IN-DEPTH ANALYSIS OF THE EMOTET PACKER

2018-12-10

AUTHOR: d00rt (@D00RT_RM)

INDEX

This investigation is independent of any company, and was done during the author's free time. Any mistake or contribution is welcome.

- *Email (d00rt.fake@gmail.com)*
- *Twitter (@D00RT_RM).*

It could be the case that at some time the terminology used is not the correct one, in case of detecting this type of failures, it would be appreciated if they were reported.

In addition to this document, the repository (https://github.com/d00rt/emotet_research) also provides various tools that aid in the analysis and detection of this family of malware as well as its infrastructure.

Tools:

- *An unpacker of Emotet's final payload.*
- *The unpacker itself is able to extract the C&C list as well as, the RSA key it uses for communication with the C&C, which could be identified as botnet.*
- *YARA rules that apart from being used for configuration extraction, could also be used to detect the presence of this malware on a computer.*
- *A final payload binary next to a documented IDC. (Although not completely documented but almost).*

Thanks to @D00m3dR4v3n for helping with some tests. To @n4r1B, PabloForThePPL and @ulexec for giving me feedback during this research.

Enjoy!

SCOPE

This document and the unpacker are based on samples dated between 2018-10-1 and the date of publication of this document.

The existence of any information about this packer or its possible versions is not known, so it is not assured that it works 100% for all samples, but it is assured that the unpacking process is identical for all Emotet, as described in the document.

**** UPDATE ****

Looking for information about Emotet's packer before the publication of the article, it has been discovered that the researcher **@herrcore** has a video on YouTube, explaining how to unpack Layer 1 for a particular sample (**90c2c10001134ab2a1cc87ec4382b197**). The video date is 2018-02-18, and the sample it analyzes was first seen in VT at 2018-01-17. Trying to test the unpacker on that sample, an error has been located in the unpacker.

The failure was that the Yara that matches with the final Payload code, did not match, so it did not finish the process of unpacking. However, the unpacking process worked correctly. This is because the version of Emotet that is protected by the packer was a different version to those currently distributed, so you can confirm that Emotet developers are evolving the code frequently, since in less than 1 year the code has changed completely.

This bug is currently fixed in the unpacker and it can almost be guaranteed that the unpacker is valid for all Emotet samples in 2018, but it is not guaranteed that some kind of problem of this type could happen.

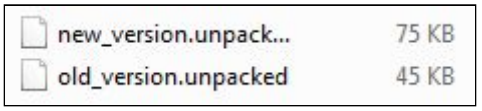

Image 1: Size of the different versions of the Payload once unpacked.

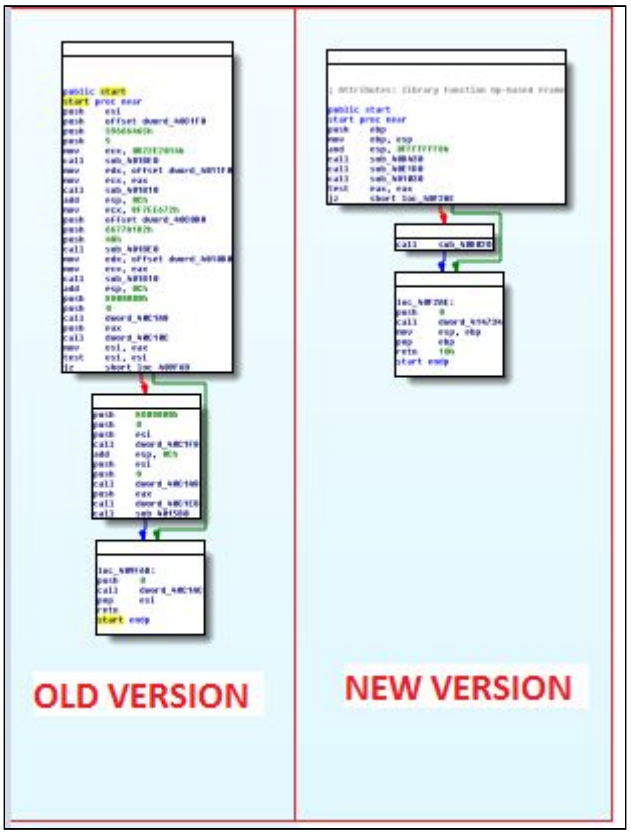

Image 2: Payload Entry Point (EP) - unpacked.

1. INTRODUCTION

Emotet is a family of malware that has been very active for a few years. New campaigns and new binaries of this malware appear almost every day.

Emotet is used as a downloader of other malwares, generally bank type malware such as *PandaBanker*, *Trickbot* and *IceID*.

Although in more than one report about this family it is said that Emotet is a banker, during this investigation no evidence has been found that Emotet has the functionality of stealing bank credentials.

This document explains in depth how the software that protects the code of Emotet malware works. Although this document is not intended to be a tutorial on how to unpack this malware, with the documentation provided, it should be more than enough to do it manually. Anyway in this same GitHub repository you can find a tool that automatically unpacks this malware. (*https://github.com/d00rt/emotet_research/unpacker*).

The main motivation of this research was to try to make a static unpacker, but due to the techniques used by the software that protects Emotet, which will be explained later, and the lack of time, has opted for a more viable solution such as a dynamic unpacker using the framework of *TitanEngine*.

Once the payload is extracted, the unpacker will also take care of extracting the static configuration from the unpacked binary. This configuration consists of 2 elements, a list with the C&C's and the RSA key used to communicate with the C&C.

The incessant activity of this family of malware and analyzing both its methods of protecting it ("custom packer") and distributing it, one can get an idea that the people behind this malware know very well what they are doing and at a technical level are very advanced.

After the publication of this article, it is very likely that this packer has evolved or completely changed. So this analysis is only valid for samples dated prior to the publication of this article. (At least for the last few months).

This research has been done during the author's free time and is not intended to be a definitive guide, however it is intended to help the research community to better understand this malware and finally try to put an end to it.

2. GENERAL DETAILS

The following image shows how the process of infection and unpacking of the Emotet is summarized.

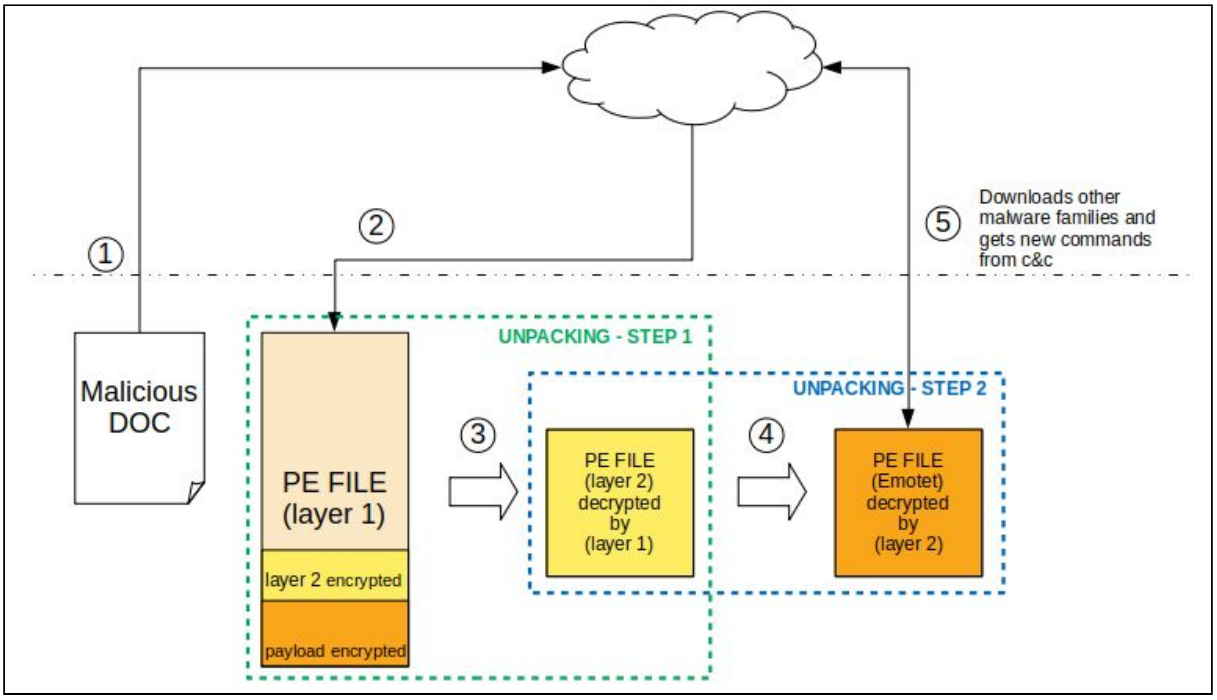

Image 3: Emotet infection and unpacking process.

