Extracting Cobalt Strike from Windows Error Reporting

bmcder.com/blog/extracting-cobalt-strike-from-windows-error-reporting

April 19, 2022

Cobalt StrikeDebuggingWindows Internals
19 Apr
Written By Blake .

Introducing malware into a network can often cause problems. You're adding an unknown software into an unknown environment and while there's a lot of testing put into preventing application crashes, it cannot be guaranteed. Often during an investigations, we'll see surges in application crashes following the actors presence due to abnormal behaviours on the network.

Windows Error Reporting is the native control for handling application crashes, leaving behind some handy logging and dumps that can help track an actors presence. This entry will go through how we can extract Cobalt Strike from a Windows Error Reporting process dump. This can be a great method of detecting abnormal behaviour after a process crashed.

What is Windows Error Reporting?

Windows Error Reporting (WER) acts as a debugging layer when an application crashes or hangs. Depending on the process specific settings, this can provide some useful troubleshooting information including:

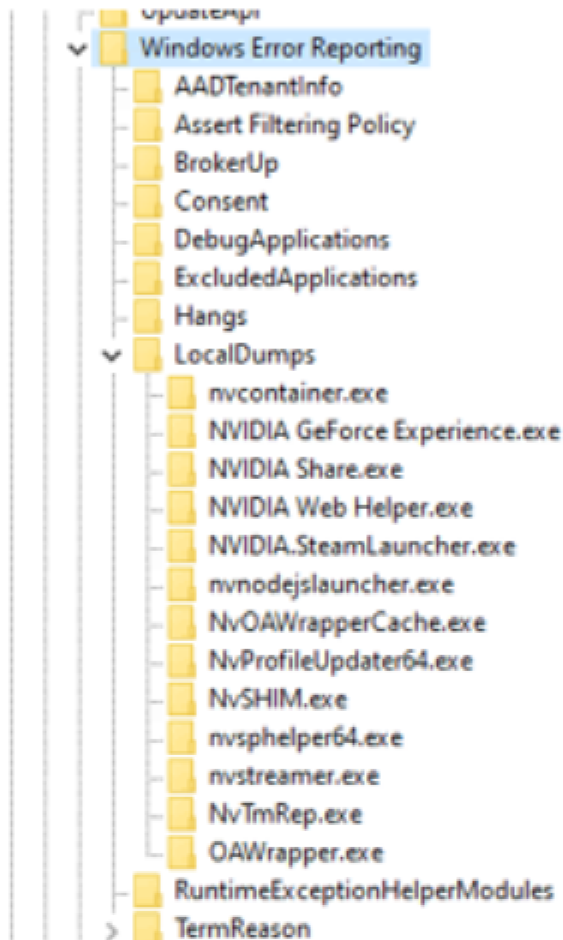
1. A report on the state of the application when it crashed,
2. Process Dump,
3. Digital Certificate and Application Combability references.

The report is the only guaranteed file created with Windows Error Reporting. These files get stored within two paths by default, one for more recent entries and one for more historic entries. They have the following paths:

1. C:\ProgramData\Microsoft\Windows\WER\ReportQueue
2. C:\ProgramData\Microsoft\Windows\WER\ReportArchive

The process dumps can be configured to allow for process specific setting. This can be particularly useful when you know an actor is reliably crashing a specific process, like a web shell crashing w3wp. You can access the process specific settings at:

`HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\Windows Error Reporting\LocalDumps\`



Within the LocalDumps key, you can configure:

1. DumpCount: # of dumps to keep before rollover.
2. DumpFolder: Where to output the process dumps.
3. DumpType:

    0. Custom Dump

    1. Mini Dump

    2. Full Dump (Process)

Name	Type	Data
(Default)	REG_SZ	(value not set)
DumpCount	REG_DWORD	0x0000000f (15)
DumpFolder	REG_EXPAND_SZ	%PROGRAMDATA%\NVIDIA Corporation\CrashDu...
DumpType	REG_DWORD	0x00000001 (1)

Enhancing the data collected for that single process can give you an analytic edge on the actors actions, while also ensuring you don't fill up the hard drive.

The Scenario

Note: I've replicated this scenario from a previous investigation. If you'd like a copy of the dump, let me know! :)

During an investigation, I had YARA flag some Cobalt Strike rules on an Windows Error Reporting dump for a native Windows process. This process was set to collect a full process dump, so we had about 400MB worth of memory to dig through.

```
PS C:\Exclusion> .\yara64.exe .\cs.yar .\explorer.exe.3444.dmp
HKTL_CobaltStrike_Beacon_Strings .\explorer.exe.3444.dmp
HKTL_CobaltStrike_Beacon_4_2_Decrypt .\explorer.exe.3444.dmp
HKTL_Win_CobaltStrike .\explorer.exe.3444.dmp
PS C:\Exclusion> dir .\explorer.exe.3444.dmp


    Directory: C:\Exclusion


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        18/04/2022   5:16 PM      409310176 explorer.exe.3444.dmp
```

Looking at the report.wer, there's no interesting module loads. This makes sense though, if it is Cobalt Strike, it's going to be reflectively loaded. Once we load the process into windbg, we can cross check the loaded modules in the report.wer against the loaded module addresses.

