paloalto NETWORKS

Dotnetfile

Parsing .NET PE Files Has Never Been Easier

Yaron Samuel

RECON Montreal

June 2022

Who am I

- Yaron Samuel - @yaron_samuel
- Principal Malware Reverse Engineer
- 10+ years in the cyber-security field
- Works at Palo Alto Networks

⚽ Soccer fan

🐶 Dog person

Contributors

Dominik Reichel

Senior Reverse Engineer

@TheEnergyStory

Former Palo Alto Networks employee

Germany

Bob Jung

Senior Manager & an awesome Reverse Engineer

Palo Alto Networks

New Mexico, USA

Yaron Samuel

Your Humble Servant

Palo Alto Networks

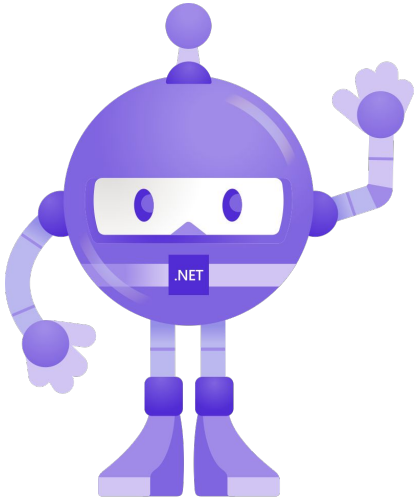
Israel

Agenda

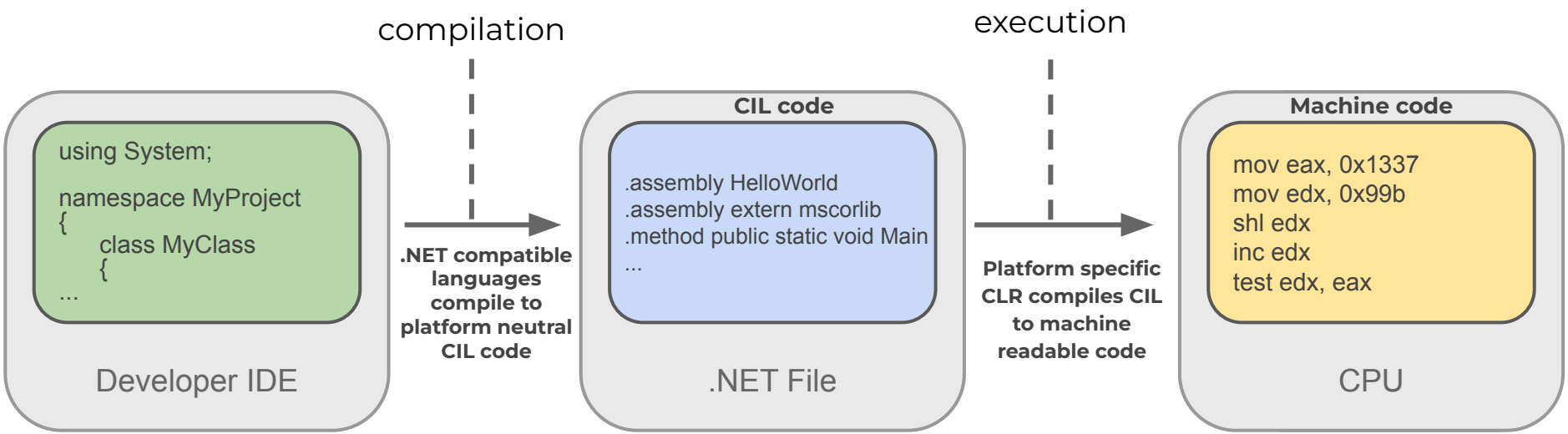
- What is .NET
  - Background
  - Execution and compilation
  - File Format
- Dotnetfile Library - New open source Python Library
  - Motivation
  - Features
  - Examples

What is .NET

- Software Framework
  - ✓ Free
  - ✓ Managed
  - ✓ Open Source*
  - ✓ Cross Platform*
- Supports C#, VB .NET, F# and a few other programming languages
  - ✓ High-level
  - ✓ May speed up development process compared to lower level languages
  - ✓ Native code interface exists
- Primarily developed by Microsoft
- dotnet bot - the .NET community mascot -->



