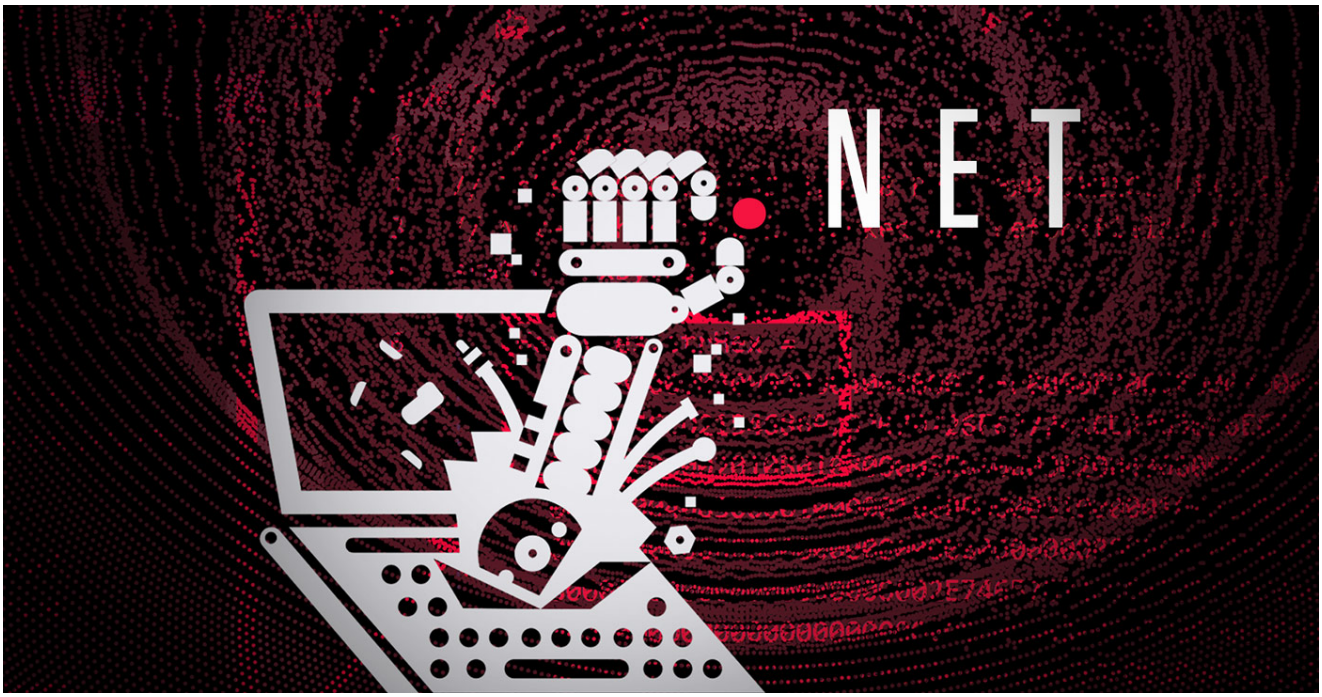
DotNET Loaders

blog.reversinglabs.com/blog/dotnet-loaders



Threat Research | March 12, 2021

Blog Author

Robert Simmons, Independent malware researcher and threat researcher at ReversingLabs. Read More...

Many families of remote access trojan (RAT) are .NET executables. As was observed in the blog post [1] from one year ago about RevengeRAT among others, much of this malware is delivered in another .NET executable with the payload encoded as an embedded text string. These RATs when they're encoded as text and posted to pastebin like sites are tracked by Scumbots [2]. An additional method of hunting for this type of dropper or loader is to leverage the *dotnet* module in YARA. This module is a parser for .NET executables and presents the parsed components that it finds as a dataset inside YARA that can then be leveraged in the conditions section of a rule. In the following analysis, this parsed dataset and the user strings [3] entry in particular are used to identify .NET executables that have another PE executable encoded as one of those user strings.

Making the Rule

The first step is to examine one of the .NET loaders in question [4]. The examples here show a sample that delivers RevengeRAT [5] according to Malpedia analysis of the payload contained within it. This embedded and encoded PE file can be seen in the screenshot of the loader as analyzed by dnSpy [6] shown in Figure 1.