As you can see in the image, it's a 2-layer packer. The first layer (layer 1) is the only one that touches the disk and maps in memory the second layer (layer 2) that will finally execute the final payload (Emotet).

The unpacking process involves 3 binaries, which are executed in sequential order within the infected machine and which from now on will be referred to as follows:

Layer 1	The binary that is downloaded by the malicious .docx. In charge of triggering the entire unpacking process. Decrypts and maps Layer 2 in memory.
Layer 2	The binary in charge of deciphering, fixing and mapping in memory the final payload. This binary is located inside the Layer 1 binary.
Payload - Emotet	This binary is the one that does all the malicious activity inside the infected system, that is, the Emotet. This binary is broken in the first instance, i.e. it cannot be executed directly without first being fixed by the Layer 2 binary.

Table 1: Description of the binaries involved in the unpacking process.

Layer 1 hides an encrypted buffer, divided into small chunks. The binary itself reconstructs these chunks using a hard-coded array. Then it writes it into memory.

A large block of the buffer is decrypted, exposing a "PE" binary which is mapped in memory and immediately passes the execution flow.

Layer 2 deciphers the remaining part of the buffer written by layer 1. On this occasion what is deciphered is another binary, the one that corresponds to the payload. As already mentioned, this binary cannot be executed directly, since it needs to be previously modified by layer 2. This whole process is explained later on.

Finally, the execution flow is passed to the payload and this is where all malicious activity begins.

3. UNPACKING PROCESS

The unpacking process is explained in detail below, step by step.

The following images represent the state of the process memory at different stages of the unpacking process. The red box on the binaries mapped in memory indicates that at that moment the execution flow is in that binary.

3.1 STEP 1

In this step Layer 1 will decipher Layer 2 in memory and immediately pass control to it.

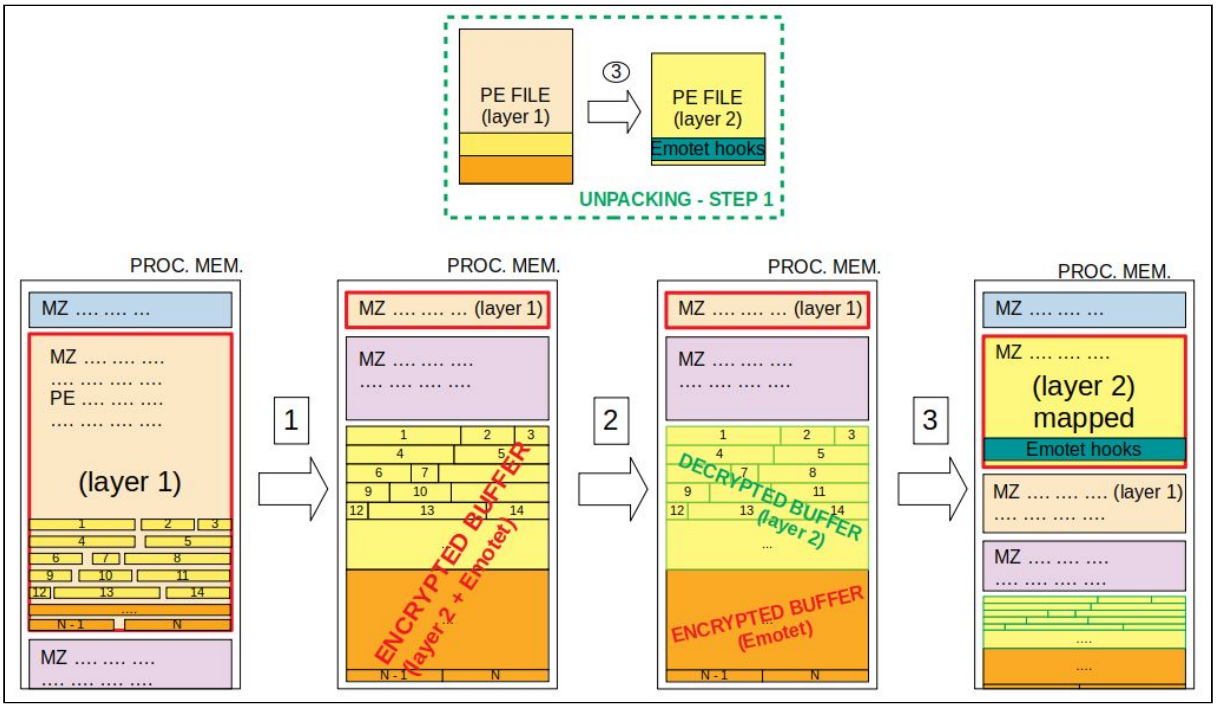

Image 4: Passing the execution from Layer 1 to Layer 2. STEP 1

The following describes the steps that are followed to obtain layer 2.

1. Layer 1 has an encrypted buffer. This buffer is not a sequence of consecutive bytes, but is divided into small chunks. Based on an array that has Layer 1, these chunks are built and written in memory.

   This array is found in one of the sections of the binary, but the indices it uses and the positions in which these chunks are found are randomized among the different Layer 1 samples that exist, so finding a pattern to do this statically is not trivial.

2. Part of this buffer is decrypted. Leaving layer 2 exposed.
3. Layer 2 is mapped in memory and then the execution flow is passed to it. In other words, layer 1 jumps to the Entry Point of layer 2.

Some of the layer 2 samples, if they are dumped and tried to open with a debugger, seem to be broken... this is because the binary on disk has a file alignment of 0x10, nor is Windows itself able to run it. So this is something that is also contemplated in the unpacking tool provided with this document. Sections are aligned to 0x200 by default. This is most likely done on purpose, as an anti-analysis measure.

3.2 STEP 2

Step two is somewhat more complex than the previous one. In this case layer 2 decrypts the other part of the buffer that left layer 1 undeciphered. What you get in this case is the Payload, in other words, the Emotet. But this binary is not functional. This binary is missing some OPCODES, which instead were replaced by the OPCODE "CC".

The original OPCODES that should be in the payload are hidden by Layer 2. Layer 2 writes these OPCODES in another memory area. While in the payload the "CC" are replaced by the necessary instructions to jump to the "hijacked" OPCODES (trampoline). In other words, in the mapped payload a trampoline is written to jump to the code that has been "hijacked" and that should be in the payload.