.NET compilation & execution



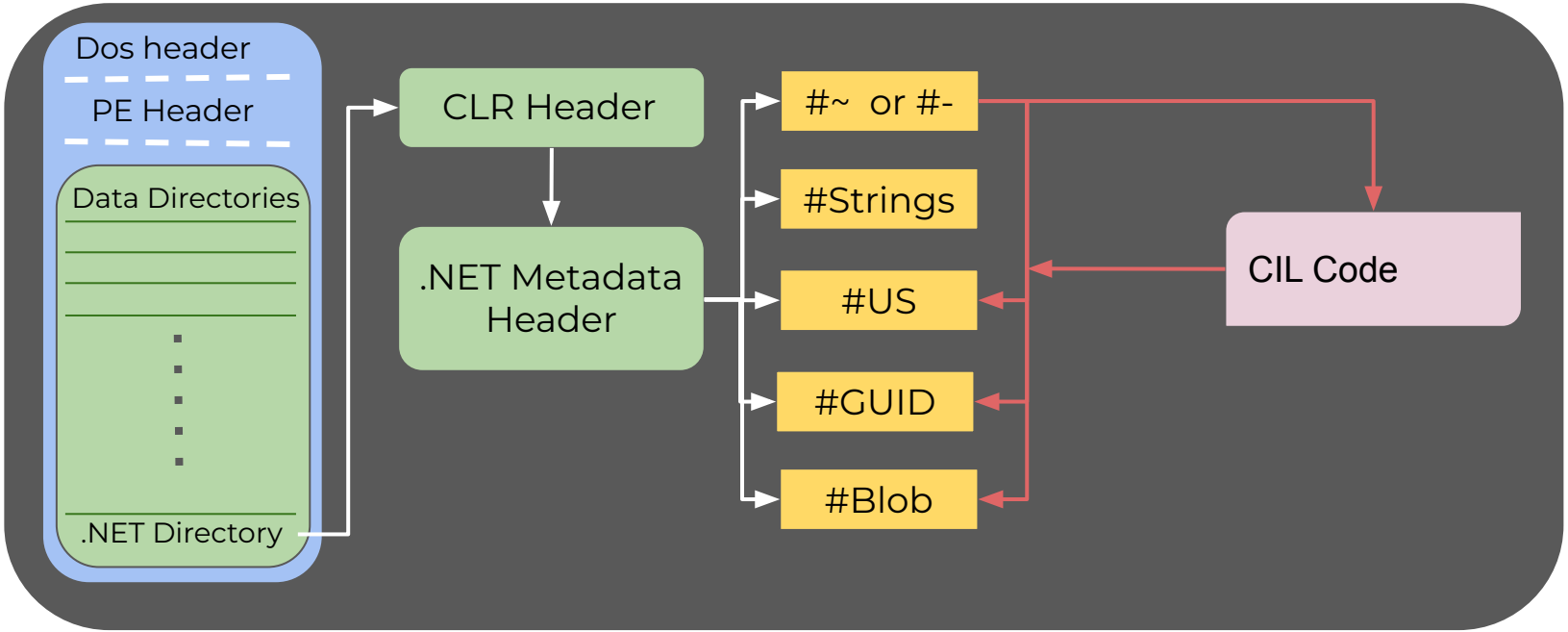
CIL = Common Intermediate Language

CLR = Common Language Runtime

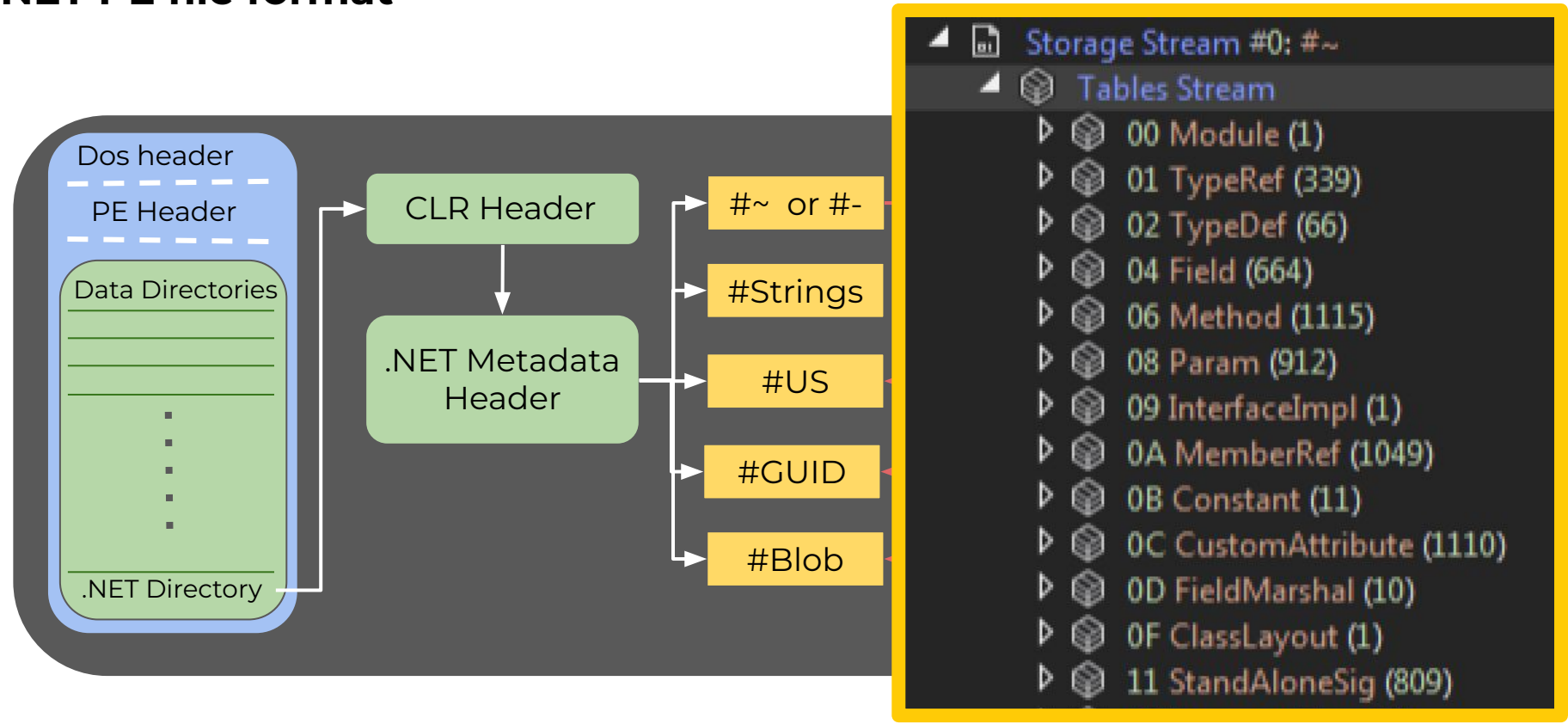
.NET Executables

- The compiler compiles the programing language into CIL packaged into various executable forms
- We will concentrate on .NET PE executables for Windows
- The .NET PE file format extends the regular native Windows PE file format
- .NET PE files usually have no native code, but only minimal native stubs alongside the compiled CIL