Windbg

The Error

First command you should run when investigating a crash dump is "!analyze -v". This will give you some basic analytics around why the command crashed and can give a better understanding of what the actor was doing.

```
ERROR_CODE: (NTSTATUS) 0xc0000005 - The instruction at 0x%p referenced memory at 0x%p. The memory could not be %s.

EXCEPTION_CODE_STR:  c0000005

EXCEPTION_PARAMETER1:  0000000000000008

EXCEPTION_PARAMETER2:  ffffffffffffffff
```

From the NTSTATUS Code 0xC0000005, we know that there was a access violation pointing to 0xffffffffffffffff.

Finding the YARA hit

We can do a search for the yara strings that hit on the process dump to get a region where Cobalt Strike might be loaded. Doing a search for three of the strings, we can find they're all located within two similar memory regions.

```
0:065> s -a 0 L?80000000 "Started service"
00000000`007cbd7b  53 74 61 72 74 65 64 20-73 65 72 76 69 63 65 20  Started service
00000000`027bcd7b  53 74 61 72 74 65 64 20-73 65 72 76 69 63 65 20  Started service
0:065> s -a 0 L?80000000 "%02d/%02d/%02d %02d:%02d:%02d"
00000000`007cb72c  25 30 32 64 2f 25 30 32-64 2f 25 30 32 64 20 25  %02d/%02d/%02d %
00000000`007cb758  25 30 32 64 2f 25 30 32-64 2f 25 30 32 64 20 25  %02d/%02d/%02d %
00000000`027bc72c  25 30 32 64 2f 25 30 32-64 2f 25 30 32 64 20 25  %02d/%02d/%02d %
00000000`027bc758  25 30 32 64 2f 25 30 32-64 2f 25 30 32 64 20 25  %02d/%02d/%02d %
0:065> s -a 0 L?80000000 "%s as %s\\%s: %d"
00000000`007cb700  25 73 20 61 73 20 25 73-5c 25 73 3a 20 25 64 00  %s as %s\%s: %d.
00000000`007cbc4d  25 73 20 61 73 20 25 73-5c 25 73 3a 20 25 64 00  %s as %s\%s: %d.
00000000`027bc700  25 73 20 61 73 20 25 73-5c 25 73 3a 20 25 64 00  %s as %s\%s: %d.
00000000`027bcc4d  25 73 20 61 73 20 25 73-5c 25 73 3a 20 25 64 00  %s as %s\%s: %d.
```

We can do a search for MZ headers within that memory those MZ address ranges.

```
0:065> s -a 0 L?03000000 "This program cannot"
00000000`007a004e  54 68 69 73 20 70 72 6f-67 72 61 6d 20 63 61 6e  This program can
00000000`007f004e  54 68 69 73 20 70 72 6f-67 72 61 6d 20 63 61 6e  This program can
00000000`0085004e  54 68 69 73 20 70 72 6f-67 72 61 6d 20 63 61 6e  This program can
00000000`0279004e  54 68 69 73 20 70 72 6f-67 72 61 6d 20 63 61 6e  This program can
00000000`027e004e  54 68 69 73 20 70 72 6f-67 72 61 6d 20 63 61 6e  This program can
00000000`0281004e  54 68 69 73 20 70 72 6f-67 72 61 6d 20 63 61 6e  This program can
00000000`0291004e  54 68 69 73 20 70 72 6f-67 72 61 6d 20 63 61 6e  This program can
00000000`02a2004e  54 68 69 73 20 70 72 6f-67 72 61 6d 20 63 61 6e  This program can
00000000`02c7004e  54 68 69 73 20 70 72 6f-67 72 61 6d 20 63 61 6e  This program can
00000000`02c8004e  54 68 69 73 20 70 72 6f-67 72 61 6d 20 63 61 6e  This program can
00000000`02da004e  54 68 69 73 20 70 72 6f-67 72 61 6d 20 63 61 6e  This program can
0:065> s -a 0 L?80000000 "%02d/%02d/%02d %02d:%02d:%02d"
00000000`007cb72c  25 30 32 64 2f 25 30 32-64 2f 25 30 32 64 20 25  %02d/%02d/%02d %
00000000`007cb758  25 30 32 64 2f 25 30 32-64 2f 25 30 32 64 20 25  %02d/%02d/%02d %
00000000`027bc72c  25 30 32 64 2f 25 30 32-64 2f 25 30 32 64 20 25  %02d/%02d/%02d %
00000000`027bc758  25 30 32 64 2f 25 30 32-64 2f 25 30 32 64 20 25  %02d/%02d/%02d %
```

We can pull out the two headers that are just before our strings. Now that we have the address of the DLLs (address minus 0x4e), and surprise surprise, it doesn't line up with any of our loaded modules.