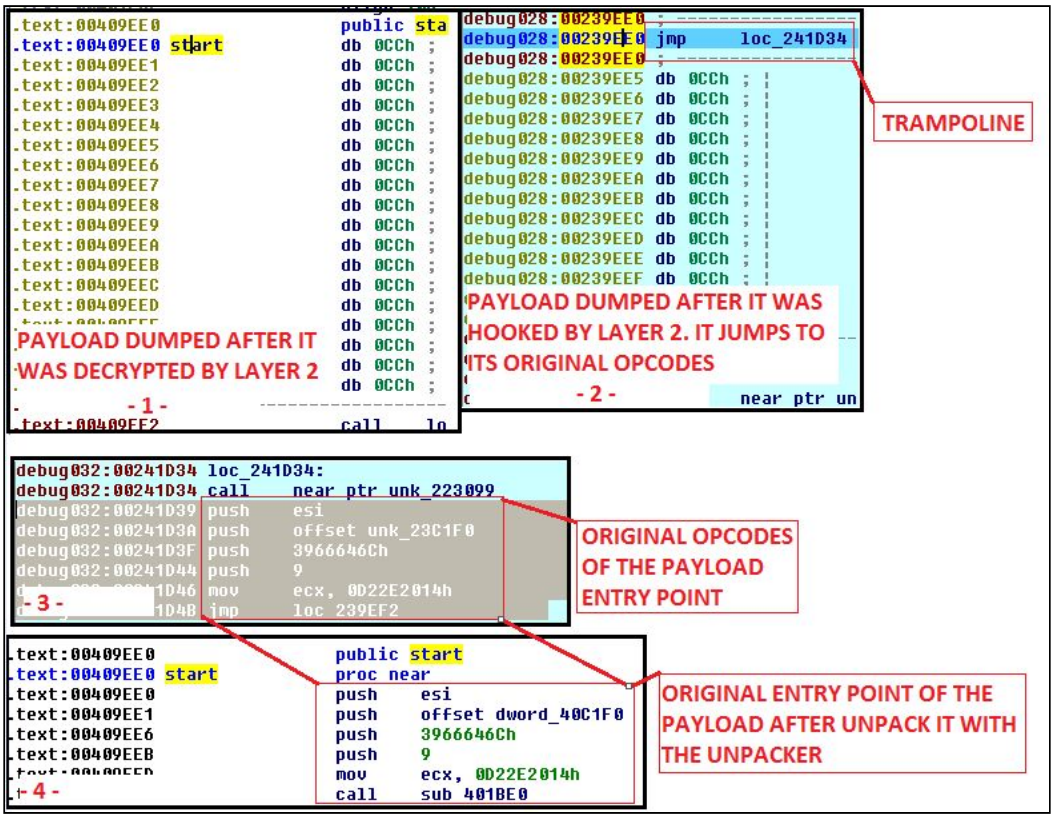

Image 5: This image shows the OEP Payload after being decrypted by Layer 2 (1), the Original Entry Point (OEP) of the Payload after being hooked (2), the original code that should be in the Original Entry Point (OEP) of the Payload (3) and the Payload fixed by the unpacker that is provided in the repository.

So finally, the payload execution flow passes through 3 different memory segments each time a function is called.

Using this technique achieves 2 very interesting things, the first of all is that thanks to the payload is modified, whether the payload is dumped before or after being mapped, you can not get any way to make it run, This makes the task of analysis completely difficult.

On the other hand, as the functions are hooked, the flow of execution is not the same as it would be in normal case and this can bring more headaches to the analyst, even the dissasembler could go crazy.

Payload → Emotet hooks → Layer 2 → Payload

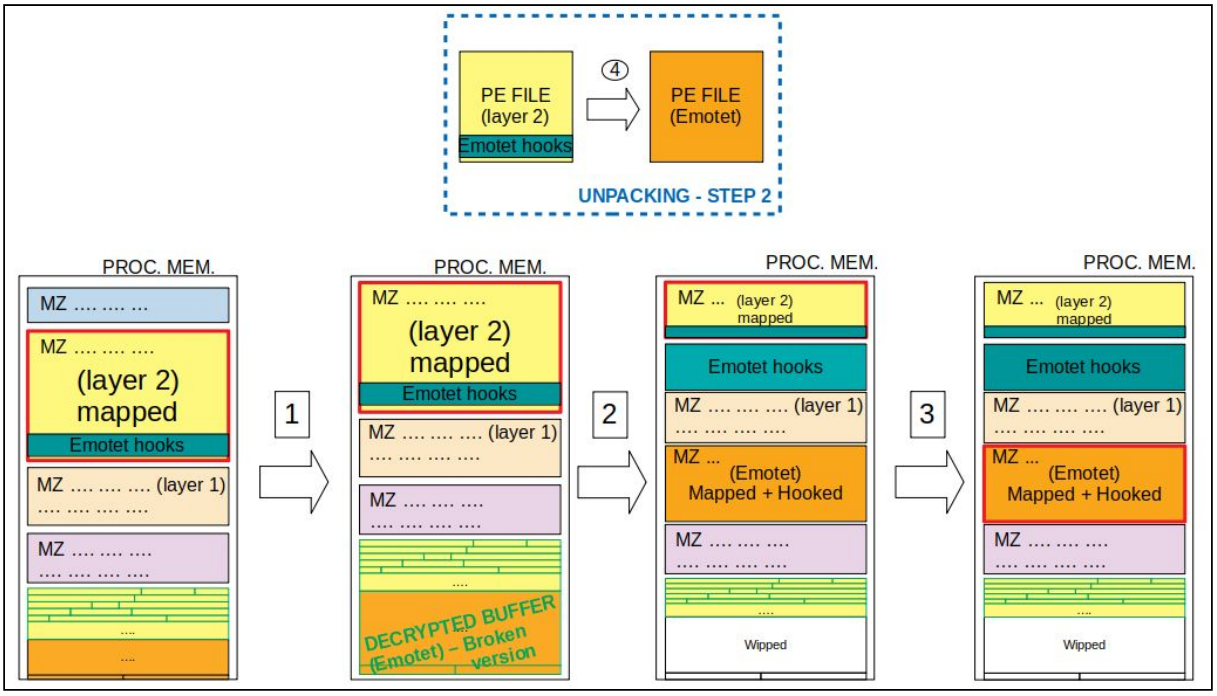

Image 6: Passing the execution from Layer 2 to Payload. STEP 2

These are the steps that Layer 2 follows to get and run the Payload.

1. Layer 2 deciphers what's left of the encrypted buffer. In this case, the binary that is decrypted is the payload (Emotet). Most of the binaries of type Layer 2 that have been analyzed, in their section ".data" they have a pointer to the key of decryption of the buffer as well as an array with the indexes of the order in which this buffer has to be decrypted.

   If the Payload is dumped once it has been decrypted, Windows cannot execute it. Analyzing the code, it is replete with "CC CC CC ..." where there really should be code.

2. Layer 2 maps the broken payload in memory. Layer 2 has another array which indicates the offset where the kidnapped OPCODES are, the offset from where they were extracted in the broken Payload and the number of OPCODES to be copied.

   Following this array, the OPCODES that should be in the broken payload are dropped in a different memory area and the “CC” OPCODES are overwritten to jump to that memory area. In other words, it is as if you were making a hook.

   Apart from the original code, the hook also adds a call to a function that belongs to Layer 2. This function seems to increase a counter each time a hooked function is called.

   For this Packer research, not much more relevance has been given to this function.

Finally, in this step, the buffer where the payload was decrypted is also wiped to prevent it from being dumped.

3. Once the payload has been rebuilt, the execution flow passes to it and here begins what is known as Emotet.

4. HOOK EXECUTION FLOW

In the image below you can see the state of the memory just when the Payload starts to run.

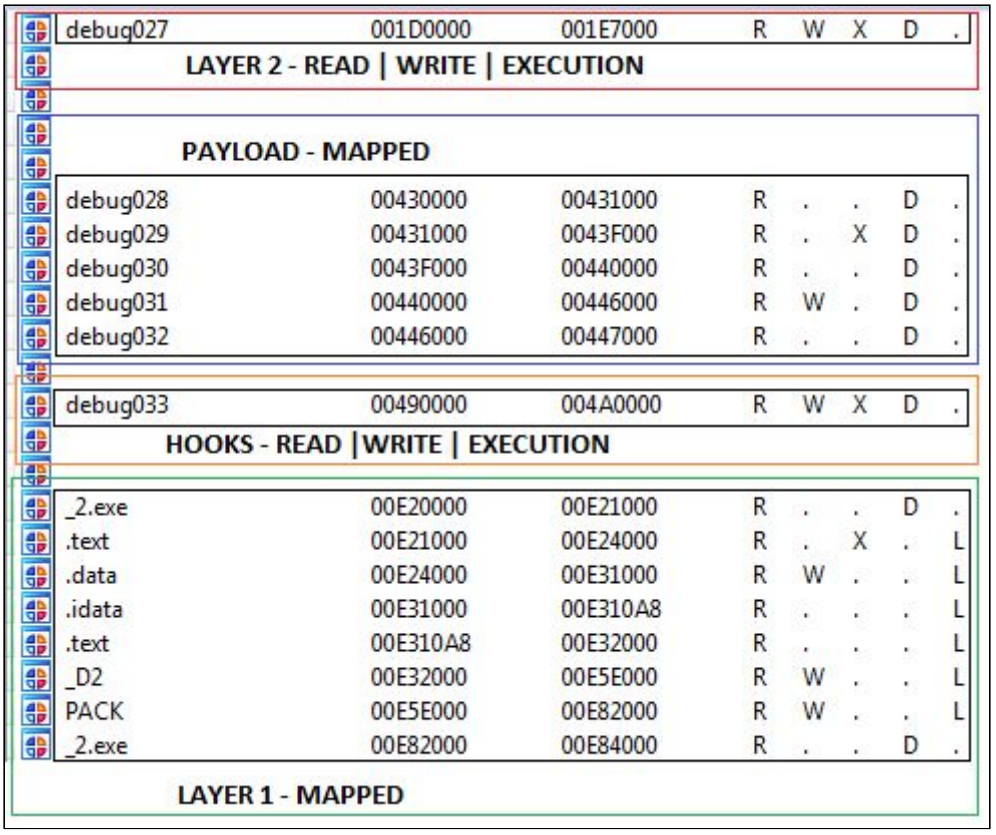

Image 7: Memory status when the payload is running.