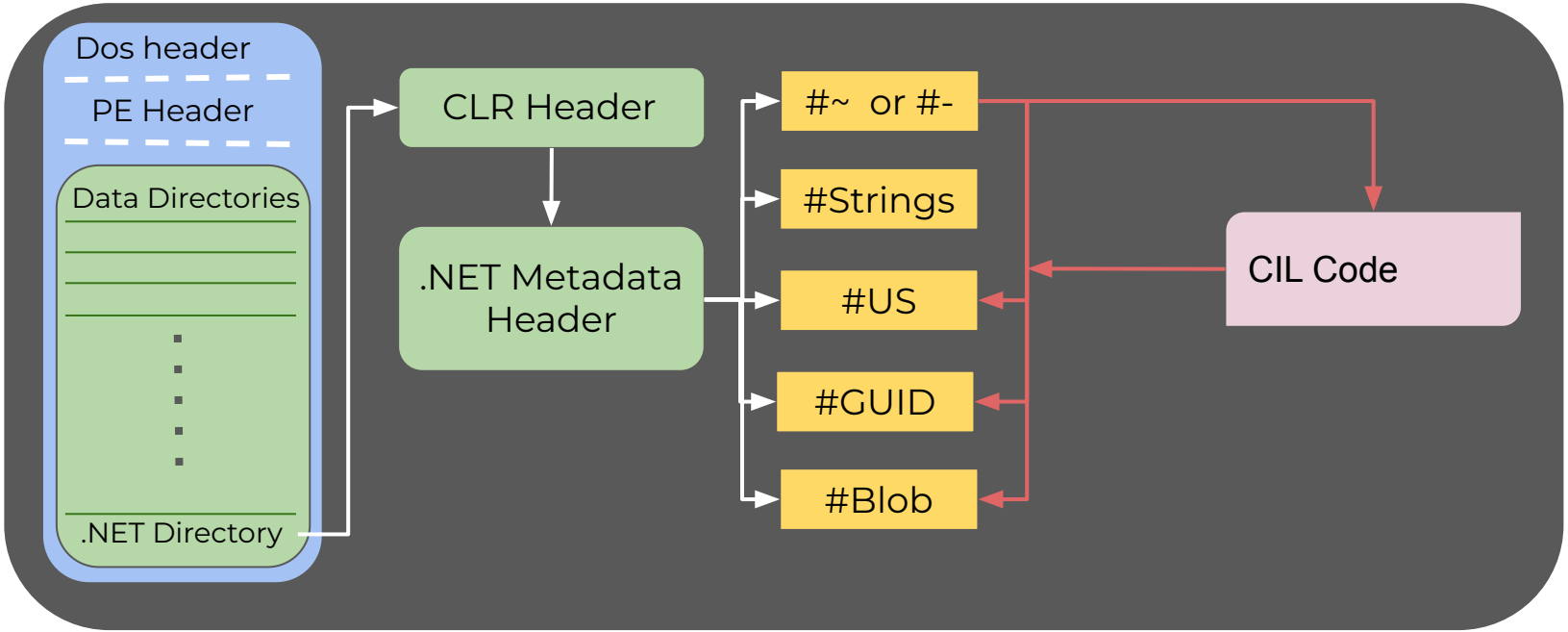
.NET PE file format



.NET PE file format



.NET PE file format



Native vs .NET

	Native PE	.NET PE
Native Code	✓	✗ in some cases there is native code
Import table	✓ Usually meaningful	✗ oftentimes contains only mscoree.dll
Export table	✓ Exist in DLLs	✗ Usually don't exist
Strings	✓	✓
.NET header	✗	✓
CIL	✗	✓

Dotnetfile Library

Dotnetfile Library - Motivation

- Malware authors like .NET
- Static analysis of .NET files using engines meant for native PE files has shortcomings
- Parsing the rich set of .NET fields can assist us in various applications
- Requirements:
  - Static analysis of .NET PE files
  - OS agnostic
  - Using our lingua franca - Python
- We could not find an alternative that did not depend on .NET
  - By the time we have finished “dnfile” and “dncil” were released
  - We believe that none of the libraries, including ours, is superior
  - Each library has its own advantages

Dotnetfile Library

- Named based on the legendary pefile library
- Usage is also very similar, in fact the main object - DotNetPE inherits from PE

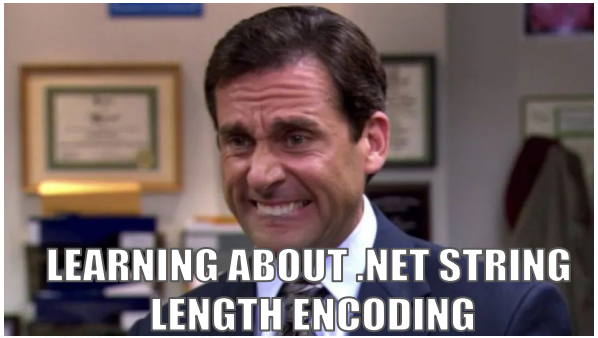
```
In [1]: import dotnetfile

In [2]: dn = dotnetfile.DotNetPE('dotnet_malware.exe')

In [3]: dn.Assembly.get_assembly_name()
Out[3]: 'NjRat 0.7D'

In [4]: dn.Assembly.get_assembly_version_information()
Out[4]: Struct.AssemblyInfo(MajorVersion=0, MinorVersion=0, BuildNumber=0, RevisionNumber=7)
```

Dotnetfile - Overview

- ✓ .NET metadata header parsing
- ✓ .NET Streams parsing
  - Metadata Tables
  - Strings
  - Methods
- ✓ Advanced features
  - Fingerprinting techniques
  - Entry points discovery
  - “Anti-Metadata”
- ✓ .NET Resource parsing
  - ✗ Only raw resources, no deserialization
- ✗ Decompilation is not part of the library

Strings & User strings

- .NET has 2 types of strings, held in 2 streams
  - #strings - System strings
    - Class names, methods names, member names, etc.
    - Strings are stored next to one another
  - #US - User strings
    - String literals that are being used by the code
    - Strings are stored in the form of length+value
    - String length encoding is somewhat strange



- If the first one byte of the 'blob' is $0bbbbbbb_2$, then the rest of the 'blob' contains the $bbbbbbb_2$ bytes of actual data.
- If the first two bytes of the 'blob' are $10bbbbbb_2$ and x, then the rest of the 'blob' contains the $(bbbbbb_2 << 8 + x)$ bytes of actual data.
- If the first four bytes of the 'blob' are $110bbbbb_2$, x, y, and z, then the rest of the 'blob' contains the $(bbbbb_2 << 24 + x << 16 + y << 8 + z)$ bytes of actual data.

Strings & User strings

```
In [21]: dn.get_user_stream_strings()[100:105]
Out[21]: ['Chat', '!~Hacker~!', 'Enter Your NickName', '~', '[']

In [22]: dn.get_strings_stream_strings()[:5]
Out[22]: ['', '<Module>', 'mscorlib', 'Microsoft.VisualBasic', 'MyApplication']
```

Metadata tables

- The “#~” stream contains up to 56 metadata tables
- The tables contain most of the .NET metadata
- The tables reveal tons of useful information
- Notable tables:
  - TypeRef - Referenced types
  - TypeDef - Defined types (classes)
  - ImplMap - Unmanaged methods
  - ModuleRef - External Libraries