When we look at the DLL address in memory panel and instantly see three of our main PE executable signs:

1. MZ header
2. This program cannot be run in DOS mode
3. PE header

```
Command    Disassembly    Memory 0
Addre Command 00`02790000
0000000002790000 E589485552415A4D 4800000020EC8148 MZARUH.?H.? ...H
0000000002790010 8948FFFFFEA1D8D 00015FF4C38148DF ..????H.?H.??_..
0000000002790020 56A2B5F0B841D3FF 89485A0000000468 ??A????Vh....ZH.
0000000002790030 000000000D0FFF9 000000F800000000 ???.........?...
0000000002790040 CD09B4000EBA1F0E 685421CD4C01B821 ..?..?.?!?.L?!Th
0000000002790050 72676F7270207369 6F6E6E6163206D61 is program canno
0000000002790060 6E75722065622074 20534F44206E6920 t be run in DOS
0000000002790070 0A0D0D2E65646F6D 0000000000000024 mode....$.......
0000000002790080 3ABCB53169D2D475 3ABCB5313ABCB531 u??i1??:1??:1??:
0000000002790090 3ABCB5303A725B57 3ABCB5A93A6E5A12 W[r:0??:.Zn:???:
00000000027900A0 3ABCB5303A7B15AF 3ABCB5183A7373C0 ?.{:0??:?ss:.??:
00000000027900B0 3ABCB5B83A7273C0 3ABCB53B3A7173C0 ?sr:???:?sq:;??:
00000000027900C0 3ABCB53A3A2FCD38 3ABCB5E13ABDB531 8?/::??:1??:???:
00000000027900D0 3ABCB5023A725A12 3ABCB5303A765B57 .Zr:.??:W[v:0??:
00000000027900E0 3ABCB5303A705B57 3ABCB53168636952 W[p:0??:Rich1??:
00000000027900F0 0000000000000000 0005866400004550 ........PE..d...
0000000002790100 0000000603E4EF9 A02200F000000000 ?N>`........?."?
0000000002790110 0002AC00000B020B 0000000000020000 .....?..........
0000000002790120 000010000001C2CC 0000000180000000 ??..............
```

Accessing the PE Header

Reading the image DOS header, we can get the address of the PE header location from “e_lfanew”. Adding the offset to the address, we can confirm that we have the “PE” header using the display ascii command.

Command   Disassembly   Memory 0

```
00000000`027900f8  "PE"
0:065> dt -r ntdll!_IMAGE_DOS_HEADER 00000000`02790000
   +0x000 e_magic          : 0x5a4d
   +0x002 e_cblp           : 0x5241
   +0x004 e_cp             : 0x4855
   +0x006 e_crlc           : 0xe589
   +0x008 e_cparhdr        : 0x8148
   +0x00a e_minalloc       : 0x20ec
   +0x00c e_maxalloc       : 0
   +0x00e e_ss             : 0x4800
   +0x010 e_sp             : 0x1d8d
   +0x012 e_csum           : 0xffea
   +0x014 e_ip             : 0xffff
   +0x016 e_cs             : 0x8948
   +0x018 e_lfarlc         : 0x48df
   +0x01a e_ovno           : 0xc381
   +0x01c e_res            : [4] 0x5ff4
   +0x024 e_oemid          : 0xb5f0
   +0x026 e_oeminfo        : 0x56a2
   +0x028 e_res2           : [10] 0x468
   +0x03c e_lfanew         : 0n248
0:065> ? 00000000`02790000 + 0n248
Evaluate expression: 41484536 = 00000000`027900f8
0:065> da 00000000`027900f8
00000000`027900f8  "PE"
```

Parsing that address using the "_IMAGE_NT_HEADERS" shows the contents of the PE header.

