EXTRACTION OF INFORMATION FROM .NET EXECUTABLE FILES

Marek Milkovič

Abstract

The occurrence of malicious software written in .NET languages is rapidly increasing. Extracting the information out of .NET executable file is therefore necessary step in order to fight against this kind of malware. In this paper, we propose methods to extract information out of .NET executable files, which are safe and platform independent. These methods include data type reconstruction, but also extraction of unique features such as *TypeLib* identifier or *Module Version* identifier. We also point out mistakes that are being done in this field so far. After implementing the proposed methods, we compare it with already existing .NET disassemblers yielding very good results. Extracted information are planned to be used in creation of detection patterns, clustering and other areas at Avast Software.

Keywords: *reverse engineering, executable files, .NET, type reconstruction, typelib*

1 INTRODUCTION

According to studies, there was an increase of .NET malicious software (malware) from 2009 to 2015 by 1600% [8]. This gives us a motivation to reverse engineer and study .NET executable files so we can understand this kind of malware because if we understand it, we can create detection patterns that can match families of malware. We can cluster these executable files based on their features. We can train neural networks with clean and malware samples so it can help us triage what is malware and what is not. For all of this, we need to decide what in .NET executable file is important, prevalent or unique enough that would make make it all work. We need to find right set of features that would give us insight on how is this executable file different from other malware strains and what it has common with the samples in the same strain.

Despite that .NET executable files are stored in the same format as native Windows executable files, we cannot use the same information as with native ones because this native format is just used as an envelope that is almost the same across all .NET executable files. This paper describes a way how to extract information from .NET executables without relying on .NET framework because this might represent the potential risk. We also want our solution to be dependency free in order to work on all commonly used platform nowadays. This paper focuses on extraction of information that is important and unique enough such that it can be used to create detection pattern or cluster the samples. The features we focus on are class names, their methods, fields, properties *TypeLib* identifier and *Module Version* identifier.

2 EXECUTABLE FILE FORMAT

There are many file formats for executable files but we will be dealing with file format called PE (Portable Executable) [6] because it is used as a container for storing .NET executable files. Therefore, we need to understand some aspects of this format at first.

PE file format was introduced by Microsoft as successor to the old MZ format used on DOS. You may sometimes see it called also PE/COFF because it was derived from COFF format. It is still used to this day on modern Windows systems not only for executable files (EXE), but also dynamic-link libraries (DLL) or multilangual interface files (MUI). Figure 1 shows the basic layout of PE file.

At the end of the PE header, there are structures containing information about data directories, which are just parts of PE file with special semantics like imported symbols, exported symbols, debugging information or CLR (Common Language Runtime) header. The latter is used for storing the .NET metadata and serves for finding all the necessary information about the .NET executable encapsulated in PE file.

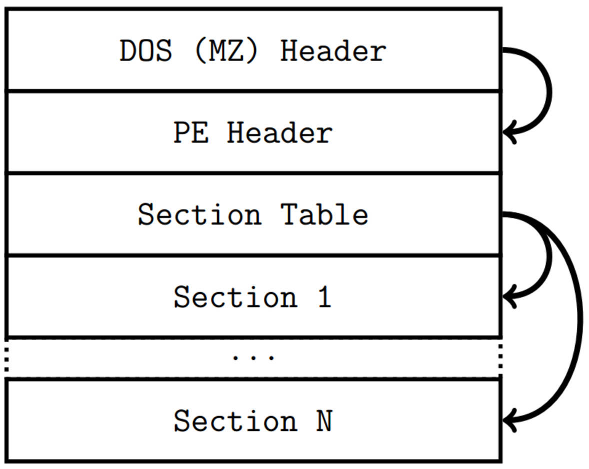

Figure 1: Structure of executable file in PE format.

