Agent Tesla - Building an effective decryptor

By map[name:Alessandro Strino]

Published: 2023-08-29 · Archived: 2026-04-05 17:58:53 UTC

General Information and preface

Agent Tesla, according to the data provided by CERT-EU, is one the most prominent threats that are hitting European cyberspace. Because of that, I found that it could be quite interesting understanding its behavior. However, as I usually prefer, instead of focusing on the infection chain or information about TA methodologies, sharing knowledge that were already provided by more authoritative sources, I preferred to focus mostly on malware analysis and in this specific case to the latest encryption algorithm adopted. Then, upon the knowledge acquired, I would like to write a decryptor script that extracts payload configuration.

Encryption variants

Agent Tesla first appeared in 2014, however, its evolution over time could be tracked by multiple TTPs. For the purpose of this article, I'm versioning it through the encryption algorithm adopted. At the time of writing there have been 4 different versions with unique characteristics:

Encryption v1: Through this encryption implementation, strings are stored (encrypted) in base64. The decryption function uses a password and a salt (both hardcoded) as input for the SHA1 algorithm, in order to generate the decryption key. Then, ciphertext and key are eventually used with AES in CBC mode.

Encryption v2: The main difference from the previous method, is that each encrypted string is paired with a dedicated key and an IV. The algorithm used is still AES in CBC mode.

Encryption v3: In this version, TA completely changed their approach, shifting to a pure *xor decryption*. The decryption function is defined within the *.cctor()* constructor. The structure of encrypted strings is quite simple. Each string is contained in a byte array paired with a key. The size of the ciphertext allows the decryption routine to iterate over the byte array distinguishing all parameters.

Encryption v4: The latest version of Agent Tesla is based on a *xor string* algorithm that stores information within a macro-structure that contains raw data organized as follow:

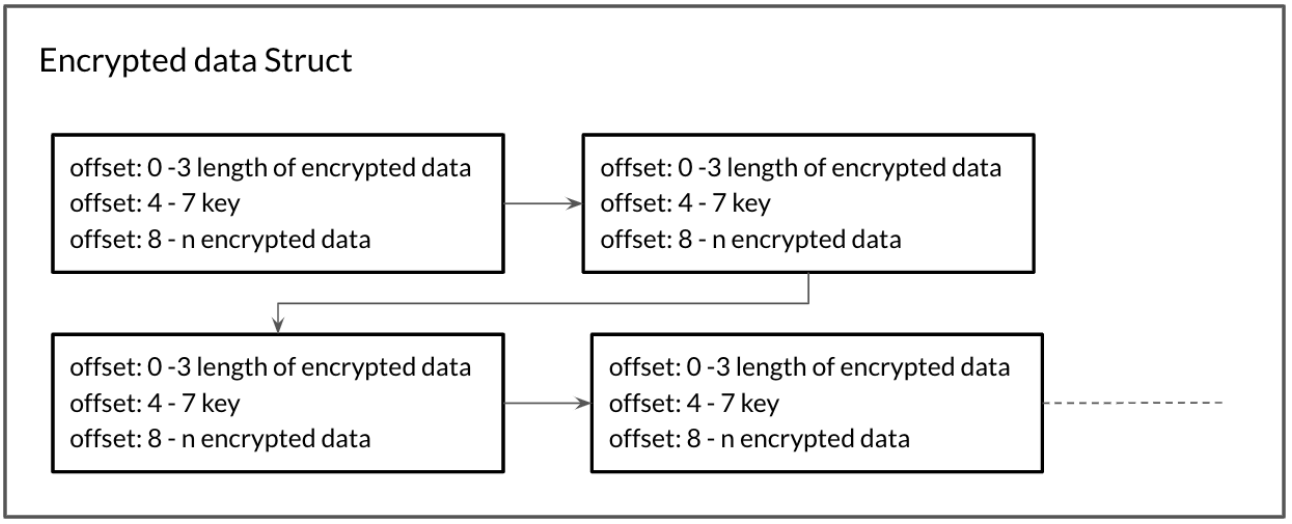

Figure 1 - Encrypted data structure

In this version we have a macro-struct that contains the encrypted data. It's actually possible to visualize it as an array where each element follows a specific structure where the first 4 bytes are dedicated to define the encryption data length, then other 4 bytes are used to describe the encryption key and the remaining bytes are reserved for the actual data.

As always, the decryption routine iterates over the encrypted data, using key bytes.