Metadata tables

```
In [14]: dn.metadata_tables_lookup
Out[14]:
{'Module': <dotnetfile.parser.MetadataTable at 0x1073f6760>,
 'TypeRef': <dotnetfile.parser.MetadataTable at 0x10740e610>,
 'TypeDef': <dotnetfile.parser.MetadataTable at 0x107424b50>,
 'Field': <dotnetfile.parser.MetadataTable at 0x107601a90>,
 'MethodDef': <dotnetfile.parser.MetadataTable at 0x10774dfd0>,
 'Param': <dotnetfile.parser.MetadataTable at 0x107a926a0>,
 'InterfaceImpl': <dotnetfile.parser.MetadataTable at 0x107c2e430>,
 'MemberRef': <dotnetfile.parser.MetadataTable at 0x107c2e640>,
 'Constant': <dotnetfile.parser.MetadataTable at 0x107e84550>,
 'CustomAttribute': <dotnetfile.parser.MetadataTable at 0x107e8a670>,
 'FieldMarshal': <dotnetfile.parser.MetadataTable at 0x1080e51f0>,
 'ClassLayout': <dotnetfile.parser.MetadataTable at 0x1080e5dc0>,
 'StandAloneSig': <dotnetfile.parser.MetadataTable at 0x1080e5f10>,
 'PropertyMap': <dotnetfile.parser.MetadataTable at 0x1081d7e20>,
 'Property': <dotnetfile.parser.MetadataTable at 0x1081e5fd0>,
 'MethodSemantics': <dotnetfile.parser.MetadataTable at 0x108289b20>,
 'MethodImpl': <dotnetfile.parser.MetadataTable at 0x1083bd3d0>,
 'ModuleRef': <dotnetfile.parser.MetadataTable at 0x1083bd4f0>,
 'TypeSpec': <dotnetfile.parser.MetadataTable at 0x1083bd730>,
 'ImplMap': <dotnetfile.parser.MetadataTable at 0x1083c3a60>,
 'Assembly': <dotnetfile.parser.MetadataTable at 0x1083ccbe0>,
 'AssemblyRef': <dotnetfile.parser.MetadataTable at 0x1083ccfd0>,
 'ManifestResource': <dotnetfile.parser.MetadataTable at 0x1083d2430>,
 'NestedClass': <dotnetfile.parser.MetadataTable at 0x1083debb0>,
 'GenericParam': <dotnetfile.parser.MetadataTable at 0x1083e8730>,
 'MethodSpec': <dotnetfile.parser.MetadataTable at 0x1083ec430>,
 'GenericParamConstraint': <dotnetfile.parser.MetadataTable at 0x1084007c0
```

- *00* **Module** (1): Generation - 2b | Name - string | Mvid - guid | EncId - guid | EncBaseId - guid
- *01* **TypeRef** (339): ResolutionScope - ResolutionScope | TypeName - string | TypeNamespace - string
- *02* **TypeDef** (66): Flags - 4b | TypeName - string | TypeNamespace - string | Extends - TypeDefOrRef
- *04* **Field** (664): Flags - 2b | Name - string | Signature - blob
- *06* **MethodDef** (1115): RVA - 4b | ImplFlags - 2b | Flags - 2b | Name - string | Signature - blob | Param
- *08* **Param** (912): Flags - 2b | Sequence - 2b | Name - string
- *09* **InterfaceImpl** (1): Class - TypeDef | Interface - TypeDefOrRef
- *0A* **MemberRef** (1049): Class - MemberRefParent | Name - string | Signature - blob
- *0B* **Constant** (11): Type - 1b | Padding - 1b | Parent - HasConstant | Value - blob
- *0C* **CustomAttribute** (1110): Parent - HasCustomAttribute | Type - CustomAttributeType | Value - bl
- *0D* **FieldMarshal** (10): Parent - HasFieldMarshal | NativeType - blob
- *0F* **ClassLayout** (1): PackingSize - 2b | ClassSize - 4b | Parent - TypeDef
- *11* **StandAloneSig** (809): Signature - blob
- *15* **PropertyMap** (29): Parent - TypeDef | PropertyList - Property
- *17* **Property** (322): Flags - 2b | Name - string | Type - blob
- *18* **MethodSemantics** (625): Semantics - 2b | Method - MethodDef | Association - HasSemantics
- *19* **MethodImpl** (1): Class - TypeDef | MethodBody - MethodDefOrRef | MethodDeclaration - Metho
- *1A* **ModuleRef** (3): Name - string
- *1B* **TypeSpec** (31): Signature - blob
- *1C* **ImplMap** (8): MappingFlags - 2b | MemberForwarded - MemberForwarded | ImportName - string
- *20* **Assembly** (1): HashAlgId - 4b | MajorVersion - 2b | MinorVersion - 2b | BuildNumber - 2b | Revisio
- *23* **AssemblyRef** (6): MajorVersion - 2b | MinorVersion - 2b | BuildNumber - 2b | RevisionNumber - 2
- *28* **ManifestResource** (21): Offset - 4b | Flags - 4b | Name - string | Implementation - Implementation
- *29* **NestedClass** (23): NestedClass - TypeDef | EnclosingClass - TypeDef
- *2A* **GenericParam** (7): Number - 2b | Flags - 2b | Owner - TypeOrMethodDef | Name - string

ImplMap & ModuleRef - the hidden import table

- ImplMap holds information about unmanaged methods that can be reached from managed code
- ModuleRef contains the respective module information
- Malware may use it for obvious reasons

```
// Token: 0x060003F0 RID: 1008
[DllImport("User32.dll")]
private static extern bool GetLastInputInfo(ref GClass7.GStruct0 gstruct0_0);
```

```
In [52]: dn.ImplMap.get_platform_invoke_information()
Out[52]:
['msvcrt.memcpy',
 'kernel32.beginupdateresource',
 'kernel32.endupdateresource',
 'kernel32.updateresource',
 'user32.getlastinputinfo',
 'kernel32.beginupdateresource',
 'kernel32.updateresource',
 'kernel32.endupdateresource']
```

Resource access

- .NET allows to pack resources into the executable
  - And even sub-resources are supported
- Resources are stored in serialized form
  - Dotnetfile doesn't deserialize the resources

```
In [89]: for res in dn.get_resources():
    ...:     res_data.append([res['Name'],res['Visibility'],res['NumberOfSubResources']])
    ...:

In [90]: print(tabulate.tabulate(res_data))
-----------------------  ------  --
NJRAT.RGv.resources      public   0
NJRAT.port.resources     public   0
NJRAT.notf.resources     public   0
NJRAT.Mic.resources      public   0
NJRAT.Resources.resources public  8
NJRAT.FURL.resources     public   0
NJRAT.Cam.resources      public   0
NJRAT.Manager.resources  public  35
```

MemberRef Hash

- Innovative fingerprinting technique
  - Can help group, cluster and detect .NET samples
- The MemberRef table contains mostly .NET runtime constructs
  - methods, properties, fields and so on
  - can't be easily obfuscated
- For each row we take the member name and the table name of the corresponding class
- We textually concatenate all the resulting values and hash the result

```
In [99]: dn.MemberRef.get_memberref_hash()
Out[99]: '8dd194d530279bee74d4ec98109839 8a06f404deb8a82bb1d4e5a4f45137fc6f'
```

MemberRef Hash - Contd.