BASE ADDRESS	MEMORY SEGMENT
0xE20000	LAYER 1
0x1D0000	LAYER 2
0x430000	PAYLOAD
0x490000	HOOKS

Table 2: List of memory segments.

The flow that follows each time you call a function is the one shown in the image below.

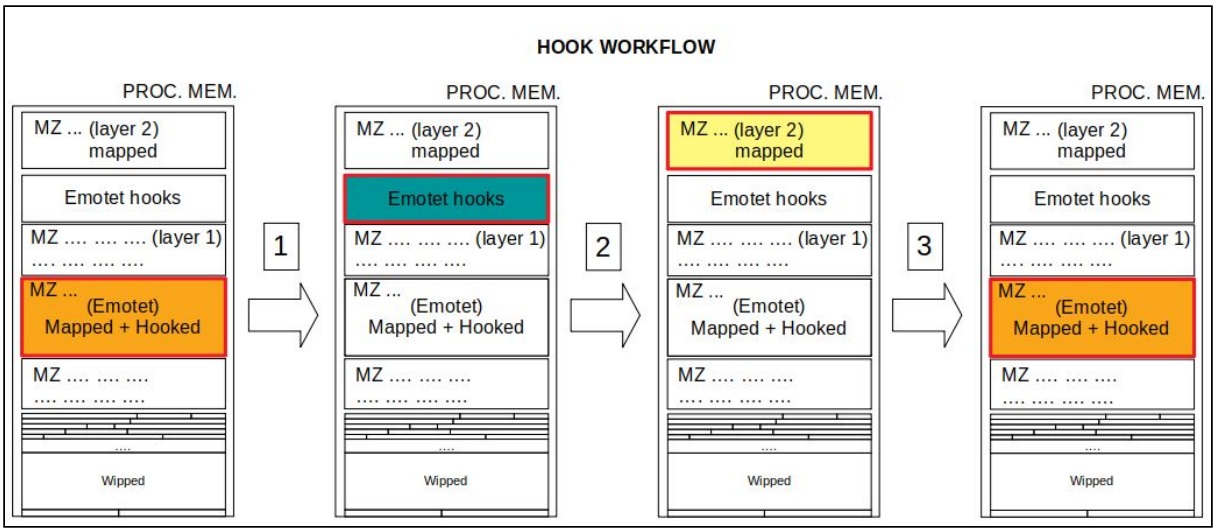

Image 8: Flow of execution of a hooked call of the Emotet

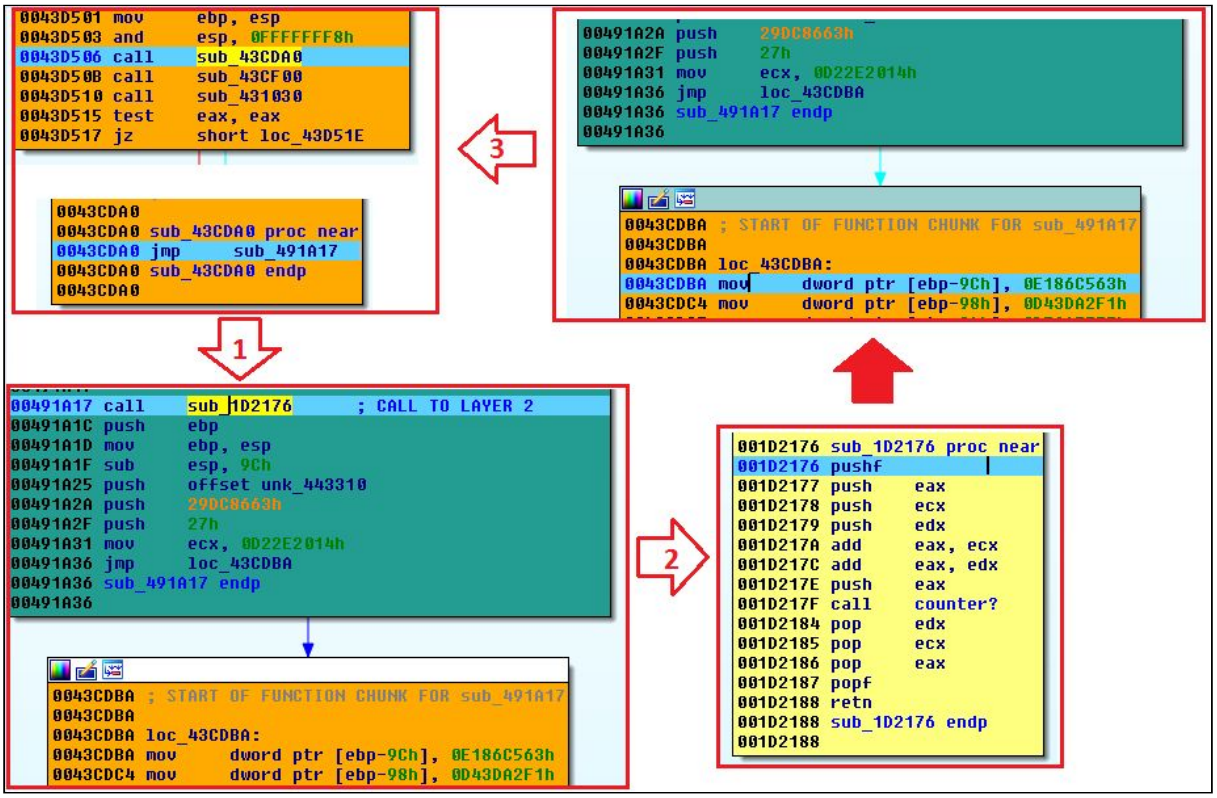

Image 9: Flow of execution of the first Payload call.

5. COMPARING LAYERS

Then, 4 files from Layer 1 were chosen randomly to make comparisons between their code. This will show the complexity and dynamism of this packer.

It also compares the Layer 2 extracted from these Layer 1 as well as the final payload they protect.

Layer 1 is undoubtedly the binary that most alters its code. Three different variants of Layer 2 have been found, while the final payload of Emotet are practically the same, except for its static configuration.

This makes a lot of sense since in the first instance the files that are going to be analyzed by AV, Firewalls... will be those of the Layer 1 type, so the less they look like each other, the more likely it is that they will not be detected through the use of signatures or other classic detection methods.

Despite the great difference between some files and others, especially in layer 1, as it is the same packer, the behavior of these binaries ends up being the same, so you can extract a pattern of behavior.

The Intezer platform was used to extract genes from the code. Intezer is a code analysis company (https://analyze.intezer.com). They extract genes from the code and are able to relate malware families to it.

The hash algorithm KOKA, implemented by Joxean Koret (**@matalaz**) which hashes functions based on his code graph, has also been used. In this way we will try to find equal hashes between the different samples to see if they share some similarity at code level. The hashes shown of KOKA have been limited to 32 characters, in this way they can be visualized well in the document.

5.1 LAYER 1

ANALYZED FILES		
ID	**SHA-1**	**SIZE**
1	465762C1C95C5086FE063FD4B2F53D8A40BBE582	120 KB
2	8BD655A87388F28F4C42637556CD4DCBE4943F3A	379 KB
3	E8ED4EBFD326266DE209CF382073D16B53A4F4E6	492 KB
4	469697E23A8A32165851300E3BEC537D18D42204	524 KB

Table 3: Files used to make the comparison. - Layer 1.

