Analyzing .NET Core Single File Samples (DUCKTAIL Case Study)

Published: 2022-08-07 · Archived: 2026-04-05 22:42:00 UTC

This post is dedicated to my colleague Matt Graeber (@mattifestation) who showed me how to do the manual calculations and carving of PEs using CFF Explorer and a hex editor, making me think "there has to be a tool for this". There are loads of ways to deploy .NET Framework applications, and I've mostly been familiar with just the traditional compile-and-run method. As .NET malware has evolved, adversaries have looked to use different deployment reasons for the same reason as legitimate developers. In some cases it's easier, allows bundling files in different ways, or other reasons. One malware family named DUCKTAIL has recently embraced deployment in as a ".NET Single File" deployment feature. This feature is interesting in the sense that it allows a developer to deploy a .NET application to systems without requiring the systems to have a pre-deployed .NET Framework runtime installed. This dependency-free deployment is achieved by appending multiple binaries together into a single file, resulting in a large executable with multiple executables inside. In this post we'll take a look at a DUCKTAIL sample and how the "single file" deployment choice affects malware analysis. The sample we're working with is here in VT: https://www.virustotal.com/gui/file/740fd780b2b45c08d1abb45cddc6d1017c9fcc6bcce54fd8415d87a80d328ff6.

Triaging the file

As always, we can start out triaging the file using a combination of `diec` and `pehash`. First, let's figure out the file type:

```
remnux@remnux:~/cases/ducktail$ diec ducktail.exe
PE32
    Compiler: Microsoft Visual C/C++(-)[-]
    Linker: Microsoft Linker(14.29**)[GUI32,signed]
```

Right off the bat, we can see our output is a little different than what we'd expect for a standard .NET Framework app. In this case we see `diec` says the file was compiled using Visual C/C++ instead of a .NET language like C# or VB. Usually for .NET apps, the output would look similar to this:

```
PE32
    Library: .NET(v4.0.30319)[-]
    Compiler: VB.NET(-)[-]
    Linker: Microsoft Linker(8.0)[GUI32]
```

We know for sure we're dealing with a Windows Portable Executable (PE) file, so let's take a look at the import and rich header hashes. If you're looking to get the imphash for a sample, you can easily do so with `pehash`. For the rich header hash, I like using the tool I made here that leverages the Python pefile library.

```
remnux@remnux:~/cases/ducktail$ pehash ducktail.exe
file
    filepath:                       ducktail.exe
    md5:                            72840061e0f1b2f4bc373b5561970303
    sha1:                           c773a6285a54183e792f23e499646f61d9b2f88f
    sha256:                         740fd780b2b45c08d1abb45cddc6d1017c9fcc6bcce54fd8415d87a80d328ff6
    ssdeep:                         1572864:ha0XsmjPyZmtmuwKl7E2LmZBhbCV6ZE5GSQiOjEBkqYnIgJM0cAZv7SGdAcA689p:jjPyZxuwz+6y
    imphash:                        34dc34e244a6f4378a06076ff16fc082

remnux@remnux:~/cases/ducktail$ ./rhh-md5.py ducktail.exe
e0f1735adef0e9f084efeaee57b351d2
```

The imphash and rich header hash values are different that what I expect of traditional .NET malware. Usually, .NET executables have an import hash of `f34d5f2d4577ed6d9ceec516c1f5a744` and .NET DLLs have an import hash of `dae02f32a21e03ce65412f6e56942daa`. In addition, .NET executables and DLLs usually don't have rich header hashes. So this sample triages more like a native, unmanaged code binary than a .NET one.