Table ID Enum
0 TypeDef
1 TypeRef
2 ModuleRef
3 MethodDef
4 TypeSpec

Class: 0x44
Name: 0x325

Table ID = 0x44 & (0b111) = 4 - TypeSpec
Row Index = 0x44 >> 3 = 8

#Strings Heap Offset

TypeSpec
.ctor

Class: 0x299
Name: 0x5979

Table ID = 0x299 & (0b111) = 1 - TypeRef
Row Index = 0x299 >> 3 = 0x53

#Strings Heap Offset

TypeRef
Enter

Class: 0x44
Name: 0x597f

Table ID = 0x44 & (0b111) = 4 - TypeSpec
Row Index = 0x44 >> 3 = 8

#Strings Heap Offset

TypeSpec
get_Count

SHA256

"TypeSpec-.ctor"

+

"TypeRef-Enter"

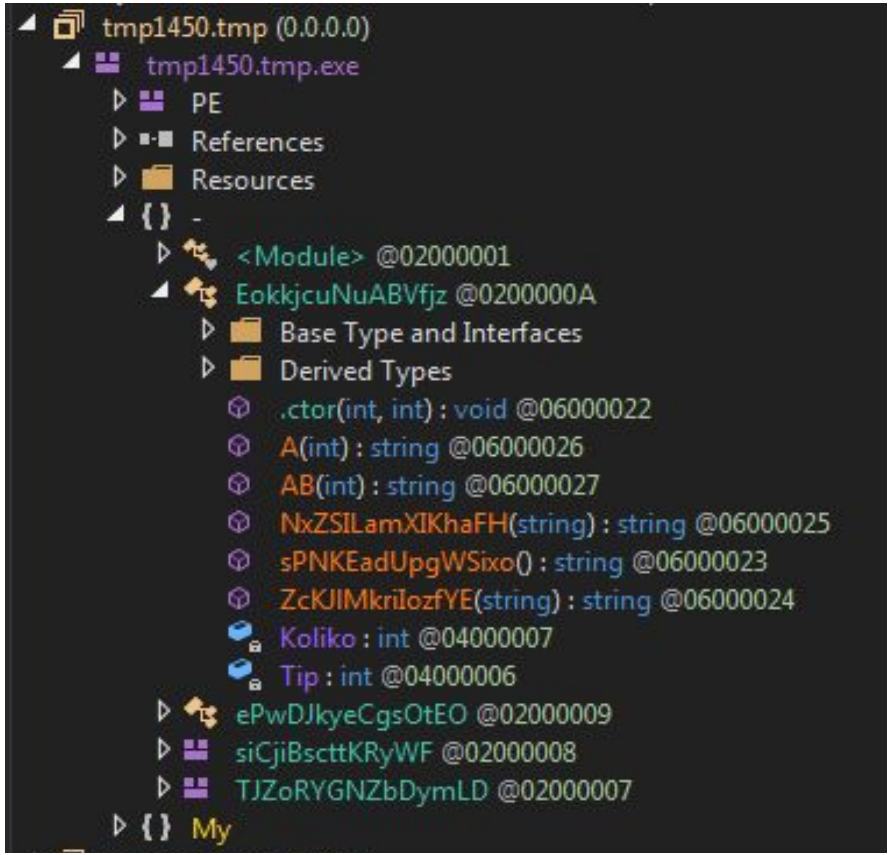
+

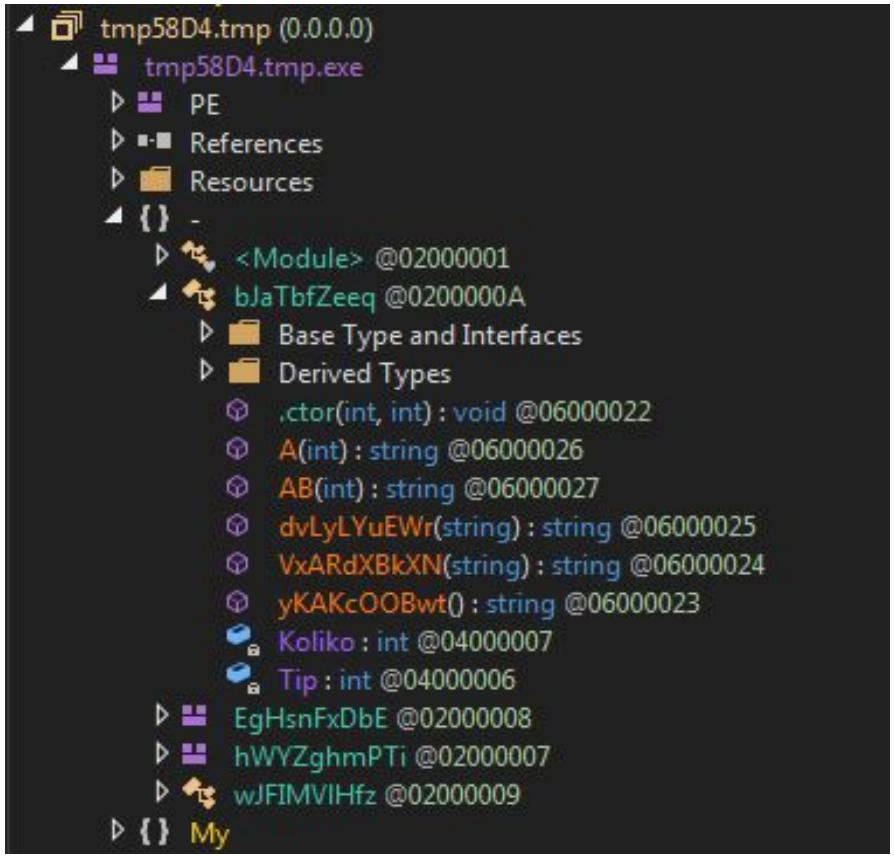
"TypeSpec-get_Count"