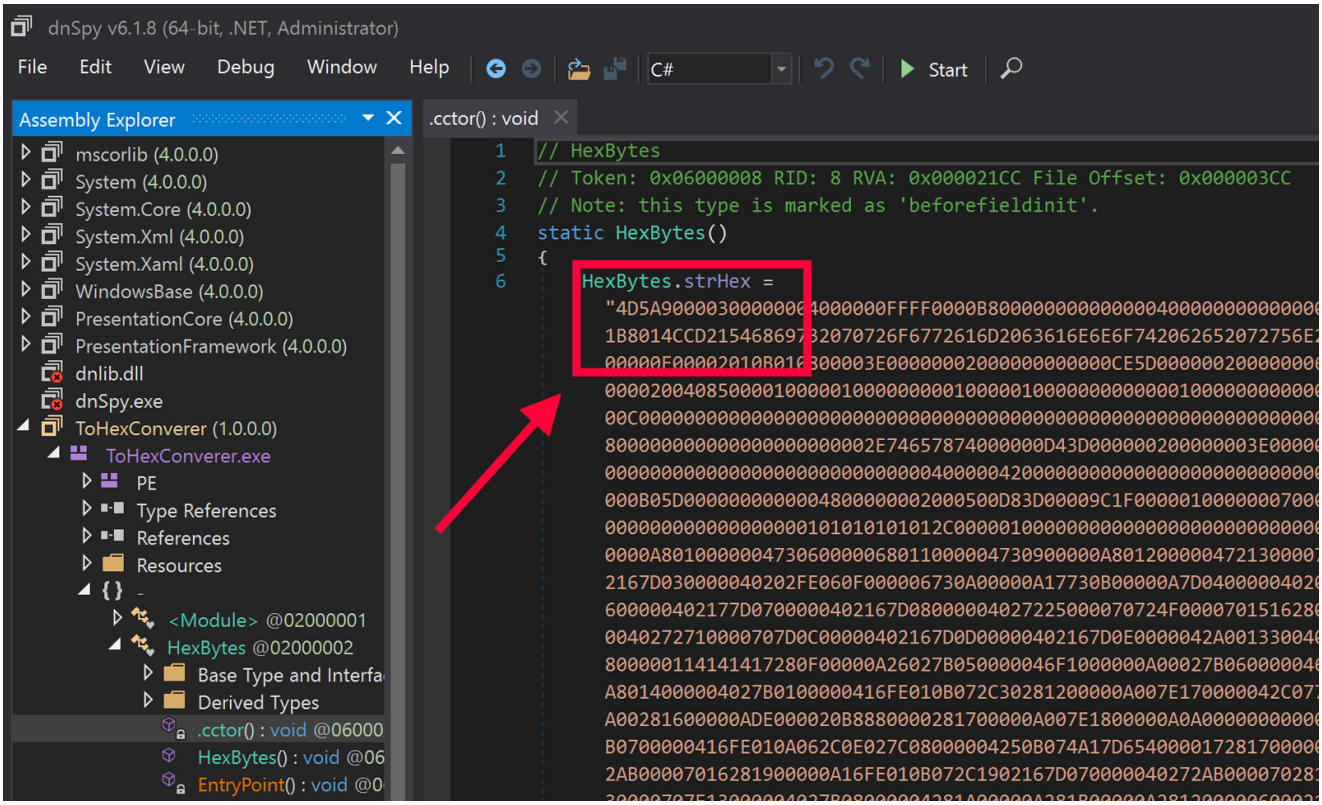


Figure 1: PE File Encoded as Text and Embedded in .NET Code

This encoded string is also visible in the debug data returned by YARA on the command line. The "-D" command line switch shows all the data that any loaded modules have available [7]. This is an extremely powerful feature for building rules because one can essentially see what YARA sees in a file and use that to build conditions for your new rules. The output from this command line switch when applied to the .NET sample above is shown in Figure 2. The string with the encoded payload is highlighted.