Command | Memory 0

```
0:078> dt -r _IMAGE_NT_HEADERS 0x07a00f8
Windows_UI_Xaml!_IMAGE_NT_HEADERS
   +0x000 Signature          : 0x4550
   +0x004 FileHeader         : _IMAGE_FILE_HEADER
      +0x000 Machine            : 0x8664
      +0x002 NumberOfSections : 5
      +0x004 TimeDateStamp      : 0x603e4ef9
      +0x008 PointerToSymbolTable : 0
      +0x00c NumberOfSymbols    : 0
      +0x010 SizeOfOptionalHeader : 0xf0
      +0x012 Characteristics    : 0xa022
   +0x018 OptionalHeader     : _IMAGE_OPTIONAL_HEADER
      +0x000 Magic              : 0x20b
      +0x002 MajorLinkerVersion : 0xb ''
      +0x003 MinorLinkerVersion : 0 ''
      +0x004 SizeOfCode         : 0x2ac00
      +0x008 SizeOfInitializedData : 0x20000
      +0x00c SizeOfUninitializedData : 0
      +0x010 AddressOfEntryPoint : 0x1c2cc
      +0x014 BaseOfCode         : 0x1000
      +0x018 BaseOfData         : 0x80000000
      +0x01c ImageBase          : 1
      +0x020 SectionAlignment : 0x1000
      +0x024 FileAlignment      : 0x200
      +0x028 MajorOperatingSystemVersion : 5
      +0x02a MinorOperatingSystemVersion : 2
      +0x02c MajorImageVersion : 0
      +0x02e MinorImageVersion : 0
      +0x030 MajorSubsystemVersion : 5
      +0x032 MinorSubsystemVersion : 2
      +0x034 Win32VersionValue : 0
      +0x038 SizeOfImage        : 0x4e000
```

The final entry in the above image, “Size of Image”, gives us a rough size of the DLL. If we add the size of image to the base address of the DLL, we can see expected end address for the DLL. Referring back to the YARA string addresses, we can confirm that those strings are located within the memory region for our DLL.

```
0:078> ? 0x07a0000 + 0x4e000
Evaluate expression: 8314880 = 00000000`007ee000
WARNING: Teb 78 pointer is NULL - defaulting to 00000000`7ffde000
WARNING: 00000000`7ffde000 does not appear to be a TEB
0:078> s -a 0 L?80000000 "run command"
00000000`007cbba9  72 75 6e 20 63 6f 6d 6d-61 6e 64 20 28 77 2f 20  run command (w/
00000000`027bcba9  72 75 6e 20 63 6f 6d 6d-61 6e 64 20 28 77 2f 20  run command (w/
```

Now that we've confirmed that start and end address, we can export that memory region to get a copy of the CobaltStrike module.


```
0:078> ? 0x07a0000 + 0x4e000
Evaluate expression: 8314880 = 00000000`007ee000
WARNING: Teb 78 pointer is NULL - defaulting to 00000000`7ffde000
WARNING: 00000000`7ffde000 does not appear to be a TEB
0:078> s -a 0 L?80000000 "run command"
00000000`007cbba9  72 75 6e 20 63 6f 6d 6d-61 6e 64 20 28 77 2f 20  run command (w/
00000000`027bcba9  72 75 6e 20 63 6f 6d 6d-61 6e 64 20 28 77 2f 20  run command (w/
```


Note: It's worth noting that you won't get a perfect copy of the DLL. Usually the SizeOfImage is inaccurate and will result in garbage being written to the end of the file because we've written too much.

Parsing the Config

We can now use a config parser to extract the Cobalt Strike config from the DLL. This gives us further indicators of compromise to continue searching across the network for.

```
C:\Users\Dev\Documents\CobaltStrikeParser-master\CobaltStrikeParser-master>python parse_beacon_config.py C:\Exclusion\cs.dat
BeaconType                       - HTTP
Port                             - 80
SleepTime                        - 60000
MaxGetSize                       - 1048576
Jitter                           - 0
MaxDNS                           - Not Found
PublicKey_MD5                    - 5a1526e50f56fe04d4b64ef24ce5136b
C2Server                         - 192.168.56.102,/ca
UserAgent                        - Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0)
HttpPostUri                      - /submit.php
Malleable_C2_Instructions        - Empty
HttpGet_Metadata                 - Metadata
                                        base64
                                        header "Cookie"
HttpPost_Metadata                - ConstHeaders
                                        Content-Type: application/octet-stream
                                   SessionId
                                        parameter "id"
                                   Output
                                        print
```

From this, we can see that the default cobalt strike config is being used.

Wrap Up

- Windows Error Reporting is a super valuable artifact. I've used it to detect everything from DLL injection to lateral movement to credential dumping.
- If you know that an actors presence is purposely crashing a process, you can use that knowledge to gather more detailed process dumps.

Quick one this week, just getting back into blogging after a couple of conference talks. As always, hit me up on twitter with any questions.

Blake .