MemberRefHash

`8dd194d530279bee74d4ec981098398a06f404deb8a82bb1d4e5a4f45137fc6f`

MemberRef Hash - Example





MemberRef Hash - Example


```
In [33]: dn1 = dotnetfile.DotNetPE('311b77ec73a4881063e0b270

In [34]: dn2 = dotnetfile.DotNetPE('e372292d0d361058d449b1c0

In [35]: dn1.MemberRef.get_memberref_hash()
Out[35]: '2297027b715e172bb32de9641078bac067d2638af701a297a8a7619c27fd65b4'

In [36]: dn2.MemberRef.get_memberref_hash()
Out[36]: '2297027b715e172bb32de9641078bac067d2638af701a297a8a7619c27fd65b4'
```

MemberRef Hash - Example

```
TypeRef-.ctor
TypeRef-.ctor
TypeRef-.ctor
TypeRef-.ctor
TypeRef-.ctor
TypeSpec-.ctor
TypeSpec-.ctor
TypeSpec-.ctor
TypeSpec-get_GetInstance
TypeSpec-get_GetInstance
TypeSpec-get_GetInstance
TypeSpec-get_GetInstance
TypeRef-.ctor
TypeRef-.ctor
TypeRef-.ctor
TypeRef-GetObjectValue
TypeRef-Equals
TypeRef-GetHashCode
TypeRef-GetTypeFromHandle
TypeRef-ToString
TypeRef-CreateInstance
TypeRef-.ctor
TypeRef-.ctor
TypeSpec-m_ThreadStaticValue
TypeRef-.ctor
TypeRef-.ctor
TypeRef-.ctor
TypeRef-Concat
TypeRef-Contains
TypeRef-get_Length
TypeRef-ToString
TypeRef-CompareString
TypeRef-Delete
TypeRef-SetProjectError
TypeRef-ClearProjectError
TypeRef-GetExecutingAssembly
TypeRef-.ctor
TypeRef-GetObject
TypeRef-Load
TypeRef-get_EntryPoint
TypeRef-Invoke
TypeRef-.ctor
TypeRef-.ctor
TypeRef-CopyArray
TypeRef-Read
TypeRef-.ctor
TypeRef-GetCurrentProcess
TypeRef-get_MainModule
TypeRef-get_FileName
TypeRef-ToString
TypeRef-.ctor
TypeRef-AddResource
TypeRef-AddResource
TypeRef-Generate
TypeRef-Close
TypeRef-Dispose
TypeRef-GetTempFileName
TypeRef-Replace
TypeRef-.ctor
TypeRef-Next
TypeRef-Randomize
TypeRef-StrReverse
TypeRef-.ctor
TypeRef-SetCreationTime
TypeRef-SetLastAccessTime
TypeRef-SetLastWriteTime
TypeRef-.ctor
TypeRef-set_WindowStyle
TypeRef-set_FileName
TypeRef-set_Arguments
TypeRef-Start
TypeRef-Kill
TypeRef-.ctor
TypeRef-ToCharArray
TypeRef-Rnd
TypeRef-Int
TypeRef-Round
TypeRef-Append
TypeRef-ToString
TypeRef-Get_Default
TypeRef-GetBytes
TypeRef-ToBase64String
TypeRef-FromBase64String
TypeRef-GetString
TypeRef-.ctor
TypeRef-CreateCompiler
TypeRef-.ctor
TypeRef-set_GenerateExecutable
TypeRef-set_OutputAssembly
TypeRef-get_ReferencedAssemblies
TypeRef-Add
TypeRef-get_EmbeddedResources
TypeRef-set_CompilerOptions
TypeRef-CompileAssemblyFromSource
TypeRef-CreateProjectError
TypeRef-Empty
TypeRef-Concat
TypeRef-.ctor
TypeRef-Append
TypeRef-.ctor
TypeRef-.ctor
```

TypeRef Hash

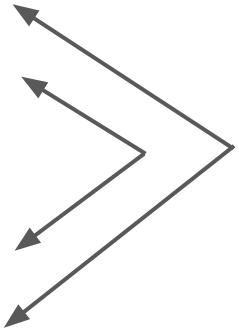
- Full credit to GDATA for the original idea
- The main idea is to calculate sha256 over all the referenced .NET types
- Re-implementation of typeref hash with small improvements
  - Our version uses the resolution scope names instead of the namespace names as they're always present
  - Optional flag to skip types that reference each other as added by some .NET protectors

TypeRef Hash - Example

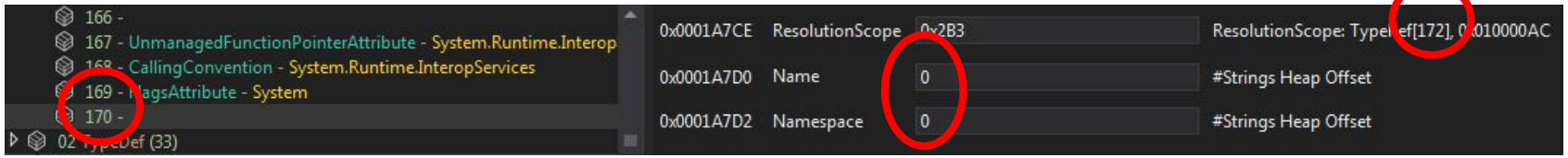
TypeRef entries that reference each other - likely garbage
No ”Namespace”

0x0001A444	ResolutionScope	0x4B	sxeHhPBNraWIHeFoxc (ResolutionScope: TypeRef[18], 0x01000012)
0x0001A446	Name	0x274	6SFRArCsVqhLk4jhGT (#Strings Heap Offset)
0x0001A448	Namespace	0	#Strings Heap Offset

0x0001A43E	ResolutionScope	0x4F	6SFRArCsVqhLk4jhGT (ResolutionScope: TypeRef[19], 0x01000013)
0x0001A440	Name	0x261	sxeHhPBNraWIHeFoxc (#Strings Heap Offset)
0x0001A442	Namespace	0	#Strings Heap Offset