CLR header points to metadata header which is no longer part of PE specification. It stores addresses of so-called *streams*. Each *stream* represents some specific type of data. Available streams are:

- *Metadata stream* - Information about classes, fields, methods and many more. Identified by name **#~** or **#-**. The latter one is not documented identifier for metadata stream but unofficial sources describe it as unoptimized stream [5]. Unoptimized streams use undocumented indirect tables which will be described latter in this chapter.
- *GUID stream* - Table of globally unique identifiers. Identified by name **#GUID**.
- *Blob stream* - Binary data such as encoded signatures of methods, types of attributes and many more. Identified by name **#Blob**.
- *String stream* - Strings such as name of classes, fields, methods and many more. Identified by name **#Strings**.
- *User string stream* - User defined strings without specific semantics. Identified by name **#US**.

Figure 2 shows the hierarchy of structures storing .NET information.

There are multiple tables in metadata stream but not all of them need to be present at the same time. Which tables are actually used is determined based on 64-bit word **Valid** written in metadata stream header. Each bit of this word represents some specific table and if it is set then also the associated table is present. Metadata stream header is followed by sizes of each table. As last come the tables that are stored sequentially.

Tables are made of records which contain attributes. Attributes in the table record can point to different streams when referencing name of the item e.g. name of the class references *string stream*. This index is 2-bytes in size if the length of *string stream* is less than 2^{16}, otherwise it is 4-bytes in size. The same applies also to indices pointing to other streams. Apart from references to other streams, attributes can also reference other tables. If the attribute points just to one specific table, then it works the same way as with the pointers to the streams. Table pointer can however be dynamic and can point to multiple tables (just one simultaneously). If the field can point to n tables, then we need $\lceil \log_2(n) \rceil$ bits to determine which table to use. The maximum size of the table for compact (2-byte) index size is then shrank down to $2^{16 - \lceil \log_2(n) \rceil}$.

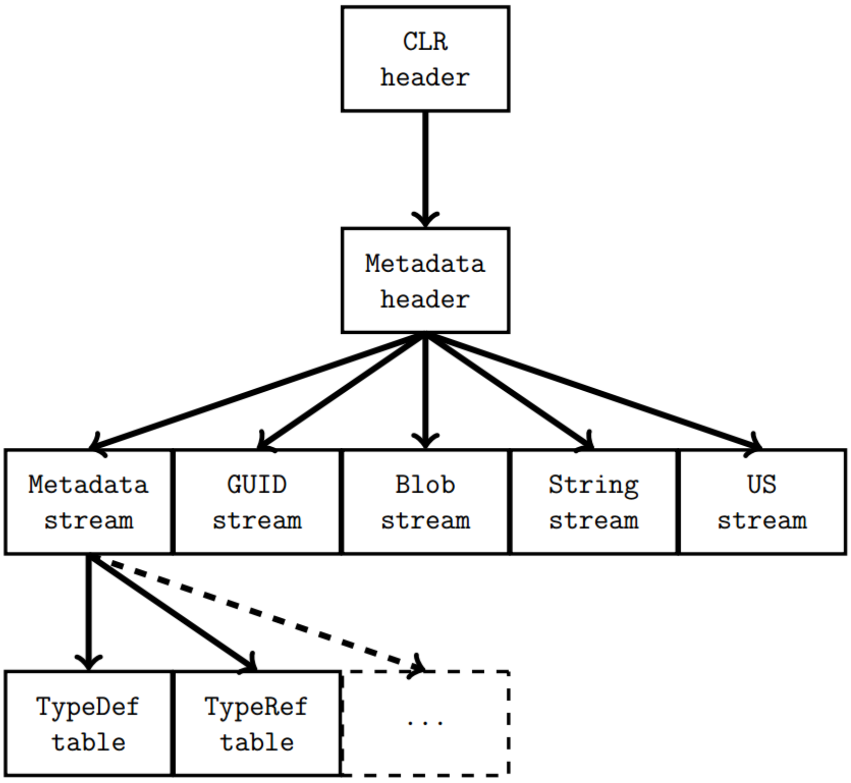

Figure 2: Hierarchy of structure in .NET executable file.