Digging deeper with exiftool and pedump

We can get our first hints of the app being compiled as a .NET Core single file executable using `exiftool` and `pedump`.

```
remnux@remnux:~/cases/ducktail$ exiftool ducktail.exe
ExifTool Version Number         : 12.42
File Name                       : ducktail.exe
Directory                       : .
File Size                       : 56 MB
File Modification Date/Time     : 2022:08:07 21:14:35-04:00
File Access Date/Time           : 2022:08:07 21:15:11-04:00
File Inode Change Date/Time     : 2022:08:07 21:15:04-04:00
File Permissions                : -rw-rw-r--
File Type                       : Win32 EXE
File Type Extension             : exe
MIME Type                       : application/octet-stream
Machine Type                    : Intel 386 or later, and compatibles
Time Stamp                      : 2022:04:13 21:36:43-04:00
Image File Characteristics      : Executable, 32-bit
PE Type                         : PE32
Linker Version                  : 14.29
Code Size                       : 233984
Initialized Data Size           : 331776
Uninitialized Data Size         : 0
Entry Point                     : 0x2f8f0
OS Version                      : 6.0
Image Version                   : 0.0
Subsystem Version               : 6.0
Subsystem                       : Windows GUI
File Version Number             : 1.0.0.0
Product Version Number          : 1.0.0.0
File Flags Mask                 : 0x003f
File Flags                      : (none)
File OS                         : Win32
Object File Type                : Executable application
File Subtype                    : 0
Language Code                   : Neutral
Character Set                   : Unicode
Company Name                    : DataExtractor
File Description                : DataExtractor
File Version                    : 1.0.0.0
Internal Name                   : DataExtractor.dll
Legal Copyright                 :
Original File Name              : DataExtractor.dll
Product Name                    : DataExtractor
Product Version                 : 1.0.0
Assembly Version                : 1.0.0.0
```

The file size is 56 MB. It's not the beefiest binary ever, but it's still pretty heavy and that can indicate multiple binaries in a single file. From here we can look at binary properties with `pedump` to get some more data. In the interest of brevity I've cut down the `pedump` output to just the exports since it contains the relevant bits.

```
remnux@remnux:~/cases/ducktail$ pedump --exports ducktail.exe

=== EXPORTS ===

```

```
# module "singlefilehost.exe"
# flags=0x0  ts="2106-02-07 06:28:15"  version=0.0  ord_base=1
# nFuncs=23  nNames=23

  ORD ENTRY_VA  NAME
    1    21580  corehost_initialize
    2    20ad0  corehost_load
    3    20f20  corehost_main
    4    21090  corehost_main_with_output_buffer
    5    21960  corehost_resolve_component_dependencies
    6     eba0  corehost_set_error_writer
    7    218e0  corehost_unload
    8     f260  hostfxr_close
    9     e8d0  hostfxr_get_available_sdks
    a     eaa0  hostfxr_get_native_search_directories
    b     ef80  hostfxr_get_runtime_delegate
    c     f190  hostfxr_get_runtime_properties
    d     efd0  hostfxr_get_runtime_property_value
    e     ed40  hostfxr_initialize_for_dotnet_command_line
    f     ee70  hostfxr_initialize_for_runtime_config
   10     e520  hostfxr_main
   11     e3e0  hostfxr_main_bundle_startupinfo
   12     e490  hostfxr_main_startupinfo
   13     e5c0  hostfxr_resolve_sdk
   14     e720  hostfxr_resolve_sdk2
   15     ef10  hostfxr_run_app
   16     eba0  hostfxr_set_error_writer
   17     f130  hostfxr_set_runtime_property_value
```

.NET core “single file” apps are multiple binaries appended to one another, right? Well, the first binary in the append chain has the responsibility of being a “.NET loader” that loads subsequent .NET resources (appended after the loader) into memory at runtime. Once the resources get loaded, the actual .NET app gets run. The export details seen here in `pedump` are from the .NET loader overhead itself, which results in some good predictability. The .NET core “single file” apps should usually have exports like `corehost_initialize`, `corehost_load`, and others.