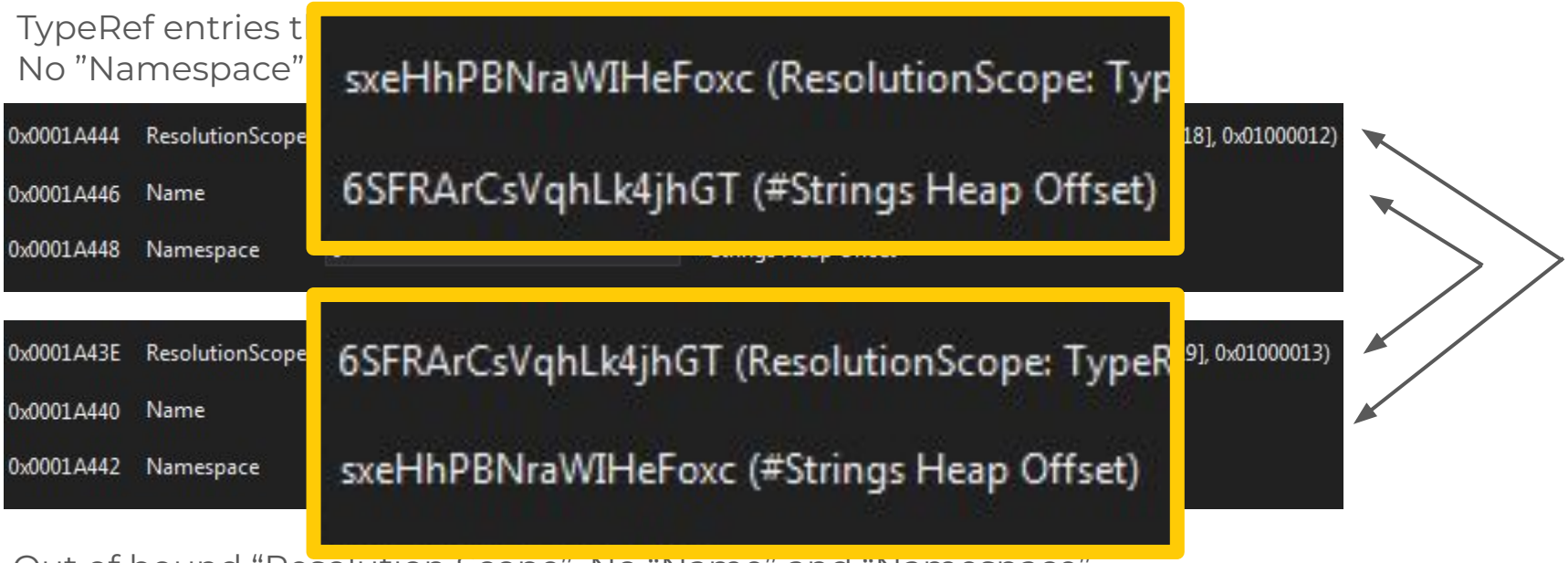


Out of bound “Resolution Scope”, No “Name” and “Namespace”

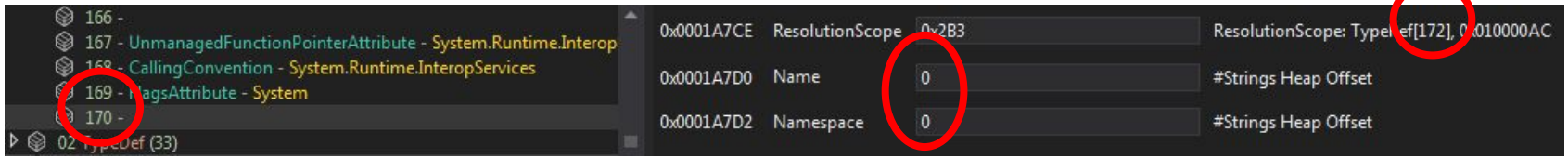


TypeRef Hash - Example

TypeRef entries t
No ”Namespace”

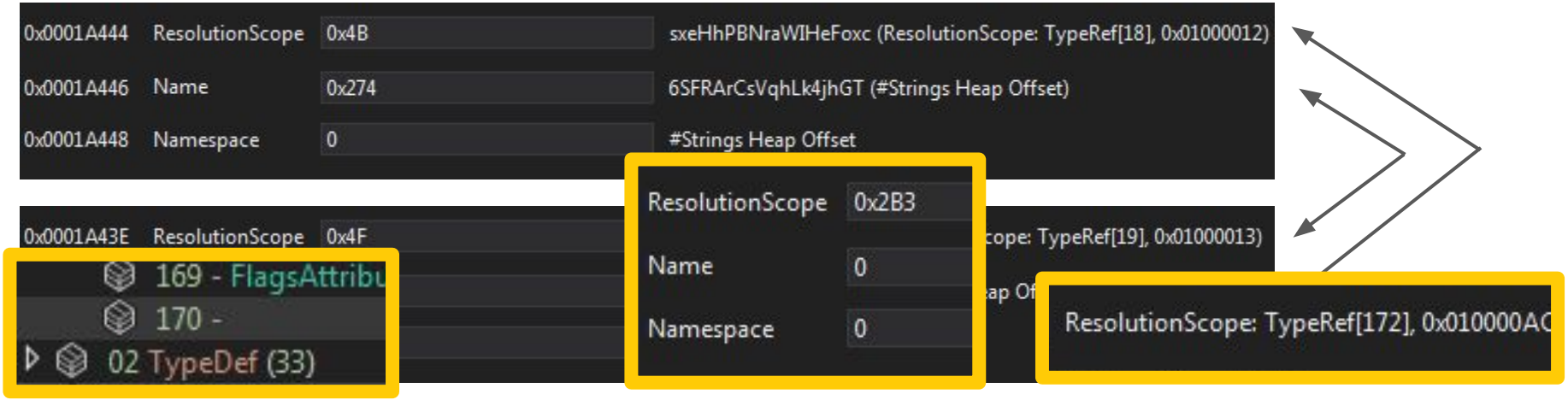


Out of bound “Resolution Scope”, No “Name” and “Namespace”

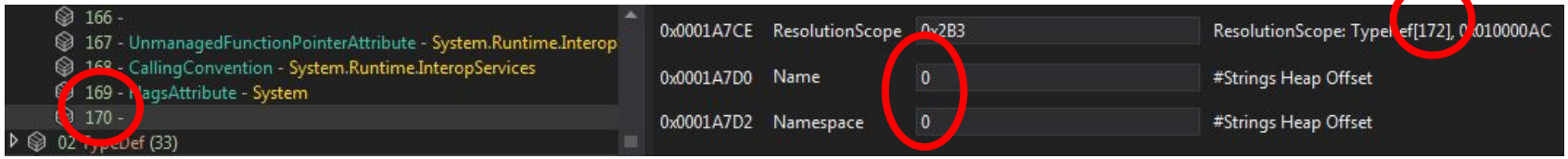


TypeRef Hash - Example

TypeRef entries that reference each other - likely garbage
No ”Namespace”