Figure 2: Debugging Output from YARA's Dotnet Module

As can be seen in the debugging output, the string is interspersed with null bytes, therefore these must be included when building the regular expression. Therefore, the characters to be used in the regular expression are the following.

4	\x00	D	\x00	5	\x00	A	\x00

Using these characters, the base of the regular expression is the following.

```
/4\x00D\x005\x00A\x00/
```

In addition to the above, two constraints can be included to reduce false positives. First, the carrot metacharacter "^" is added as the first character in the regex to make sure that only matches at the beginning of the string are possible. Second, a dot wildcard that matches any character with a repetition of 186 or more times is added to the end of the regular expression. The number chosen here for the repetition is based on research done on the smallest possible PE file with that size being 97 bytes [8]. This number is doubled because of the interspersed null bytes and the four characters of the PE magic number along with its accompanying null bytes are subtracted from the total.

```
186 = (2 * 97) - 8
```

Putting these parts together, the regular expression is the following.

```
/^4\x00D\x005\x00A\x00.{186,}/
```

Next we need to account for capitalization. Some of the samples use an all lowercase alphabet to encode the embedded PE file. To account for that, two small character sets must be used in place of the "D" and "A" characters in the regular expression. After making this change, the regular expression is the following.

```
/^4\x00[dD]\x005\x00[aA]\x00.{186,}/
```

Finally, this regular expression needs to be applied to each of the user strings that YARA's dotnet module parses out of a file. With the release of YARA 4.0, the syntax for iterating over this type of structure is really simple and easy to use [9]. Since the user strings are a zero-based array, one can iterate over them directly. The condition that iterates over each entry in this array and applies the regular expression to each entry in the array is the following.

```
for any str in dotnet.user_strings : (
  str matches /^4\x00[dD]\x005\x00[aA]\x00.{186,}/
)
```

Putting this all together yields the YARA rule shown in Figure 3. This rule is provided at the end of the blog.

```
import "dotnet"

rule DotNet_EmbeddedPE
{
    meta:
        author = "Malware Utkonos"
        date = "2021-01-18"
        description = "This detects a PE embedded in a .NET executable."
    condition:
        for any str in dotnet.user_strings : ( str matches /^4\x00[dD]\x005\x00[aA]\x00.{186,}/ )
}
```

Figure 3: YARA Rule to Match Files with an Embedded PE Executable

If one is using an older version of YARA, the following syntax achieves the same results.

```
  for any i in (0..dotnet.number_of_user_strings-1) : (
    dotnet.user_strings[i] matches
/^4\x00[dD]\x005\x00[aA]\x00.{186,}/
  )
```

Running this rule as a retro hunt in the Titanium Platform results in thousands of files that are detected as malicious or suspicious with very few files that are undetected. The hunting results are shown in Figure 4.

Figure 4: YARA Retro Hunting Results

One question to ask when presented with good results like this is whether this technique is used in legitimate software. According to a question of how to include a windows DLL in a .NET project posted on StackOverflow, this data should be packaged as a resource[10]. The technique analyzed in these samples is definitely non-standard.

Static Extraction

The next step is to extract all the embedded payloads from all these files and see what is there. To do this, a fascinating feature of the yara-python package is used: the modules_callback parameter of a YARA rule match object [11]. The beauty of this parameter is that it allows a function with whatever code one wants to run to be executed if a rule matches. Inside that function, the data returned from a YARA module is made available to the function in the form of a data dictionary. The specific callback function used to extract many of the payloads from the hunting results dataset is shown in Figure 5.

```
def modules_callback(data):
    for user_string in data['user_strings']:
        if re.match(b'4\x00[dD]\x005\x00[aA]\x00', user_string):
            try:
                binary = bytes.fromhex(user_string[::2].decode())
            except ValueError:
                return yara.CALLBACK_CONTINUE
            sha256 = hashlib.sha256(binary).hexdigest()
            with open(working.joinpath(sha256), 'wb') as fh:
                fh.write(binary)
            return yara.CALLBACK_CONTINUE

    return yara.CALLBACK_CONTINUE
```

Figure 5: Callback Function to Extract Payloads

This callback function extracts the very most basic form of encoding encountered in these files. There are additional obfuscation techniques observed in the dataset where additional character replacements are required to recover the payload in its original form. These additional techniques will be examined in a future blog post, but many of the resulting extractions from these techniques are included in the IOC data provided below.