5.1.1 Metadata

In the following image you can see how each binary does not have much similarity with the others.

property	value
md5	E65EAFA0474DDFDCB5B39B6695DEF4E1
sha1	465762C1C95C5086FE063FD4B2F53D8A40BBE582
imphash	n/a
cpu	32-bit
size	387584 bytes
entropy	5.359
file-version	11.00.9600.18616 (winblue_ltsb.1703
file-description	IE ETW Collector Pro
compiler-stamp	Tue Dec 04 11:15:23 2018
debugger-stamp	n/a
type	executable
subsystem	GUI
signature	n/a

- 1 -

property	value
md5	BD2794723343095B01F784A6F4053B48
sha1	8BD655A87388F28F4C42637556CD4DCBE4943F3A
imphash	n/a
cpu	32-bit
size	122880 bytes
entropy	7.363
file-version	6.1.7600.1
file-description	Softpub Forwarder DLL
compiler-stamp	Tue Aug 21 04:59:10 2007
debugger-stamp	Tue Dec 04 06:46:19 2018
type	executable
subsystem	GUI
signature	n/a

- 2 -

property	value
md5	E454D4F712F505553904B631EA5A9774
sha1	E8ED4EBFD326266DE209CF382073D16B53A4F4E6
imphash	n/a
cpu	32-bit
size	503808 bytes
entropy	4.718
file-version	7.6.7601.1
file-description	Windows
compiler-stamp	Sun Dec 02 13:33:10 2018
debugger-stamp	Sun Dec 02 13:33:10 2018
type	executable
subsystem	GUI
signature	n/a

- 3 -

property	value
md5	67A9D53B66CB757DE7BE8B6B777B9CAE
sha1	469697E23A8A32165851300E3BEC537D18D42204
imphash	1CC45F2926C4CDC47EC2479BB5AE71BB
cpu	32-bit
size	536576 bytes
entropy	4.819
file-version	7.6.7601.1
file-description	Windows
compiler-stamp	Mon Dec 03 17:17:03 2018
debugger-stamp	Mon Dec 03 17:17:03 2018
type	executable
subsystem	GUI
signature	n/a

- 4 -

Image 10: Difference by metadata - Layer 1

5.1.2 Sections

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
.text	1000	2000	1000	1F5E	60000020	0	0	0
.rdata	3000	1000	3000	7D0	40000040	0	0	0
.CRT	4000	17000	4000	17770	C0000040	0	0	0
CONST	1B000	1000	1C000	AC	C0000040	0	0	0
.rsrc	1C000	1000	1D000	720	40000040	0	0	0
.reloc	1D000	1000	1E000	180	42000040	0	0	0

- 1 -

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
.text	400	3000	1000	2FAE	60000020	0	0	0
.data	3400	AE00	4000	C7F8	C000003F	0	0	0
.text	E200	600	11000	536	4000003F	0	0	0
_D2	E800	2B200	12000	2B18D	C0000040	0	0	0
PACK	39A00	23A00	3E000	23974	C0000040	0	0	0
.rsrc	5D400	A00	62000	980	40000040	0	0	0
.reloc	5DE00	C00	63000	ABC	42000040	0	0	0

- 2 -

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
.text	1000	5000	1000	489A	60000020	0	0	0
.rdata	6000	2000	6000	178E	4000003F	0	0	0
.rsrc	8000	71000	8000	72218	C0000040	0	0	0
DATA	79000	1000	7B000	A88	40000040	0	0	0
.reloc	7A000	1000	7C000	828	42000040	0	0	0

- 3 -

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
.text	1000	D000	1000	C10A	60000020	0	0	0
.rdata	E000	2000	E000	18B0	4000003F	0	0	0
.rsrc	10000	71000	10000	71FDC	C0000040	0	0	0
DATA	81000	1000	82000	A88	40000040	0	0	0
.reloc	82000	1000	83000	78C	42000040	0	0	0

- 4 -

Image 11: Difference by sections - Layer 1

In the image above you can see how neither the number of sections, nor their name and much less their characteristics coincide.

This means that Layer 1 is 100% custom in compilation time so that they do not look at all like each other's samples.

5.1.3 Code

Analyzing the EntryPoint of each of these binaries we can see that with the exception of binaries 3 and 4, the rest have nothing in common.

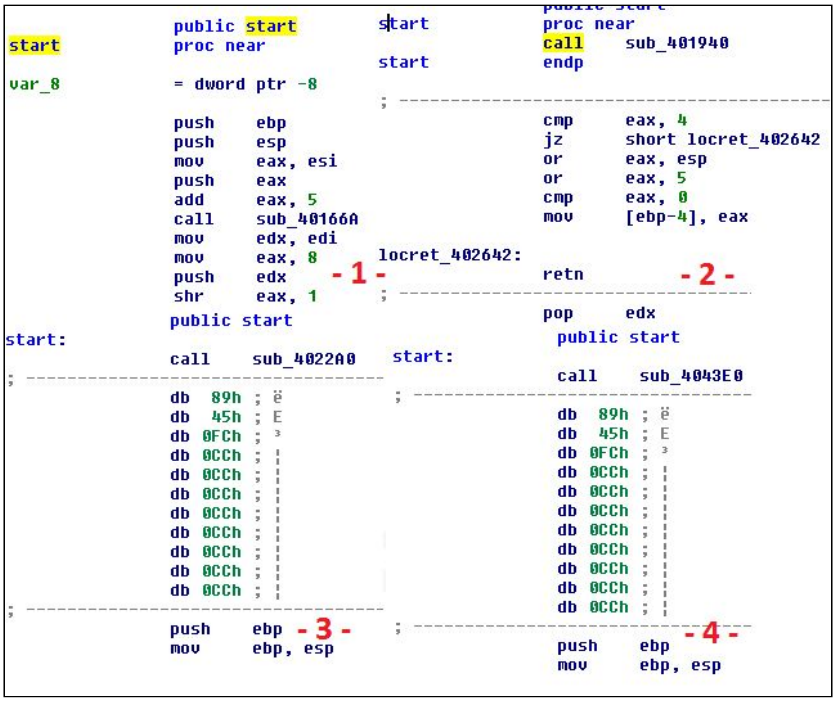

Image 12: EntryPoint code for each sample - Layer 1

However, comparing the graphics of the functions used to decrypt, map, and run Layer 2, they have nothing to do with each other.

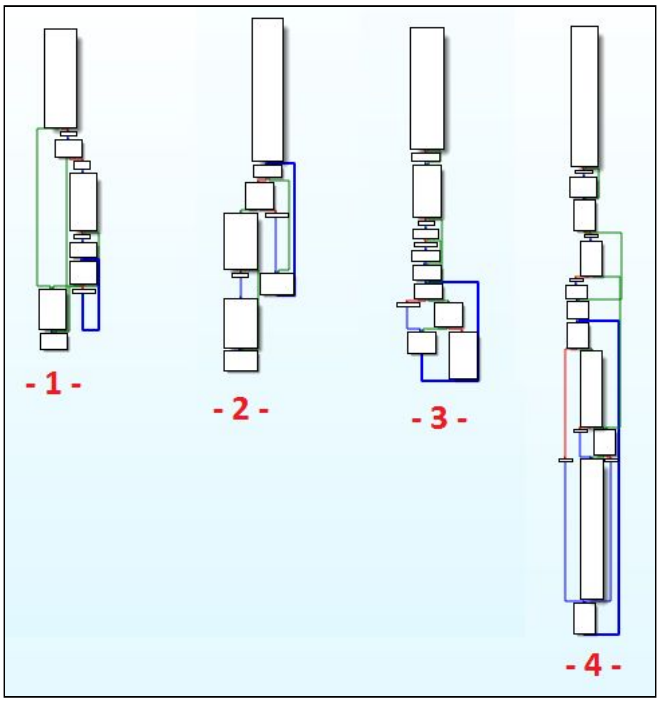

Image 13: Graph of the function that loads in memory and pass the execution to Layer 2 - Layer 1