Getting the actual app/malware

We’ve got our bearings a bit and we know from documentation that a “single file” app is just a bunch of binaries appended to each other. So, logically, we should be able to walk the file and extract all the PEs from it. We can do this using `pecheck.py`.

```
remnux@remnux:~/cases/ducktail$ pecheck -l P ducktail.exe
1: 0x00000000 EXE 32-bit 0x0350a17f 72840061e0f1b2f4bc373b5561970303 0x0350a17f (EOF) b'' b'singlefilehost.exe'
2: 0x0008a71c DLL 64-bit 0x0013e6cb d35f8c57d217a41dfc5e68bf25e5ecb1 0x0350a17f (EOF) b'empty' b'clrcompression.dll'
3: 0x0013e6cc DLL 64-bit 0x0024687b e127d23181160e02391e628192b1d08a 0x0350a17f (EOF) b'clrjit.dll' b'clrjit.dll'
4: 0x0024687c DLL 64-bit 0x00652623 99004b84b758edc90f90671221152667 0x0350a17f (EOF) b'CoreCLR.dll' b'coreclr.dll'
5: 0x00652624 DLL 64-bit 0x007443c3 ea613da6eeb3f2968faa2d65dabadab1 0x0350a17f (EOF) b'mscordaccore.dll' b'mscordac.dll'
6: 0x007443c4 DLL 32-bit 0x008655c3 e02613d1a6211eb1bfc8d15431acbd68 0x0350a17f (EOF) b'' b'e_sqlite3.dll'

...

24: 0x0087fbd0 DLL 32-bit 0x010aed77 d3cfe3422fb4d5a93c1cf9807debd230 0x0350a17f (EOF) b'' b'System.Private.CoreLib.dll'
25: 0x010aed80 DLL 32-bit 0x0111d57f 4ef7d9040e94a8c3a9ede74a8f66a73f 0x0350a17f (EOF) b'' b'Dapper.dll'
26: 0x0111d580 DLL 32-bit 0x0117b77f a660b3d199853c0b014812f39e46eaa8 0x0350a17f (EOF) b'' b'HtmlAgilityPack.dll'
27: 0x0117b780 DLL 32-bit 0x011ce97f 2904b6192503177cf287f6ae23ed65d5 0x0350a17f (EOF) b'' b'Microsoft.Data.Sqlite.dll'
28: 0x011ce980 DLL 32-bit 0x0135e57f 47d413a62176af3f801b9f6a1146e1a7 0x0350a17f (EOF) b'' b'Newtonsoft.Json.dll'
29: 0x0135e580 DLL 32-bit 0x019a2f7f 6697ec4f0f13bed443f3b070cc4192df 0x0350a17f (EOF) b'' b'BouncyCastle.Crypto.dll'
30: 0x019a2f80 DLL 32-bit 0x019a4b7f 9b59e64ef76c1a543983b8dcb1ce8d75 0x0350a17f (EOF) b'' b'SQLitePCLRaw.batteries_v2.dll'
31: 0x019a4b80 DLL 32-bit 0x019a637f 8b477db107c8ac8c219d90d94d93aaa4 0x0350a17f (EOF) b'' b'SQLitePCLRaw.nativelibrary.dll
32: 0x019a6380 DLL 32-bit 0x019ba57f 5bacb4c47e3ba56dd53cf88781bb4e05 0x0350a17f (EOF) b'' b'SQLitePCLRaw.core.dll'
33: 0x019ba580 DLL 32-bit 0x019d077f 7a9ca8439b58afd87f4faec21968c087 0x0350a17f (EOF) b'' b'SQLitePCLRaw.provider.dynamic_
34: 0x019d0780 DLL 32-bit 0x019d837f 5b015246ff6883063438c8ecf4af101e 0x0350a17f (EOF) b'' b'System.Security.Cryptography.P
35: 0x019d8380 DLL 32-bit 0x01a3417f ed5bdc648cba3d82edd0b14bed18b931 0x0350a17f (EOF) b'' b'Telegram.Bot.dll'
36: 0x01a34180 DLL 32-bit 0x01aa217f 6a62b196160d1a477effa8e07ae48533 0x0350a17f (EOF) b'' b'DataExtractor.dll'
```

```
37: 0x01aa2180 DLL 32-bit 0x01b7bb7f 0b360b2e48ad740b2045c96c228d8dfa 0x0350a17f (EOF) b'' b'Microsoft.CSharp.dll'

...

92: 0x03459580 DLL 32-bit 0x034c917f 6d306c25b62c2422a8411315307f5bf5 0x0350a17f (EOF) b'' b'System.Text.RegularExpressions
93: 0x034c9180 DLL 32-bit 0x034e0b7f 21fef48538579c3d2533532c4b143e75 0x0350a17f (EOF) b'' b'System.Threading.Channels.dll'
94: 0x034e0b80 DLL 32-bit 0x034f037f e58c38c4e4bfc5151c0f1ff350bfe6b7 0x0350a17f (EOF) b'' b'System.Threading.dll'
```

Using `pecheck` with arguments to list the available PEs in the file reveals a whopping 94 different PE files within the single original `ducktail.exe` sample. Thankfully, there are only a couple of PEs here that are interesting to us: 35/Telegram.Bot.dll and 36/DataExtractor.dll. We can extract those with `pecheck` as well!

```
remnux@remnux:~/cases/ducktail$ pecheck -l 35 -g s -D ducktail.exe > Telegram.Bot.dll
remnux@remnux:~/cases/ducktail$ diec Telegram.Bot.dll
PE32
    Library: .NET(v4.0.30319)[-]

remnux@remnux:~/cases/ducktail$ pecheck -l 36 -g s -D ducktail.exe > DataExtractor.dll
remnux@remnux:~/cases/ducktail$ diec DataExtractor.dll
PE32
    Library: .NET(v4.0.30319)[-]
```

Excellent, we've successfully extracted the actual DUCKTAIL .NET code from its "single file" container!