The *almost* complete list of all tables can be found in the standard ECMA-335 [2]. The almost means that there are some tables which are not even mentioned in the standard, but still can be found in the executable so be aware of these:

- Table #3 - **FieldPtr**
- Table #5 - **MethodPtr**
- Table #7 - **ParamPtr**
- Table #22 - **PropertyPtr**

These are so called indirect tables [5]. They are used in unoptimized streams. Methods, fields, parameters or properties which belong to the same parent type are kept together in their tables when stored in optimized stream. For unoptimized streams, rather than referencing child table directly, indirect table is used. Records in indirect tables are kept together as in direct table and they point to some item in their direct counterpart. That allows interleaving of items in direct tables.

2.1 TypeLib identifier

TypeLib identifier is a generic unique identifier used when interacting with your .NET library as COM (Component Object Model) interface. It is usually generated automatically when creating new .NET project in Visual Studio but can be also generated manually when Visual Studio does not provide you one. However, there may also be .NET modules without *TypeLib* identifier if author did not choose to create one or found its bytes in the file and replaced them with bogus values. It remains the same across different builds and consists of 16 bytes, so there is very little probability that two .NET libraries has the same *TypeLib* identifier. Therefore, it is a good source of information when classifying malware. It can be found in *blob stream* and is usually written down in form described by this regular expression:

[a-z0-9]{8}-([a-z0-9]{4}){3}-[a-z0-9]{12}

2.2 Module Version identifier

Module version ID (or just *MVID*) is always automatically generated when building the project and it uniquely identifies the build of .NET module. Same as with *TypeLib* identifier, it consists of 16 bytes. In contrary to *TypeLib* identifier, it is always present in every single .NET module. Since it changes with every new build it does not provide very broad detection, but still can be used to identify matching modules where each may be packed with different kind of packer or modified with some post-build scripts. This identifier is the easiest to change because it is usually placed at the beginning of *GUID stream*. That is also the reason we sometimes see executable files with *MVID* consisting of all zeroes in the wild.

3 RELATED WORK

The official documentation for CLI (Common Language Infrastructure) [2] describes the concepts for transforming high-level constructs into low-level representation in CIL (Common Intermediate Language). It explains what streams are and what metadata tables should be used to store which kind of information. However, it does not mention anything about unoptimized streams or indirect tables. You can read about these in the book *.NET IL Assembler* [4] which provides more detailed explanation of each metadata table, contains many information about undocumented features and is currently the best source of information regarding CLI. Both of these however do not deal with type reconstruction.

From the analytical software point of view, there are currently many .NET disassemblers which can already reconstruct the data types however all of them have a drawback. Some of them cannot handle corrupted, packed or obfuscated files. Others are written in some other .NET language (most likely C#) making it very hard to incorporate into other C/C++ libraries. This approach can also be harmful if not done properly. Loading .NET module into memory needs to be done in reflection-only mode otherwise malicious code can be executed on load. The last problem we often face are undocumented features of CLI which are not supported.

The most notable example of a library for parsing .NET executables is **dnlib** [1] which is the most robust from all the other libraries. However it is written in C# and thus relies on .NET framework. Another example is **ildasm** [3], IL disassembler from Microsoft which comes with every installation of Visual Studio. It is both GUI and console application. The problem is that it is not very robust and tends to crash with packed or obfuscated binaries. Its output is also not suitable for automated processing, because it dumps the whole disassembled code. Similar to **ildasm** is **ikdasm** [4] which is just its clone that does not run on *System.Reflection* module but rather on its reimplementation *Managed.Reflection*. The newer alternative to **ikdasm** is **monodis** [7], which rely on Mono framework and has many bugs of its predecessor fixed, but still not suitable for automated processing.