5.1.4 Genes

Verdict	Family	Time	SHA256	Filename
Malicious		23:29	8c1daa3b27e6d5fb9d7e476937507953f97dac1eb25b8a12a04...	4.exe
Malicious	Emotet	23:29	dc9f1171bee4f5cac436903afb7520c3349bdafa3061c00bc3a1...	3.exe
Malicious	Emotet	23:29	a6492280560d012bf18891908b905f993b231cde63a1311ede...	1.exe
Malicious	Emotet	23:28	a58952b881e09dc348e48c914e1d11ccecf67a4c4dd7f0a6bd6...	2.exe

Image 14: Gene based family detection - Layer 1

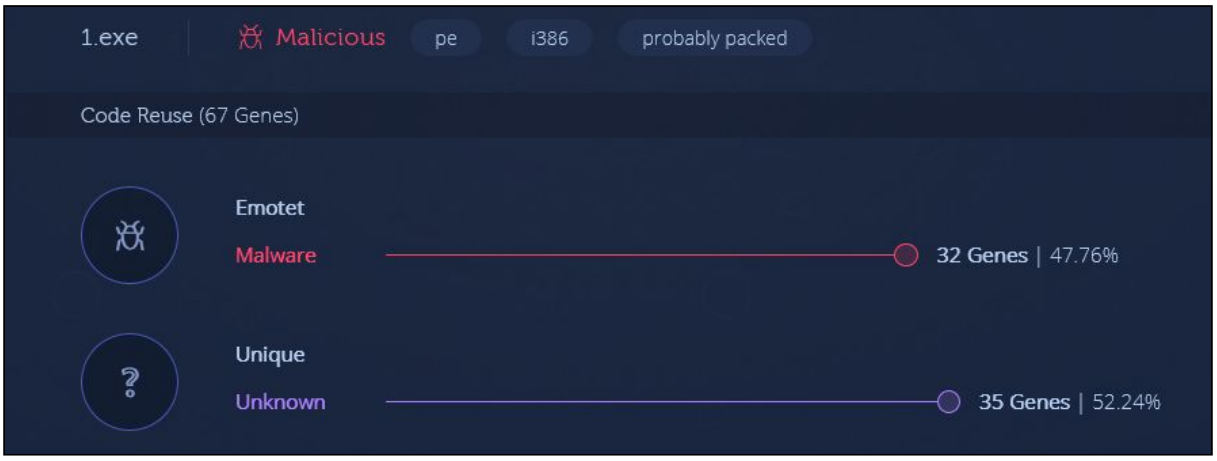

Image 15: 32 genes related to Emotet in sample 1 - Layer 1

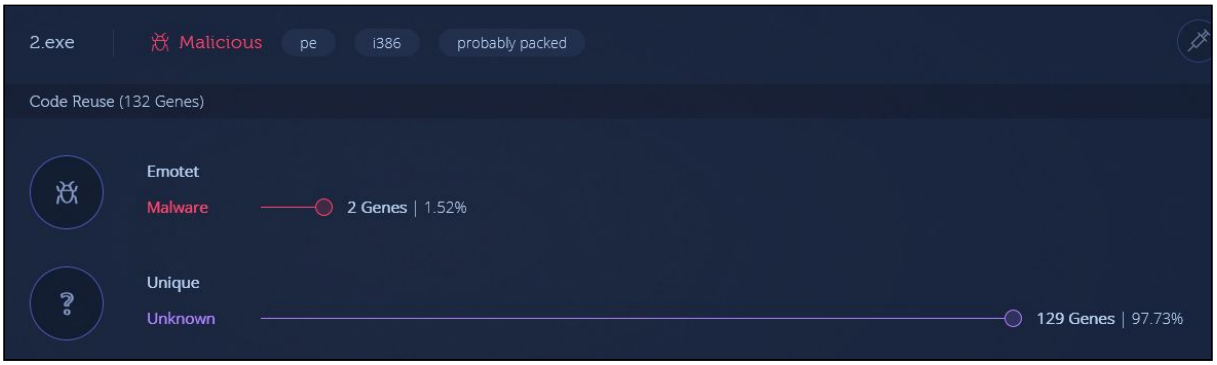

Image 16: 2 genes related to Emotet in sample 2 - Layer 1

Image 17: 3 genes related to Emotet in sample 3 - Layer 1

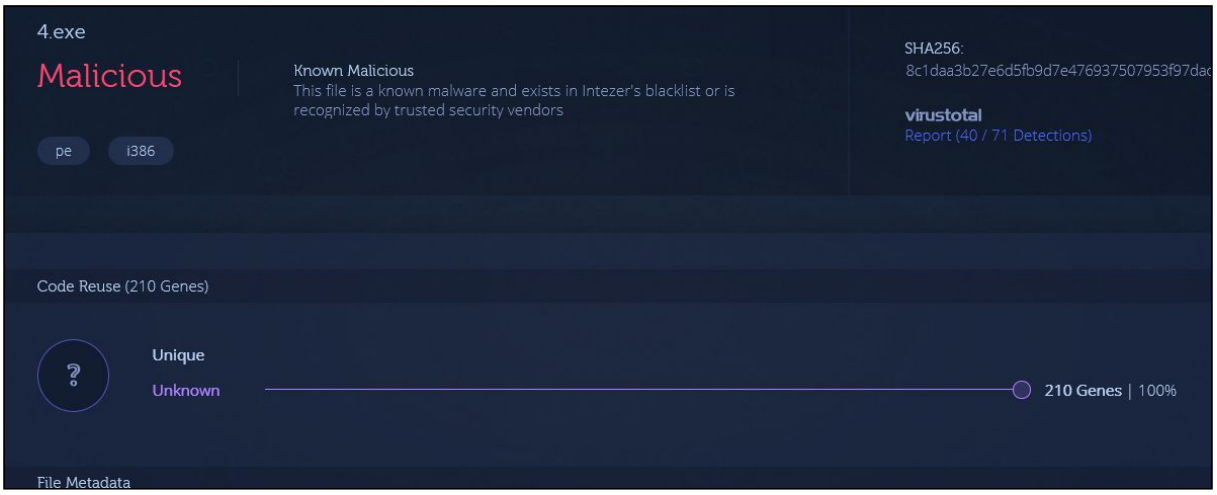

Image 18: No gene related to Emotet in sample 4 - Layer 1

As shown in the images above, except for the first binary, the rest barely have genes in common with the Emotet, this means that this packer is even able to pass a malware unnoticed even when using cutting-edge technologies such as gene comparison. This is a good example of the cat and mouse game that has always happened and will happen between researchers and malware creators.

Anyway, it is a good result as they are able to relate the genes to the right malware family.

5.1.5 KOKA

HASH	ID - MATCH ADDRESS			
	1	2	3	4
1426568515250758...	x	0x00401940	x	0x004043e0

Table 4: KOKA Hash common between files. - Layer 1.

5.2 LAYER 2

The Layer 2 binaries used for this comparison have been extracted from the binaries used in the previous point.

ANALYZED FILES		
ID	**SHA-1**	**SIZE**
1	66EA7ABF72B9B5698AFBB9B2C4767358624E9363	20 KB

2	553D7427F43FFFBE62352EAEC3A77F94849AB934	19 KB
3	112A82E2B91A79A0D5F3C2E9AF959D137F66A407	18 KB
4	BDA06AFEC3F6A3DC47C92DEE95DA531CAFCC0A8F	18 KB

Table 5: Files used to make the comparison. - Layer 2.

5.2.1 Metadata

In this case there is nothing remarkable, except for the MD5, everything seems very similar, even the entropy.