Correcting the Record

After as many files as can be extracted with the python callback function as possible have been analyzed along with their parent loaders, the resulting analysis dataset is loaded into an Elasticsearch instance for easier analysis. Sorting the parent loaders by the threat level reveals a few files with a zero score. The example shown in Figure 6 is a really old file, so the detection may be stale.

Time	Malware Type	Cluster ID	ReversingLabs.Threat Name	ReversingLabs.Threat Level
Jul 3, 2011 @ 09:04:00.000	Loader	12a471e9-5acf-4602-a521-381b75509201	-	0

Expanded document

Table JSON

t Cluster ID	12a471e9-5acf-4602-a521-381b75509201
Compile Timestamp	Aug 4, 2008 @ 22:20:22.000
Corrupted	false
# Exiftool.ExifToolVersion	12.16
t Exiftool.FileType	Win32 EXE
t Exiftool.FileTypeExtension	exe
t Exiftool.MIMEType	application/octet-stream
# File Size	215,552
t File Type	Win32 EXE
t Filename	b669280fe2496b5aaa72e6de2fae2191b212c237f14e4052236ec2ae0ed06fba
First Seen	Jul 3, 2011 @ 09:04:00.000
t Import Hash	f34d5f2d4577ed6d9ceec516c1f5a744
# Imports.mscoree.dll	1
t MD5	7b5d8a9c36bdf6deaceec88956267fb2
t MIME Type	application/x-dosexec
t Magic	PE32 executable (GUI) Intel 80386 Mono/.Net assembly, for MS Windows
t Malware Type	Loader
t ReversingLabs.Classification Reason	Cloud
t ReversingLabs.Report Link	https://a1000.reversinglabs.com/639bcc9b9ec6a9c480f9531aead93430f7c7122c
# ReversingLabs.Threat Level	0
t ReversingLabs.Threat Name	-
t ReversingLabs.Threat Status	Known
# ReversingLabs.Trust Factor	2

Figure 6: Zero Threat Level on Very Old File

Running this file in a sandbox shows that it is some type of hacking tool and definitely something that one would consider at least suspicious if not malicious. A screenshot of this sandbox session is shown in Figure 7.