The first one to introduce *TypeLib* and *Module Version* identifier for .NET malware classification was Brian Wallace [12]. He also suggested the ways to extract it either with using .NET or just simply searching for strings with regular expression. The first approach is not suitable for environments where .NET cannot be used and the latter is not reliable because .NET binaries may contains many GUIDs. He also provided Python script for extraction of *TypeLib* ID but this script does not know how to deal with some packed or obfuscated binaries. This script was also incorporated into VirusTotal [10] resulting in inability to extract *TypeLib* ID from certain executables. Project YARA [11], tool for searching detection patterns written in declarative language called YARA, also supports parsing of *TypeLib* identifier but it does not support unoptimized streams and also *Module Version* identifier.

4 INFORMATION EXTRACTION

In this chapter, we will describe how several selected information can be extracted from .NET executable files. There are still many more information to extract than this chapter describes but our focus is brought on data type reconstruction and detection of *TypeLib* identifier.

4.1 User-defined classes

For parsing user-defined classes we will use table **TypeDef**. Each record in this table contains index into *string stream* with the name of the class and its namespace. In case of generic class, the name of the class contains special suffix `*N* (backquote followed by number), where *N* is the number of generic parameters. Additional information such as the accessibility of a class, seal, abstractness or interface are determined based on **Flags**.

Structure of table **TypeDef**.

```
struct TypeDef {
   uint32 Flags
   Index<StringStream> Name
   Index<StringStream> Namespace
   Index<TypeDef, TypeRef> Extends
   Index<Field> FieldList
   Index<MethodDef> MethodList
}
```

4.2 External classes

External classes are imported from other modules. We can use this information to get slight idea on what this binary does, but it is also important for proper reconstruction of user-defined types. Table **TypeRef** is used for these kinds of types. It contains name and namespace such as user-defined types, but nothing more.

Structure of table **TypeRef**.

```
struct TypeRef {
   Index<Module, ModuleRef, AssemblyRef, TypeRef> ResolutionScope
   Index<StringStream> Name
   Index<StringStream> Namespace
}
```

4.3 Class fields

Records in the table **TypeDef** contain field **FieldList**, which is index to **Field** table (**FieldPtr** for unoptimized streams). The number of fields is determined from the indices of two consecutive **TypeDef** records. For the last record, the number of fields is determined as difference between the size of **Field** table (**FieldPtr** for unoptimized streams) and the index of the last **FieldList** value.

Each field record represents one field in class. It contains index into *string stream* with its name. The data type of field is determined from the encoded signature that is located in *blob stream*. Visibility and more additional information is determined from **Flags**.

Structure of table **Field**.

```
struct Field {
   uint16 Flags
   Index<StringStream> Name
   Index<BlobStream> Signature
}
```

4.4 Class properties

Reconstruction of properties is not as straightforward as fields. The table **Property** contains all properties in user-defined types. Association of records in **TypeDef** table and **Property** table is then done through table **PropertyMap**. Everything else is the same as with fields. Each record in **PropertyMap** contains field **PropertyList**, which points to the first property of associated class in the table **Property** (**PropertyPtr** for unoptimized streams). The number of properties is determined from two consecutive **PropertyMap** records.

Structure of tables **Property** and **PropertyMap**.

```
struct Property {
   uint16 Flags
   Index<StringStream> Name
   Index<BlobStream> Signature
}

struct PropertyMap {
   Index<TypeDef> Parent
   Index<Property> PropertyList
}
```

4.5 Methods

The table **MethodDef** is used for reconstruction of class methods. The number of methods is calculated the same way as with fields and properties. The number and types of formal parameters needs to be extracted from the encoded signature present in the *blob stream* which is indexed with attribute **Signature**. In case of generic method, the first byte of signature has sixth bit (starting from 1) set to 1. It is followed by encoded number of parameters which are further followed by encoded types of each individual parameter. Identifiers of parameters are reconstructed from the table **Param** which the attribute **ParamList** points to (**ParamPtr** for unoptimized streams).