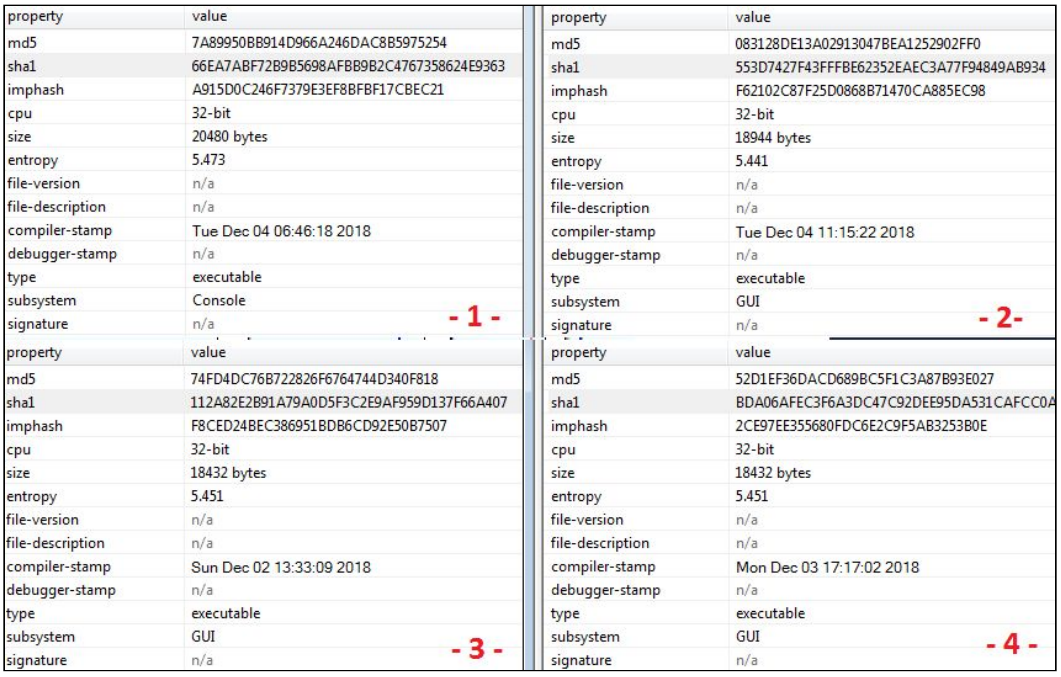

Image 19: Difference by metadata - Layer 2

5.2.2 Sections

In this case, all binaries have the same number of sections, even the same names. In the case of 3 and 4, as seen in the image below, they match up the size of the sections.

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics
.text	400	2000	1000	1EAC	60000020
.rdata	2400	2200	3000	210C	40000040
.data	4600	600	6000	408	E0000040
.reloc	4C00	400	7000	F384	42000040 1 -
Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics
.text	400	1C00	1000	1A3A	60000020
.rdata	2000	2000	3000	1F1A	40000040
.data	4000	600	5000	404	E0000040
.reloc	4600	400	6000	10350	42000040 2 -
Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics
.text	400	1A00	1000	194C	60000020
.rdata	1E00	2000	3000	1EC0	40000040
.data	3E00	600	5000	404	E0000040
.reloc	4400	400	6000	1034C	42000040 3 -
Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics
.text	400	1A00	1000	194C	60000020
.rdata	1E00	2000	3000	1EC0	40000040
.data	3E00	600	5000	404	E0000040
.reloc	4400	400	6000	1034C	42000040 4 -

Image 20: Difference by sections - Layer 2

5.2.3 Code

Analyzing the Entry Point graph, we can find more similarities than in the Layer 1 samples. 3 and 4 look the same.

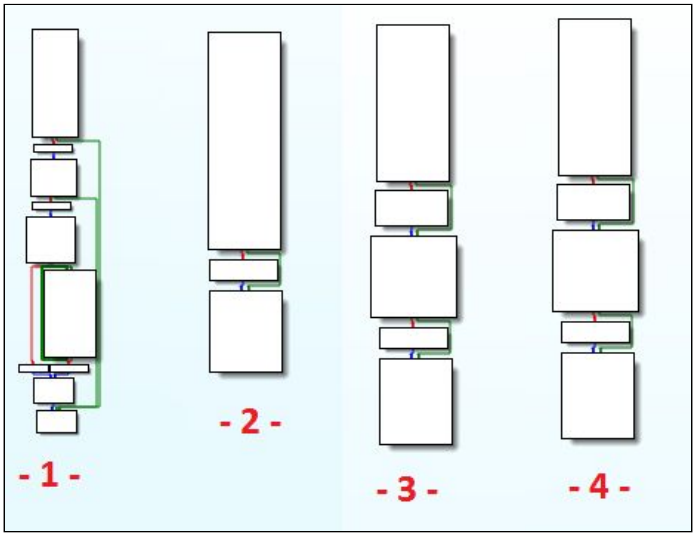

Image 21: Entry Poin function graph - Layer 2

If you look at the function in charge of passing the execution to the Payload, in this case we only find 3 different graphs between the 4 samples.

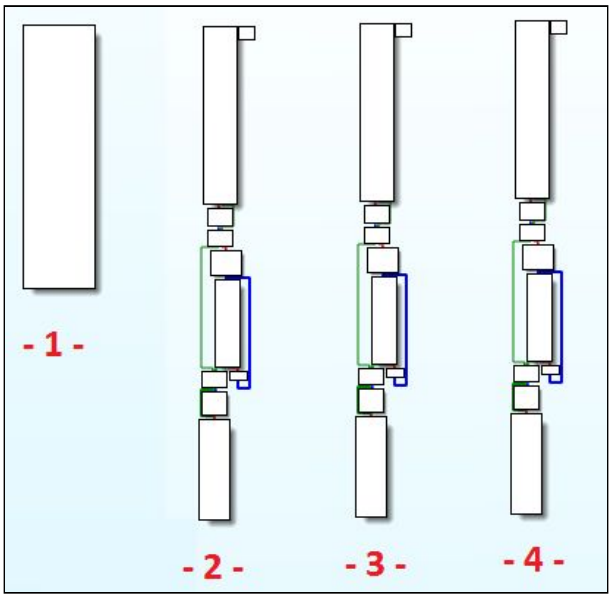

Image 22: Graphic of the function that passes the execution to the Payload - Layer 2

5.2.4 Genes

In this case the genes reveal many more relationships than in Layer 1, something obvious since it is seen how little by little these files look each other more.

All binaries of type Layer 2 have genes related to Emotet.

Verdict	Family	Time	SHA256	Filename
Malicious	Emotet	00:49	d2ab5b81ea2839c2d3976bc07ffe73723a2b163834f534576eb...	4.exe.layer2
Malicious	Emotet	00:49	cdc2499e1719278946f80b21093213d63c94940947675af5908...	3.exe.layer2
Malicious	Emotet	00:49	fb29d79ff3f6417c70e79fbab53896a104c5ab9bdd3b43554ad1...	2.exe.layer2
Malicious	Emotet	00:48	c6127f48bc84c10679b51d7df3b719f9c5676cdf6da53d28d43...	1.exe.layer2