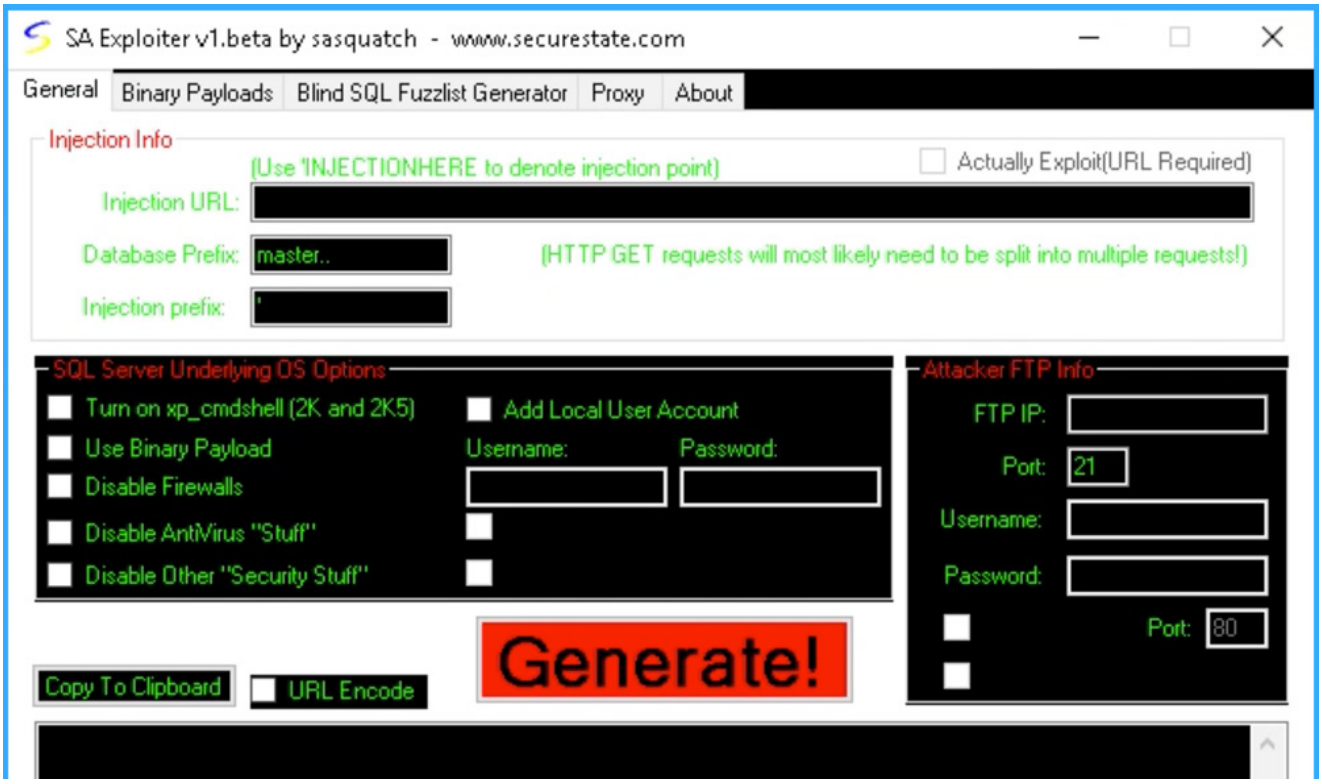


Figure 7: Hacking Tool User Interface in Sandbox

Therefore, the Titanium Platform provides a way to correct the record and freshen an old analysis result such as this. To do this, open the file in the A1000, and click the "Reanalyze" button. This button is shown in Figure 8. As can be seen also in Figure 8, the threat level is now shown to be 2 rather than zero.

