An AI Based Solution to Detecting the DoubleZero .NET Wiper

By Akshata Rao, Zong-Yu Wu, Wenjun Hu

Published: 2022-11-19 · Archived: 2026-04-05 15:16:18 UTC

Executive Summary

Unit 42 researchers introduce a machine learning model that predicts the maliciousness of .NET samples based on specific structures in the file, by analyzing a .NET wiper named DoubleZero. We identify the challenges of detecting this threat through PE structural analysis and conclude by examining the cues picked up by the machine learning model to detect this sample.

While wipers are not necessarily new, the recent discovery of several new wipers associated with the ongoing Russia-Ukraine cyber activity has driven renewed public interest in understanding how to identify and defend against their use.

Palo Alto Networks customers receive protections from .NET malwares with Cortex XDR or the Next-Generation Firewall with cloud-delivered security services, including WildFire and Advanced Threat Prevention.

DoubleZero Wiper

The wiper DoubleZero was revealed to the public by Ukraine CERT in March 2022. It is one of many wipers – including HermeticWiper, IsaacWiper and CaddyWiper – that were apparently targeted against their country.

DoubleZero is implemented in C#, and it is heavily obfuscated. This wiper is capable of typical, non-reversible and destructive actions that include the following:

- Gaining the highest privilege in the host
- Hunting down all available targeted files from every volume
- Destroying targeted files or disk volumes within a short period of time

Unlike HermeticWiper and other wipers that damage the Master Boot Record (MBR) or GUID Partition Table (GPT), DoubleZero only wipes the first 4096 bytes in selected files using the file system driver. We limit our focus on the .NET samples feature engineering, but this Splunk article provides a more in-depth technical analysis of the wiper sample.

SHA256	3b2e708eaa4744c76a633391cf2c983f4a098b46436525619e5ea44e105355fe
File size	419.50 KB

Examining the PE File Structure

Researchers have been commonly using Microsoft Windows Portable Executable (PE) file structure to detect PE malware for many years. The thought behind this tactic is that variants of the same malware family can often share similarities in their file structure.

Machine learning models also detect PE malware based on the file structure. To do so, they have historically used PE header characteristics, imports and section attributes to recognize known malware characteristics.

While the .NET PE file structure is based on the PE format, a lot of its functionality is encoded within the .NET specific structures. In the next section, we'll dive deeper into the .NET structural attributes.

An Inside Look Into .NET files

The .NET open source framework was introduced in 2002 and is available by default in most Microsoft Windows platforms. The framework provides developers with powerful, built-in methods to access the internet, file system and encryption. Having these options already available makes this framework attractive to both system administrators and malware authors.

Microsoft Windows PE loader will already know how to load and execute the .NET sample once the framework is ready, since .NET assembly re-uses the PE file format.

During code generation, the source code for .NET compatible languages is compiled to platform-agnostic Common Intermediate Language (CIL) byte code. A special .NET IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR can be found in the data directory for referencing the Cor20 header, as shown in Figure 1. The Cor20 header (otherwise known as the CLR header) contains .NET specific information like versioning and resource information and the location of the Metadata header. The resources, streams, type references, methods, CIL byte codes and usage of external assemblies are all in the metadata. For a more detailed description, please refer to our article on dotnetfile.