Decompilation and further steps

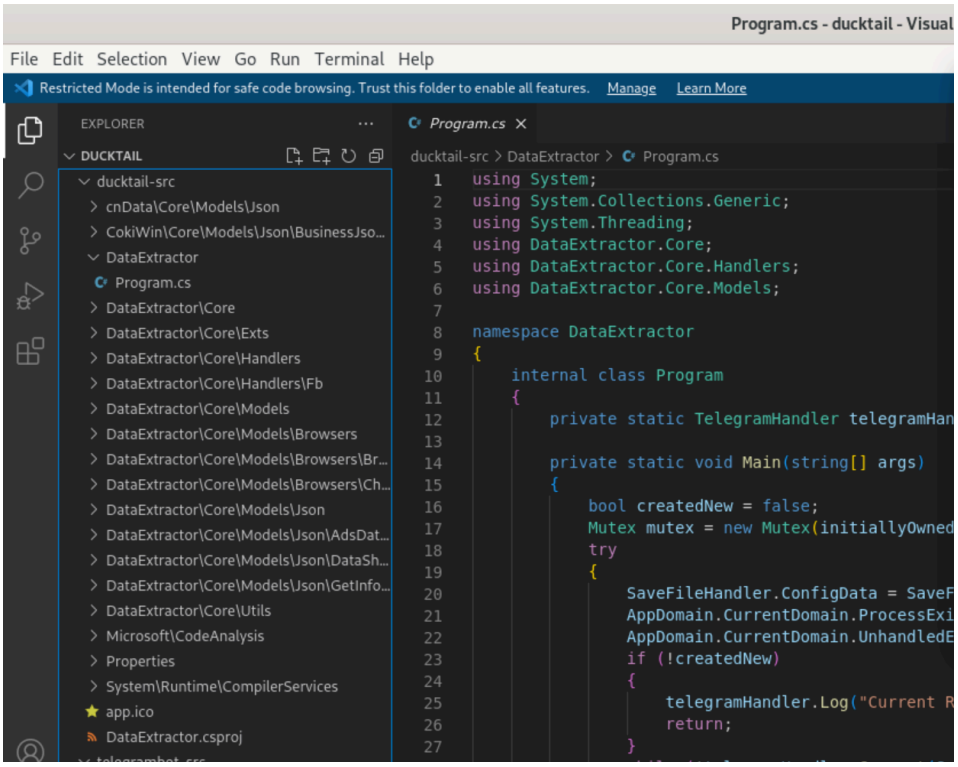
Decompiling is a breeze with this sample thanks to `ilspycmd`. Lately I've been using it with command line arguments to export code as a .NET project so I can get extra details in there like the icon used by the malware.

```
remnux@remnux:~/cases/ducktail$ mkdir ducktail-src
remnux@remnux:~/cases/ducktail$ ilspycmd -p -o ./ducktail-src/ DataExtractor.dll

remnux@remnux:~/cases/ducktail$ mkdir telegrambot-src
remnux@remnux:~/cases/ducktail$ ilspycmd -p -o ./telegrambot-src/ Telegram.Bot.dll

remnux@remnux:~/cases/ducktail$ tree -a ducktail-src/
ducktail-src/
├── app.ico
├── cnData\Core\Models\Json
│   └── DataJsonModel.cs
├── CokiWin\Core\Models\Json\BusinessJsonModel
│   ├── Adaccount_Permissions.cs
│   ├── BusinessJsonModel.cs
│   ├── Clients.cs
│   ├── Cursors1.cs
│   ├── Cursors.cs
│   ├── Datum1.cs
│   ├── Datum.cs
│   ├── Paging1.cs
│   └── Paging.cs
├── DataExtractor
│   └── Program.cs
...
```

From here, if you want to get into deeper analysis you can start with the `Program.cs` file and simply follow the flow of code to other files as relevant!



Source: https://forensicitguy.github.io/analyzing-net-core-single-file-ducktail/