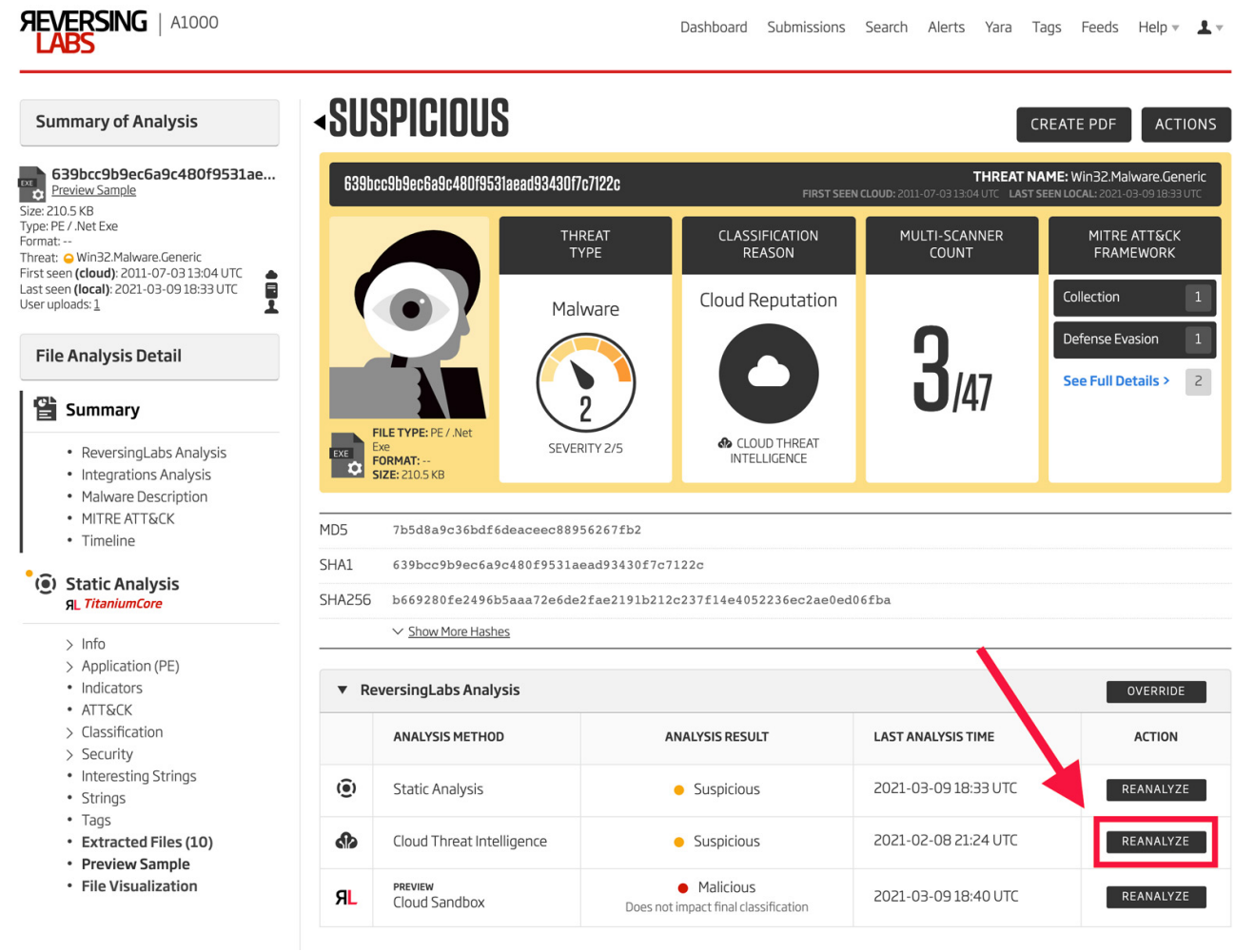


Figure 8: Reanalyze Sample

General Analysis

As mentioned above, there are a few files that use more complex obfuscation techniques in addition to the embedding analyzed here. Even with these files excluded, a group of 1,641 loaders and their corresponding payloads are identified. A few files from this dataset along with the timeframe of the results from 2011 to now is shown in Figure 9.

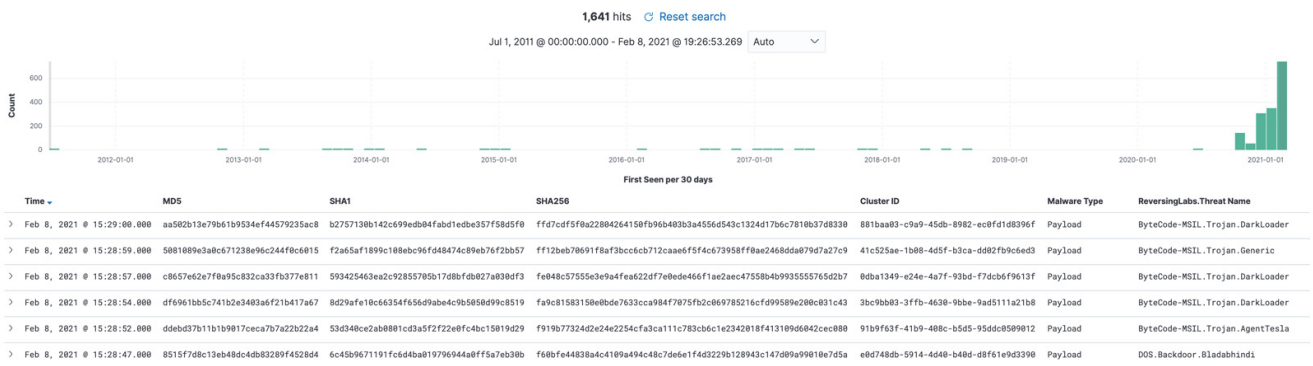


Figure 9: Extraction Results

The cluster IDs shown above are assigned per unique payload to the payload and each of the unique parent files that have the same embedded payload. These cluster IDs are randomly generated UUIDs per payload. The full dataset including the cluster data and threat names is provided at the end of the blog. The breakdown of the embedded files by file type according to libmagic is shown in Figure 10 with the files fairly evenly split between DLLs and EXEs.