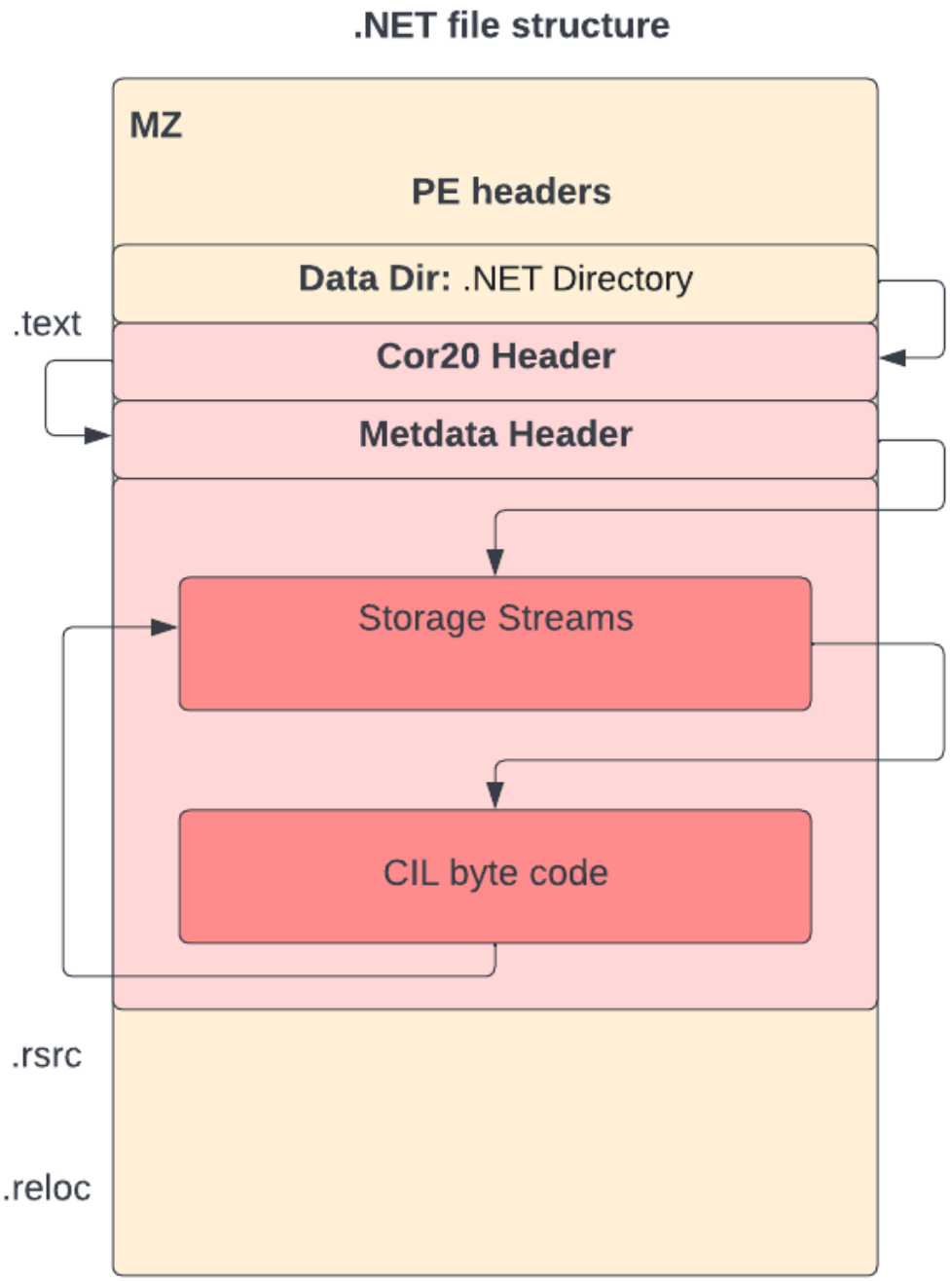

Figure 1. A simple illustration of .NET file structure.

Unlike most PE files, where dependent libraries and APIs can be found in the import table, for a .NET file the import table contains just one module (the mscoree.dll) and one imported function (_CorExeMain for executables and _CorDllMain for dynamic load libraries). The resources are contained within the .NET storage stream instead of the .rsrc section.

The entry point of the PE header points to the _CorExeMain or _CorDllMain of the mscoree.dll, while the actual entry point can be found in the Cor20 header as shown in Figure 2. Thus, some of the most useful attributes to identify PE malware files do not contain many clues for .NET malware.