Structure of table **MethodDef**.

```
struct MethodDef {
   uint32 RVA
   uint16 ImplFlags
   uint16 Flags
   Index<StringStream> Name
   Index<BlobStream> Signature
   Index<Param> ParamList
}
```

4.6 Inheritance

Inheritance is determined from **TypeDef** table. Records there contain **Extends**, which is reference to one of tables **TypeDef**, **TypeRef** or **TypeSpec**. The latter is used in case of inheritance from generic class specialization such as **Class<int>**.

There is no support for multiple inheritance, but single class can implement several interfaces. These are not handled through **Extends** but rather table **InterfaceImpl**, which contains just two fields for each record and that is reference to **TypeDef** as class implementing the interface and the reference to either **TypeDef**, **TypeRef** or **TypeSpec** as interface being implemented.

Structure of tables **InterfaceImpl** and **TypeSpec**.

```
struct InterfaceImpl {
   Index<TypeDef> Class
   Index<TypeDef, TypeRef> Interface
}

struct TypeSpec {
   Index<BlobStream> Signature
}
```

4.7 Generic parameters

It is appropriate to reconstruct also names of generic parameters for which the table **GenericParam** is used. Attribute **Owner** references either **TypeDef** or **MethodDef** depending on whether the generic parameters belongs to class or method. The name is then references with attribute **Name** that points to *string stream*.

Structure of table **TypeRef**.

```
struct GenericParam {
   uint16 Number
   uint16 Flags
   Index<TypeDef, TypeRef> Owner
   Index<StringStream> Name
}
```

4.8 Nested classes

When dealing with nested classes, we face a problem because class defined in another class does not have right namespace set in **TypeDef** table and if we want to reconstruct fully qualified names of all types, we need to deal with this situation. The table **NestedClass** helps with this because it provides information about nesting. Nested class is references with attribute **NestedClass** and its parent is references with **EnclosingClass**.

Structure of table **NestedClass**.

```
struct NestedClass {
   Index<TypeDef> NestedClass
   Index<TypeDef> EnclosingClass
}
```

4.9 TypeLib ID & MVID

In order to find *TypeLib*, we first need to locate record in **TypeRef** table that refers to type named **GuidAttribute** and is from module **mscorlib**. After that, we search the table **MemberRef** for reference to record in **TypeRef** we found in the first step. The last step involves searching the table **CustomAttribute** for a reference to record found in the second step. There can be multiple such references so we need to check each one whether it points to string in *string stream* that can be matched with regular expression for generic unique identifier.

Finding *MVID* is fairly easy. We just need to take the very first record out of **Module** table and of it its attributes is actually called **Mvid** which points to *GUID stream*.

5 RESULTS

We implemented the reconstruction of .NET information in pure C++ based on the description given in previous chapter as a part of tool **fileinfo**. This tool is part of project RetDec [9] developed by Avast. It is used to extract various information out of binary files in multiple file formats.

We then compared the results with some of the tools mentioned in Chapter 3. We were not able to compare every single one because some of them does not provide any interface to automatize the testing. The interesting for us was robustness when dealing with corrupted, packed or obfuscated binaries, which are fairly common in the world of malware. We have downloaded 100,000 executable files from VirusTotal [10], which were marked with tags *peexe* and *assembly* and imported at least one symbol from library *mscoree.dll*.

The machine used for testing was server with 2x Intel Xeon E5-2699 v4@2.20GHz and 256 GB of RAM. Each program was run separately. All 100,000 samples were processed in parallel. There was a timeout of 60 seconds and if the tested program did not produce any result in that time it was killed. The success rate results are shown in Table 1. Errors means how many times the program returned something other than 0. Crashes show the number of segmentation faults, aborts and other unexpected terminations. Timeouts just means how many times we had to kill the program after 60 seconds. You can clearly see that the best results achieved **fileinfo**. The single timeout was caused by sample that had really complex .NET class hierarchy so it would need more than 60 seconds to finish.

Table 1: Success rate results.