Analysis of encryption v4 and obfuscation routine

As I wrote in the first paragraph Agent Tesla analysis will be carried out on the latest encryption mechanism. Because of that, let's start analyzing the main components of this algorithm:

The key is generated randomly through **RandomNumberGenerator.GetInt32(int.MaxValue)** function. The *int.MaxValue* constraint is related to the key size limitation of 4 bytes within the encrypted data structure. The algorithm it's pretty straightforward, performing *xor* between plaintext and key bytes. However, what really matters here is the **obfuscator** that happens at runtime.

The **obfuscation routine** is part of an open source project. Basically, this obfuscation works creating multiple placeholders that are going to be replaced at runtime. Analyzing the sample statically, this technique messes up with code decompilers such as DnSpy. However, the author wrote a detailed blogpost explaining obfuscator features and its modus operandi:

> "After processing all methods we need to do some patches in the injected runtime. First, we need to set up the placeholder struct with the correct attribute values. The struct needs a **ClassLayout** with **packing size 1** and the **length of the encrypted data as its size**."

> "We also need to create a new **field which will be an initialized version of our struct**. By adding a DataSegment in its **FieldRva**, we can use the field to store any raw data we want, in this case, our encrypted string data."

Following the indication provided by the author, we should have a more clear idea of what these components are and how to hunt for them in the code. Now it's possible to start looking at the code gathering as much information as possible (e.g., locating these placeholders) and finding out a pattern to write an effective configuration decryptor.

Binary inspection

If we open up DnSpy, we could be overwhelmed by the mess that is going to be presented in front of us. However, trying to follow the code flow, we could start exploring.

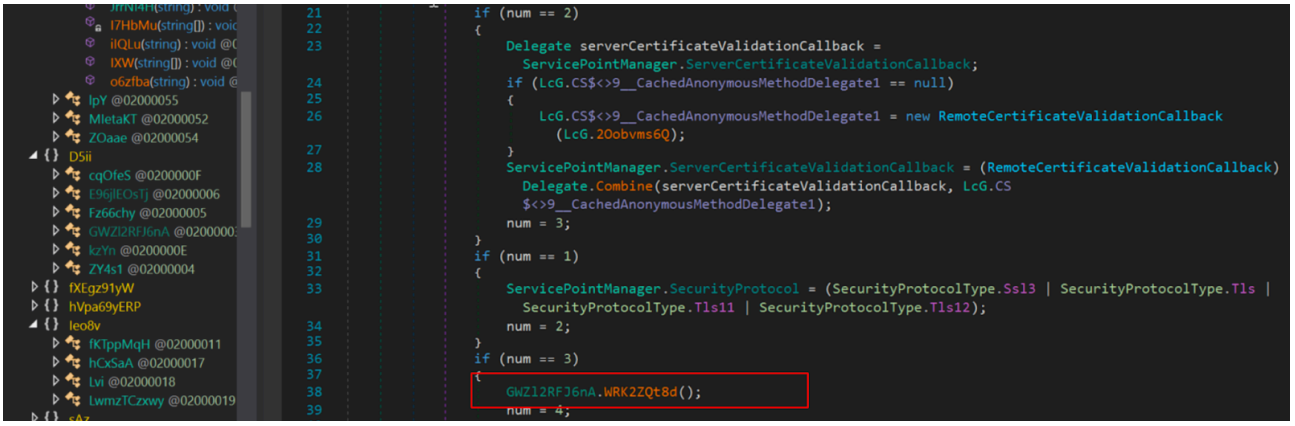

Figure 2 - Entry point

Starting from the Entry Point, it's possible to locate something promising. Exploring variable and function calls and following references to the object **GWZl2RFJ6nA**, it's possible to bump into quite interesting piece of code.