Out of bound “Resolution Scope”, No “Name” and “Namespace”



Entry Points Discovery

- .NET Assemblies usually have no defined entry-point
- Dotnetfile provides means to easily locate the possible entry-point
  - Searches for public classes or nested public classes
  - Filters out methods with non-EP characteristics
  - Parses out the method signature for a small but prevalent subset of signatures
  - It is not bulletproof but works pretty well

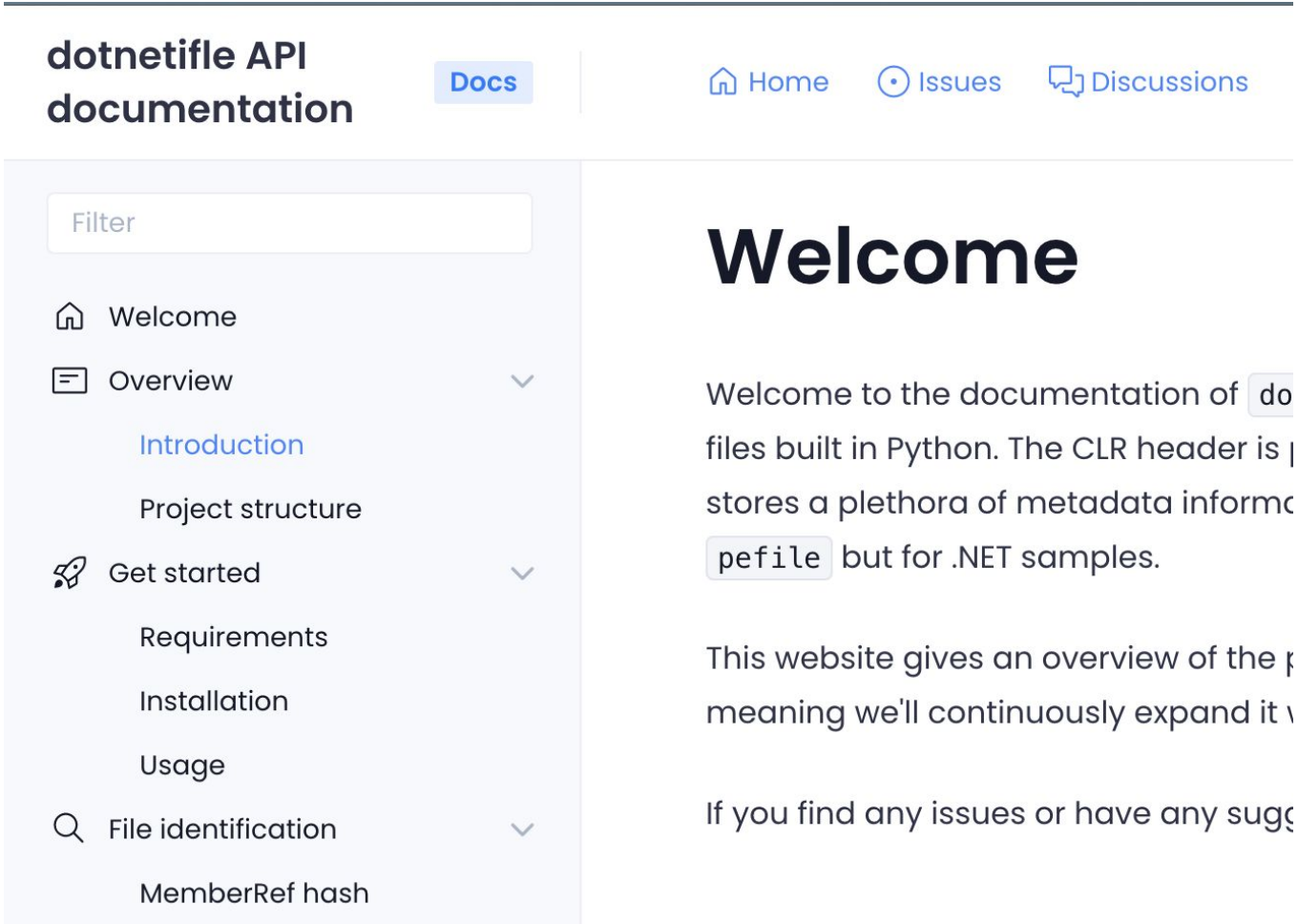
```
In [125]: dn.MethodDef.get_entry_points()
Out[125]: [Struct.EntryPoint(Method='Main', Signature={'hasthis': False, 'return': 'System.Void',
'parameter': 'System.String[]'}, Type='MyApplication', Namespace='NJRAT.My')]
```

```
In [128]: for ep_struct in dn_assembly.MethodDef.get_entry_points():
     ...:     print(f'{ep_struct.Type}-{ep_struct.Method}')
     ...:
wbemcon_RES-Sygtxa_FiceAnawqsis
wbemcon_RES-Systec_WkpeColleotor
wbemcon_RES-AphRModec_3006
wbemcon_RES-mmcws_Scfg
wbemcon_RES-in41_ws_syafWCN
```

Anti-Metadata

- Packers and malware often try to break parsing
- We devised some logic to identify abnormal .NET metadata structure
- The feature was named Anti-Metadata
- The logic includes
  - Checks if there are fake .NET streams
  - Validates whether the Module and Assembly table has more than 1 entry
  - Detects invalid string entries - added by ConfuserEx
  - Checks if there are extra bytes in the .NET metadata header
  - Looks for invalid TypeRef entries
  - Checks if the data directories number (OPTIONAL_HEADER.NumberOfRvaAndSizes) was tampered, so it effectively hides the .NET data directory
  - Detects self referencing TypeRef entries - by examining the corresponding resolution scope

Documentation



references

Source code: https://github.com/pan-unit42/dotnetfile/

Documentation: https://pan-unit42.github.io/dotnetfile/

Common Language Infrastructure Spec:
https://www.ecma-international.org/wp-content/uploads/ECMA-335_6th_edition_june_2012.pdf

GDATA - Introducing the typeref hash:
https://www.gdatasoftware.com/blog/2020/06/36164-introducing-the-typerefhash-trh

paloalto NETWORKS

Dotnetfile

Parsing .NET PE Files Has Never Been Easier

Yaron Samuel

RECON Montreal

June 2022