	Errors	**Crashes**	**Timeouts**
fileinfo	0	0	1
monodis	905	47042	6
ikdasm	9342	149	192

In the Table 2, you can see performance results. Times you see in the table are calculated as average of 10 consecutive test runs. Running time means how long were tests actually running on all 100,000 samples. Total time is sum of all durations of every single instance of tested program. Even though fastest was **monodis**, we need to take into account that it crashed a lot of times, therefore skewing the performance results.

Table 2: Performance results.

	Running time	**Total time**
Fileinfo	6m 45s	9h 45m 56s
Monodis	1m 47s	1h 47m 12s
Ikdasm	26m 36s	1d 13h 46m 26s

6 CONCLUSION AND FUTURE WORK

We have dealt with extraction of data types, *TypeLib* identifier and *Module Version* identifier (*MVID*) out of .NET executable files in this paper. We have described how we can reconstruct classes, methods, fields, properties, inheritance hierarchy and nesting of classes. We have also described the way to properly extract *TypeLib* identifier and *MVID*. We implemented proposed methods and compared the results against other available .NET disassemblers. We pointed out the mistakes that are made in the current world of malware hunting. During work on this article, we have contacted VirusTotal [10] and reported them our findings about some packed and obfuscated binaries that do not have any or have an incorrect *TypeLib* identifier shown on their web. We also made pull request with adding support for unoptimized streams into project YARA [11].

The achieved results are more than satisfactory. The hardened implementation backed up by research in reverse engineering of .NET executable files proved to be better than usual implementations based on .NET framework reflection. In the time of writing of this article, the source code is not available as it is part of retargetable decompiler RetDec [9] developed by Avast Software. However, it is already in the process of open sourcing and it should be available on official Avast GitHub profile page by the end of the year 2017.

In the future, we would like to research for additional features that can be extracted from .NET executable files. We would also like to build further on the proposed methods and start creating detection patterns out of the extracted information. Clustering and machine learning are also the long-term goals for .NET executable files.

Acknowledgment

This work was supported by Avast Scholar program created by Avast Software s.r.o.

Sources

1. 0xd4d/dnlib: dnlib is a library that can read, write and create .NET assemblies and modules. https://github.com/0xd4d/dnlib. [online; 30-11-2017].
2. ECMA International. *Standard ECMA-335 – Common Language Infrastructure (CLI).* https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.pdf. [online; 26-11-2017]. June 2012.
3. Ildasm.exe (IL Disassembler) | Microsoft Docs. https://docs.microsoft.com/en-us/dotnet/framework/tools/ildasm-exe-il-disassembler. [online; 30-11-2017].
4. jfrijters/ikdasm: Managed.Reflection based ildasm clone. https://github.com/jfrijters/ikdasm. [online; 30-11-2017].
5. Lidin, S. *.NET IL Assembler.* Apress, 2014. ISBN 1430267615.
6. Microsoft. *PE Format.* https://msdn.microsoft.com/en-us/library/windows/desktop/ms680547(v=vs.85).aspx. [online; 26-11-2017].
7. mono/mono: Mono open source ECMA CLI, C# and .NET implementation. https://github.com/mono/mono. [online; 30-11-2017].
8. Pontiroli, S., Martinez, R., *The rise of .NET and Powershell malware.* https://securelist.com/the-rise-of-net-and-powershell-malware/72417/. [online; 26-11-2017].
9. Retargetable Decompiler. https://retdec.com/, [online; 28-11-2017].
10. VirusTotal – Free Online Virus, Malware and URL Scanner. https://www.virustotal.com/en/. [online; 26-11-2017].
11. VirusTotal/yara: The pattern matching swiss knife. https://github.com/VirusTotal/yara. [online; 26-11-2017].

12. Wallace, B. *Using .NET GUIDs to help hunt for malware.* https://www.virusbulletin.com/uploads/pdf/magazine/2015/vb201506-NET-GUIDs.pdf. [online; 27-11-2017].

Contact
Ing. Marek Milkovič
Božetěchova 1/2, 612 00 Brno-Královo Pole, Czechia
Tel: +421 948 038 667
Email: imilkovic@fit.vutbr.cz