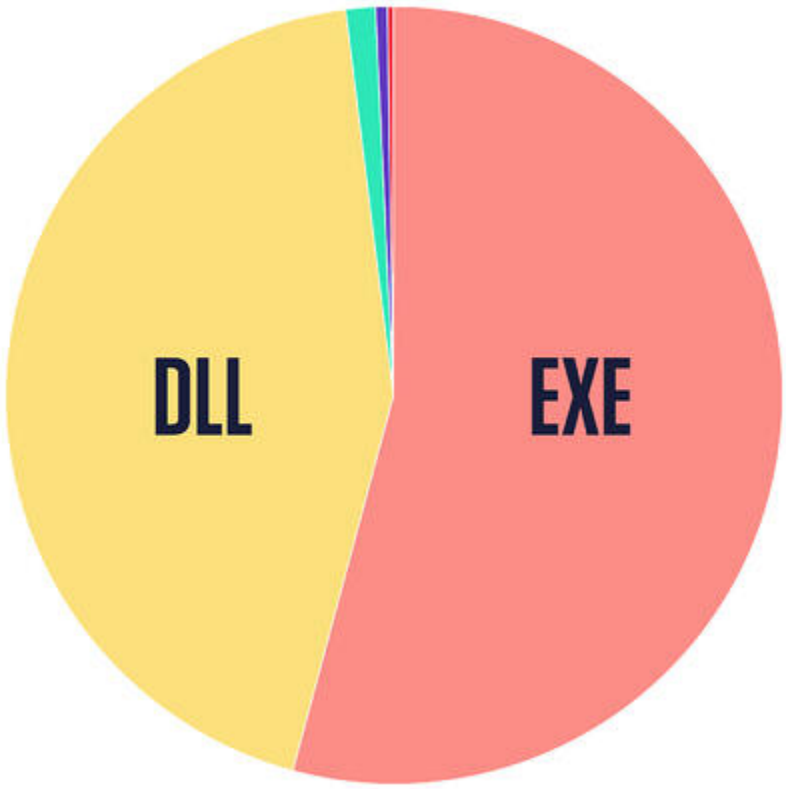


Figure 10: Libmagic Analysis of Embedded Payloads

Conclusion

The technique of encoding an executable as a text string using various encoding techniques is widely used by the adversaries who post the resulting strings on pastebin-like sites. It is interesting that some of the same string encoding techniques used by these adversaries to hide payloads on download sites across the internet are also found inside the delivery binaries. This obfuscation can definitely provide some cover from detection. To identify binaries that leverage this technique, YARA's dotnet module provides the best methods available. What was shown above is just the most simple and basic encoding technique. Further research into other encodings such as Base64 will be the topic of future blog posts.

YARA Rule

```
import "dotnet"
```

```
rule DotNet_EmbeddedPE
{
  meta:
    author = "Malware Utkonos"
    date = "2021-01-18"
    description = "This detects a PE embedded in a .NET executable."
  condition:
    for any str in dotnet.user_strings : ( str matches
/^4\x00[dD]\x005\x00[aA]\x00.{186,}/ )

}
```

References:

1. https://blog.reversinglabs.com/blog/rats-in-the-library
2. https://twitter.com/ScumBots
3. https://yara.readthedocs.io/en/stable/modules/dotnet.html#c.user_strings
4. 0efe600018208dd66107727362dd8f7498813755ce14f76fa19f0964c654e14a
5. https://malpedia.caad.fkie.fraunhofer.de/details/win.revenge_rat
6. https://github.com/dnSpy/dnSpy
7. https://yara.readthedocs.io/en/stable/commandline.html#cmdoption-yara-d
8. https://webserver2.tecgraf.puc-rio.br/~ismael/Cursos/YC++/apostilas/win32_xcoff_pe/tyne-example/Tiny%20PE.htm
9. https://yara.readthedocs.io/en/stable/writingrules.html#iterators
10. https://stackoverflow.com/questions/72264/how-can-a-c-windows-dll-be-merged-into-a-c-sharp-application-exe
11. https://yara.readthedocs.io/en/stable/yarapython.html#yara.Rules

MORE BLOG ARTICLES