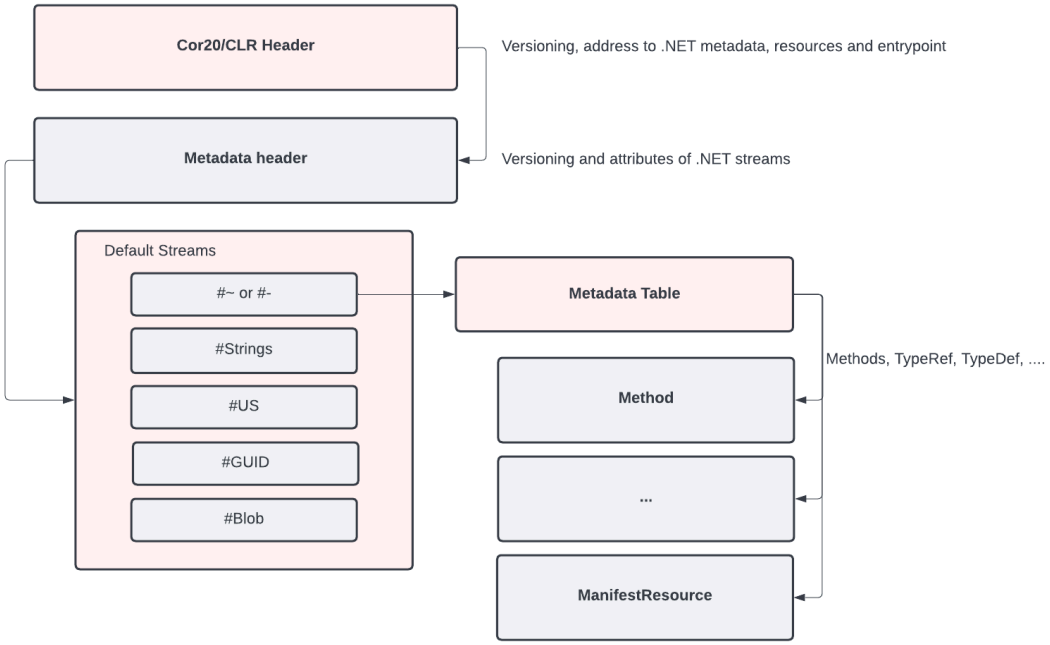

Figure 2. .NET specific structures and the data they hold.

The metadata table contains defined tables for Module, MethodDef and Assembly among others. By examining the .NET file structure, we can see it provides a lot more information about the file as compared to the PE file structure alone. Instead of only using the general PE features for the .NET machine learning solution, we added the data specifically parsed from the structure of .NET samples. These customized features improve the quality of our machine learning solutions.

Applying Machine Learning to .NET Malware Detection

We have trained a machine learning model on custom features learned on the PE file structure, .NET file structure and other file characteristics of .NET samples, as shown in Figure 3.

As .NET is an extremely popular file type in customer environments, it is necessary for us to have a quick turnaround time in case any disruptive false positives emerge. This means that our model's functionality has to be well understood by researchers, and that the predictions could be reverse engineered.

Because .NET files are so ubiquitous, they also have a highly imbalanced distribution of benign vs. malicious samples. Consequently, the trained model has to ensure that we maintain extremely low false positive rates while retaining detection accuracy.

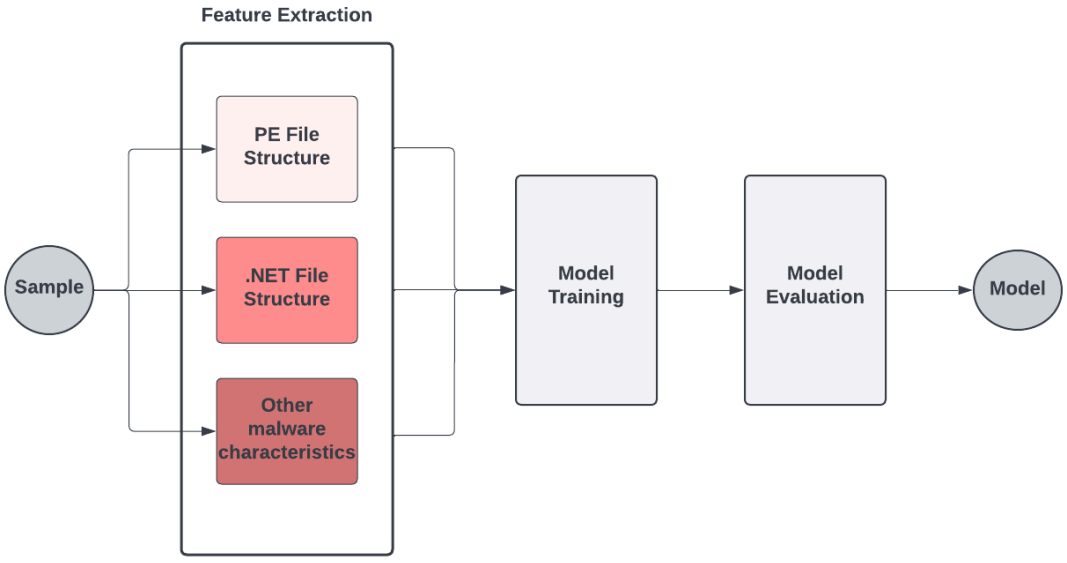

Figure 3. An overview of our detection workflow.

Detecting the DoubleZero Wiper

The malware sample has a single import, _CorExeMain, from mscoree.dll, consistent with other .NET executable files. It contains only .text and .reloc sections, all of which are native to the PE file format and have fairly low entropy. Hence, there's not much we can learn from this file's PE structure.