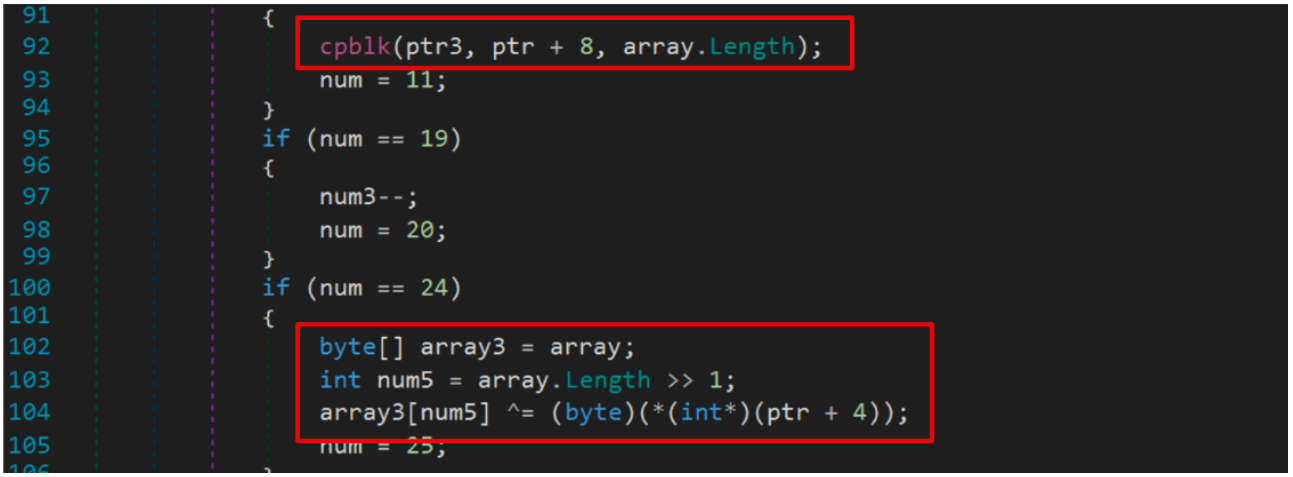

Figure 3 - Obfuscated decryption routine

First of all, we see the function **cpblk** that it's necessary to perform the injection at runtime, moreover, scrolling the code it's possible to get insight about pointer operations paired with xor. In fact, observing the line 104 in Figure 3, it is very similar to the encryption routine that we saw in the previous paragraph. Nevertheless, on the top of that, as a final proof that we are looking in the right place, scrolling at the end of the code we see a class that fulfills all requirements requested by the obfuscator.

- **ClassLayout**
- **Pack size = 1**
- **Size = encrypted data length**

```
208
209     // Token: 0x02000093 RID: 147
210     [StructLayout(LayoutKind.Explicit, Pack = 1, Size = 19335)]
211     private struct zqRrwrwgu
212     {
213     }
214 }
```

Figure 4 - Class placeholder for runtime decryption

Moreover, following the instruction provided by the author, we should be able to locate the encrypted string in raw bytes within the binary. If we have a closer look to struct **zqRrwrwgu** examining the raw value, we are redirected to a very suspicious sequence of bytes.

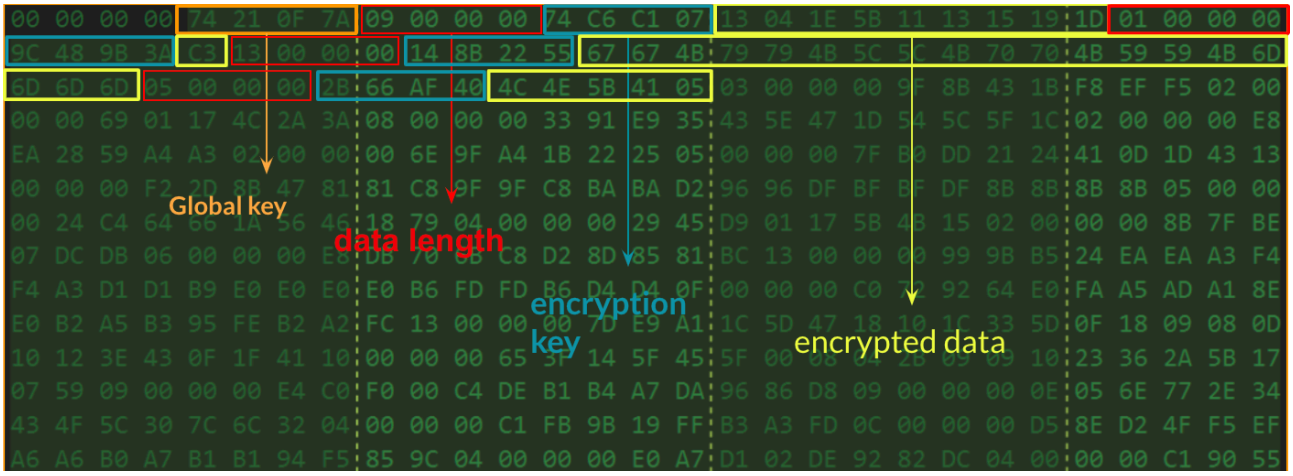

Figure 5 - Encrypted string

Analyzing those bytes, it's immediate to find out that we are dealing with an encrypted payload that is ready to be decrypted.

