, an operator could include a PDB path or strings associated with a known malware family or threat group to confound analysts. Why not throw in a small homage to one of your favorite malware operators or authors, such as the infamous APT33 persona xman_1365_x? Or perhaps throw in a "\Homework\CS1101\" to make the activity seem more academic? For whatever reason, if there is PDB manipulation to be done, it is generally doable with common software development tools.

The Glory and the Nothing of a (Malware) Name

In the context of PDB paths and malware author naming conventions, it is important to acknowledge the interdependent (and often circular) nature of "offense" and "defense." Which came first, a defender calling a piece of malware a "trojan" or a malware author naming their code project a "trojan"? Some malware is inspired by prior work. An author names a code project "MIMIKATZ", and years later there are hundreds of related projects and scripts with derivative names.

Although definitions may vary, we see that both the offensive and defensive sides characterize the functionality or role of a piece of malware using much of the same vernacular and inspiration. We suspect this began with "virus" and that the array of granular, descriptive terms will continue to grow as public discourse advances the malware taxonomy. Who would have suspected that *how* we talked about malware would ultimately lead to the possibility detecting it? After all, would a rootkit by any other name be as evil? Somewhere, a scholar is beaming with wonder at the intersection of malware and linguistics.

Conclusions

If by now you're thinking this is all kind of silly, don't worry, you're in good company. PDB paths are indeed a wonky attribute of a file. The mere presence of these paths in an executable is by no means evil, yet when these paths are present in pieces of malware, they usually represent acts of operational indiscretion. The idea of detecting malware based on PDB paths is kind of like detecting a robber based on what type of hat a person is wearing, if they're wearing one at all.

We have been historically successful in using PDB paths mostly as an analytical pivot, to help us cluster malware families and track malware developers. When we began to study PDB paths holistically, we noticed that many malware authors were using many of the same naming conventions for their folders and project files. They were naming their malware projects after the functionality of the malware itself, and they routinely label their projects with unique, descriptive language.

We found that many malware authors and operators leaked PDB paths that described the functionality of the malware itself and gave us insight into the development environment. Furthermore, outside of the descriptors of the malware development files and environment, when PDB files are present, we identified anomalies that help us identify files that are more likely to be circumstantially interesting. There is room for red team and offensive operators to improve their tradecraft by falsifying PDB paths for purposes of stealth or razzle-dazzle.

We remain optimistic that we can squeeze some juice from PDB paths when they are present. A survey of about 2200 named malware families (including all samples from 41 APT and 10 FIN groups and a couple million other uncategorized executables) shows that PDB paths are present in malware about five percent of the time. Imagine if you could have a detection "backup plan" for five plus percent of malware, using a feature that is itself inherently non-malicious. That's kind of cool, right?

Future Work on Scaling PDB Path Classification

Our ConventionEngine rule pack for PDB path keyword, term and anomaly detection has been fun and found tons of malware that would have otherwise been missed. But there are a lot of PDB paths in malware that do not have such obvious keywords, and so our manual, cherry-picking, and extraordinarily laborious approach doesn't scale.

Stay tuned for the next part of our blog series! In **Part Deux**, we explore scalable solutions for PDB path feature generalization and approaches for classification. We believe that data science approaches will better enable us to surface PDB paths with unique and interesting values and move towards a classification solution without any rules whatsoever.

Recommended Reading and Resources

Inspiring Research

- http://www.hexacorn.com/blog/2013/05/08/and-the-most-popular-windows-account-for-compiling-malware-is/
- https://securelist.com/operation-shadowhammer-a-high-profile-supply-chain-attack/90380/
- https://www.mandiant.com/resources/reports/apt41-double-dragon-dual-espionage-and-cyber-crime-operation

Debugging and Symbols

- https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/symbols
- https://docs.microsoft.com/en-us/windows/win32/dxtecharts/debugging-with-symbols
- http://www.debuginfo.com/articles/debuginfomatch.html

Debug Directory and CodeView

- https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image_debug_directory
- https://github.com/dotnet/corefx/blob/master/src/System.Reflection.Metadata/specs/PE-COFF.md

Debugging and Visual Studio

- https://docs.microsoft.com/en-us/cpp/build/reference/debug-generate-debug-info?view=vs-2019
- https://docs.microsoft.com/en-us/visualstudio/debugger/specify-symbol-dot-pdb-and-source-files-in-the-visual-studio-debugger?view=vs-2019
- https://docs.microsoft.com/en-us/visualstudio/debugger/remote-debugging?view=vs-2019

PDB File Structure

- https://github.com/microsoft/microsoft-pdb
- https://docs.microsoft.com/en-us/windows/win32/debug/symbol-files
- https://github.com/microsoft/microsoft-pdb/blob/master/docs/ExternalResources.md
- http://www.godevtool.com/Other/pdb.htm

PDB File Tools

- Peupdate: http://bytepointer.com/tools/index.htm#peupdate

Posted in

- Threat Intelligence
- Security & Identity