The CIL bytecode uses obfuscation to hide the string resources that are critical for pattern based detection. It also interleaves random code among actual operations, together with flattened control flow, to make the code harder to follow. However, we can glean a lot about the objective of this malware by observing the imported libraries and associated API calls found in the decompiled code.

The imports System.DirectoryServices.ActiveDirectory and System.Security.AccessControl indicate the intention to interact with the Active Directory and access control attributes of the files.

The regular expression function imported by System.Text.RegularExpressions is used to locate certain files by patterns.

The most significant signs of malicious code usage are the calls to the unmanaged DLL functions. Platform Invoke (P/Invoke) enables the managed codes in .NET to call the unmanaged functions. Calling the lower level APIs directly is a well known anti-analysis technique to evade API hooking by sandbox. The following unmanaged APIs are used in the sample:

- ntdll.ntopenfile
- ntdll.ntfscontrolfile
- ntdll.rtlntstatustodoserror
- ntdll.rtladjustprivilege
- kernel32.closehandle
- kernel32.getfilesizeex

- kernel32.getlasterror
- user32.exitwindowsex

In most of the use cases, there are equivalent managed functions in .NET runtime libraries to use. However, this sample intends to operate at a lower level with some of the unmanaged APIs in ntdll.dll**.**

The target files and the regular expression patterns are surprisingly stored as plain strings, as shown in Figure 4. This possible mistake by the threat actor allows us to get more insights about the objective of the sample.

```
new Regex(GClass4.smethod_0() + "\\Users\\\\.*?\\\\Local Settings.*", RegexOptions.IgnoreCa
new Regex(GClass4.smethod_0() + "\\Users\\\\.*?\\\\AppData\\\\Local\\\\Application Data.*",
new Regex(GClass4.smethod_0() + "\\Users\\\\.*?\\\\Start Menu.*", RegexOptions.IgnoreCase |
new Regex(GClass4.smethod_0() + "\\Users\\\\.*?\\\\Application Data.*", RegexOptions.Ignore
new Regex(GClass4.smethod_0() + "\\ProgramData\\\\Microsoft.*", RegexOptions.IgnoreCase | R
```

Figure 4. An example decompiled code snippet where the sample uses regular expressions to search the target files.

Though obfuscation exists, we are still able to get the argument FsControlCode for the call to ntdll.NtFsControlFile, which is 0x980c8 = FSCTL_SET_ZERO_DATA. This is how the wiper writes 4K of null bytes to a targeted file that was opened by ntdll.NtOpenFile.

The sample calls ntdll.RtlAdjustPrivilege several times with privileges 9, 17, 18 and 19. These privileges are SeTakeOwnershipPrivilege, SeBackupPrivilege, SeRestorePrivilege and SeShutdownPrivilege respectively. These privileges are used to ensure the wiper has the access right to destroy the target file and reboot the system after all the actions are done.

Unlike HermeticWiper and ransomware like LockBit, DoubleZero doesn't delete the shadow copies that can possibly be used to recover files from the damage. However, the wiper does kill any process named lsass. Lsass is important to the Windows system, and any application that kills this process is most likely malicious.

The aforementioned file characteristics give us clues that the machine learning model uses to detect the sample. Though obfuscation is more common among malicious .NET files, because the sample uses several unmanaged APIs as well as sensitive target path patterns, these stand out as potentially malicious indicators.

Conclusion

Researchers have been using PE file structure to detect variants within malware families for years, as they often share similarities in their file structure. Despite the .NET framework being well known, it can still be challenging to detect files using this strategy. Some of the most useful attributes for identifying PE malware files don't offer many clues for identifying .NET malware.

We have highlighted the fundamental difference between .NET and PE samples to help identify where those clues could be hiding. In doing so, we illustrated that machine learning feature engineering can be improved by correctly parsing out the .NET specific data structure.

We also showed that imported libraries, unmanaged API calls and unencrypted strings are the most useful features for detecting .NET malware. While a recent wiper called DoubleZero was analyzed as an example, the features

that are important for detecting .NET can be generalized, and this feature can be broadly useful for future research efforts.

Palo Alto Networks customers receive protections from .NET malware with Cortex XDR or the Next-Generation Firewall with cloud-delivered security services including WildFire and Advanced Threat Prevention.

Indicators of Compromise

3b2e708eaa4744c76a633391cf2c983f4a098b46436525619e5ea44e105355fe

---