Decrypting Strings

The idea behind this script is simple:

- Find out a class that is big enough to contain the encrypted data (usually the biggest class in the code). Then we could retrieve raw bytes related to its size from the binary file and forward them towards our decryption routine

Before proceeding, it's worth mentioning that I'm quite a newbie in .NET interaction with python and generally, I'm still learning .NET layout. Because of that, if someone else is going to produce more efficient code. Please do it! But for now, let's do a quick look to this this script:

```
31  def main():
32      if len(sys.argv) < 2:
33          print("Error in argument file")
34          sys.exit(1)
35      file_path = sys.argv[1]
36      d_file = DotNetPE(file_path)
37      md_tables = d_file.metadata_tables
38
39      class_sizes = []                                    40 - 44
40      for m in md_tables:
41          if 'ClassLayout' in m.string_representation:
42              class_layout_rows = m.table_rows
43          if 'FieldRVA' in m.string_representation:
44              field_rva_rows = m.table_rows
45                                                          46 - 47
46      for i in range(0, len(class_layout_rows)):
47              class_sizes.append(class_layout_rows[i].ClassSize.value)
48
49      encrypted_data_size = max(class_sizes)
50                                                                  51
51      physical_address = d_file.get_physical_by_rva(field_rva_rows[-1].RVA.value)
52      data = d_file.get_data(physical_address, encrypted_data_size)
53      print(decrypt(data))
```

Figure 6 - Agent Tesla decryptor

- **Line 40 - 44:** Gathered references to ClassLayout and FieldRVA Tables.
- **Line 46 - 47:** Following the initial idea that encrypted data should be stored in a quite large class, I started to enumerate class, selecting the biggest (likely the one that contains the encrypted strings)
- **Line 51:** retrieve the last element of the FieldRVA table that contains our data (I don't know if it is an unexpected behavior caused by the obfuscation process applied, but the raw data containing the encryption string resulted to be always the last element of the FieldRVA table).

Running our script on one of the latest Agent Tesla sample, we got the following result:

```
ackageSid', b'pAuthenticatorElement', b'IE/Edge', b'UC Browser', b'UCBrowser\\', b'*', b'Login Data', b'journal', b'wow_
logins', b'Safari for Windows', b'\\Common Files\\Apple\\Apple Application Support\\plutil.exe', b'\\Apple Computer\\Pre
ferences\\keychain.plist', b'<array>', b'<dict>', b'<string>', b'</string>', b'<string>', b'</string>', b'<data>', b'</d
ata>', b' -convert xml1 -s -o "', b'\\fixed_keychain.xml" ', b'"', b'"', b'\\Microsoft\\Credentials\\', b'\\Microsoft\\C
redentials\\', b'\\Microsoft\\Credentials\\', b'\\Microsoft\\Credentials\\', b'\\Microsoft\\Protect\\', b'\\', b'credent
ial', b'QQ Browser', b'Tencent\\QQBrowser\\User Data', b'\\Default\\EncryptedStorage', b'Profile', b'\\EncryptedStorage'
, b'entries', b'category', b'Password', b'str3', b'str2', b'blob0', b'password_value', b'IncrediMail', b'PopPassword', b
'SmtpPassword', b'Software\\IncrediMail\\Identities\\', b'\\Accounts_New', b'PopPassword', b'SmtpPassword', b'SmtpServer
', b'EmailAddress', b'Eudora', b'Software\\Qualcomm\\Eudora\\CommandLine\\', b'current', b'Settings', b'SavePasswordText
', b'Settings', b'ReturnAddress', b'-', b'Falkon Browser', b'\\falkon\\profiles\\', b'profiles.ini', b'startProfile=([A-
z0-9\\/\\.\\"]+)', b'profiles.ini', b'\\browsedata.db', b'autofill', b'ClawsMail', b'\\Claws-mail', b'\\clawsrc', b'\\cl
awsrc', b'passkey0', b'master_passphrase_salt=(.+)', b'master_passphrase_pbkdf2_rounds=(.+)', b'\\accountrc', b'smtp_ser
```

Figure 7 - Decryptor result

References

Agent Tesla Decryptor:

- agent_tesla_decryptor_v4.py

DotNet references:

- General .NET implementation info
- Python parser for .NET

Samples:

- ae26382f191225447550e9a691453fc3ea2e02127222787c662efc8db63c59e3 (SHA256) MalwareBazaar
- acedd97c8350bacc485e87ae56c851d08b497c202deb46df28c3e7218cb4469a (SHA256) MalwareBazaar
- f5751d89bc6f15c3ade6513d3cf44f92a25e8cd25d2f5ad239b8c44f8f732cb8 (SHA256)MalwareBazaar

---

Source: https://viuleeenz.github.io/posts/2023/08/agent-tesla-building-an-effective-decryptor/