Image 23: Family detection by genes - Layer 2

Image 24: 2 genes related to Emotet in sample 1 - Layer 2

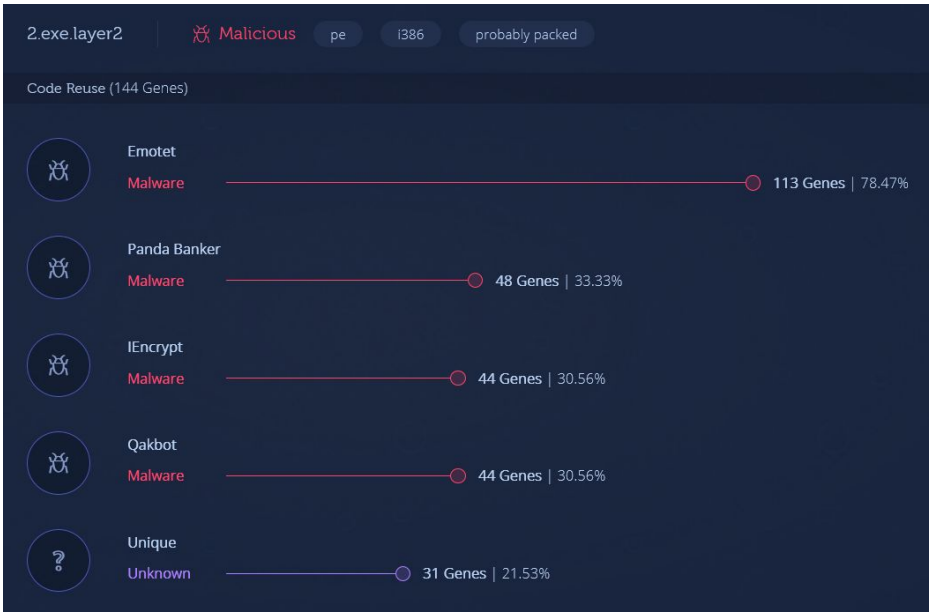

Image 25: 113 genes related to Emotet in sample 2 - Layer 2

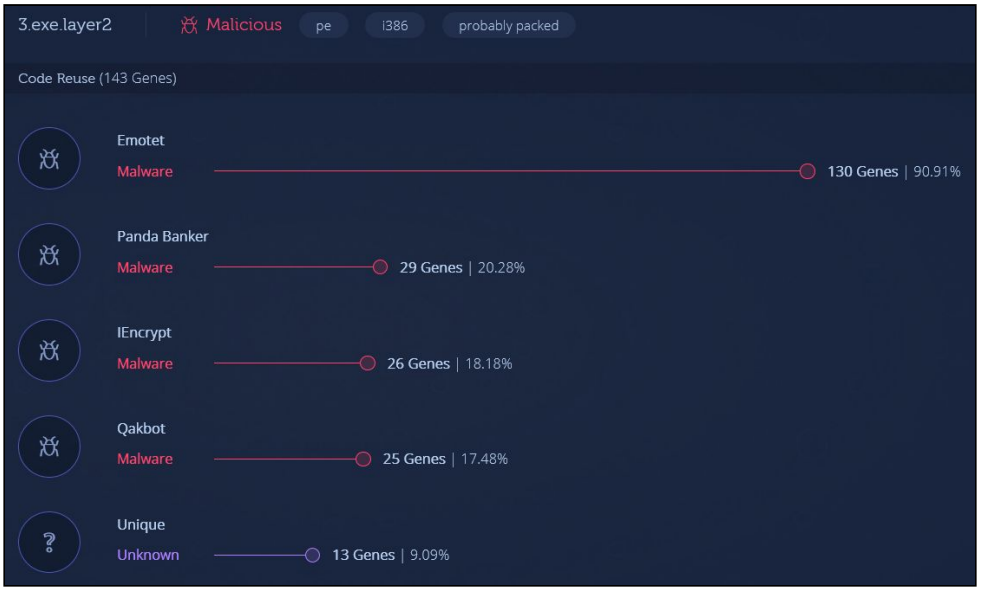

Image 26: 130 genes related to Emotet in sample 3 - Layer 2

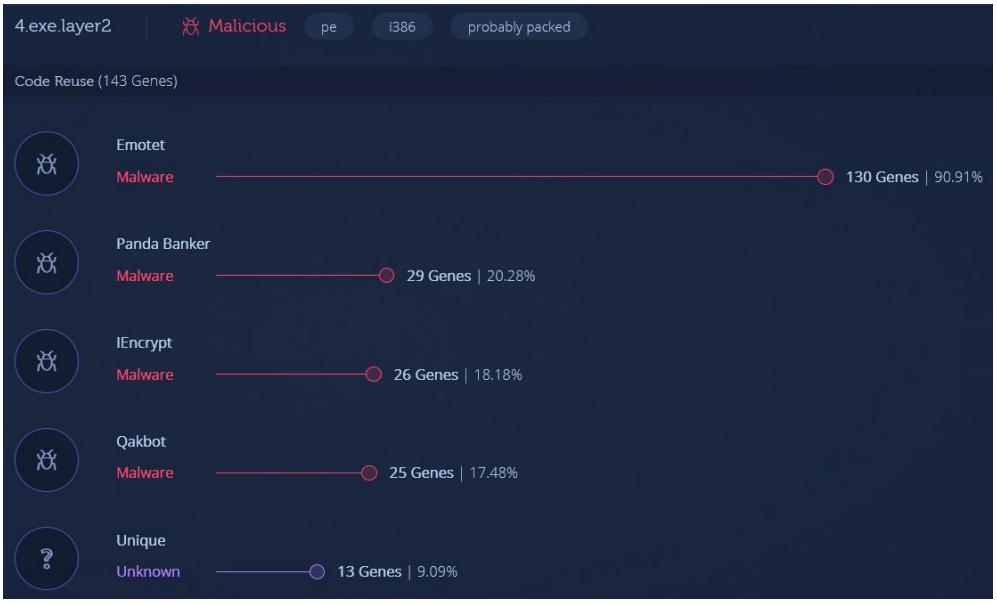

Image 27: 130 genes related to Emotet in sample 4 - Layer 2

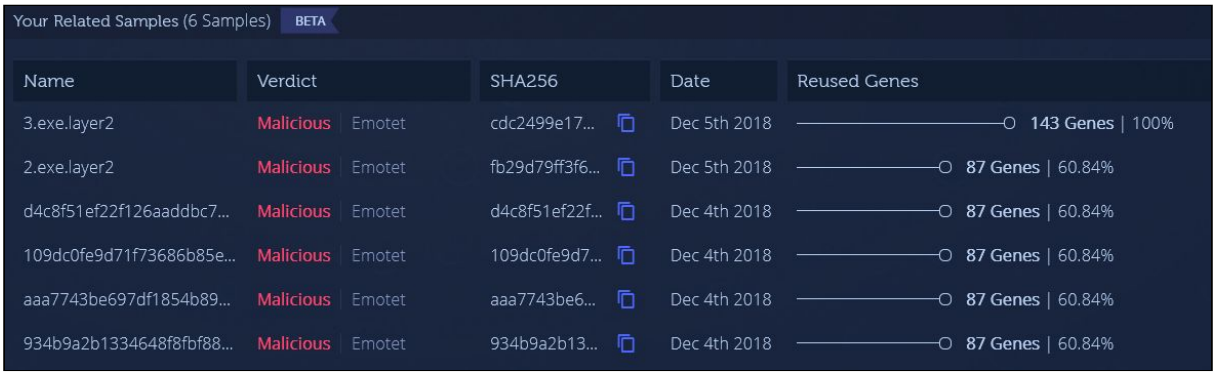

Image 28: Binaries related to sample 4 - Layer 2

As expected, binaries 2, 3, 4 share genes. Specifically 3 and 4 share 143 genes. As already seen in the code level comparisons, the functions such as the sections are very similar.

5.2.5 KOKA

HASH	ID - MATCH ADDRESS			
	1	2	3	4
1828817393860985...	x	x	0x00401527	0x00401527
2657949054763752...	x	0x00402189	0x00402080	0x00402080
2166862924285466...	x	0x00401425	0x00401425	0x00401425
7761583045172118...	x	x	0x00401ccd	0x00401ccd

3753977961719393...	x	0x00401123	0x00401127	0x00401127
2716965301668314...	x	x	0x004028b7	0x004028b7
2700436380578111...	x	0x00402023	0x00401f39	0x00401f39
1789901355864619...	x	x	0x00401f85	0x00401f85
4159986491537749...	x	x	0x00401aca	0x00401aca
2416108689769958...	x	0x00402943	0x00402846	0x00402846
2235842324693619...	x	0x00401ebe	0x00401dd4	0x00401dd4
1215250050465560...	x	x	0x00401c17	0x00401c17
2595167388678237...	x	0x004012e5	0x004012e5	0x004012e5
2113985795951996...	x	x	0x004022c4	0x004022c4
1393482880147061...	x	x	0x00401931	0x00401931
9402495682879960...	x	x	0x004018a8	0x004018a8
1739950245997134...	x	0x0040104c	0x00401050	0x00401050
1914949788848249...	x	x	0x00402537	0x00402537

Table 6: KOKA Hash common between files. - Layer 2.

5.3 PAYLOAD

The binaries of the Payload type used for the comparison have been extracted from the binaries used in the previous points. In the case of the payload, the binaries being compared have been extracted and modified by the unpacker provided in this same repository.

ANALYZED FILES		
ID	**SHA-1**	**SIZE**
1	AE522C21592C134E50C54F973D6E89F1BE99C600	69 KB
2	AE522C21592C134E50C54F973D6E89F1BE99C600	69 KB
3	67AE20F0ADF4917B1741861E619306BBF4D1AD03	69 KB
4	AE522C21592C134E50C54F973D6E89F1BE99C600	69 KB

Table 7: Files used to make the comparison. - Payload.

Looking carefully at the table above, we see that file 1, 2 and 4 are the same, that is to say, although their Layer 1 and Layer 2 are so different, they protect the same binary. And this is where the potential of this packer is found.

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	
.text	400	D400	1000	D395	60000020	
.rdata	D800	C00	F000	B00	40000040	
.data	E400	2800	10000	5608	C0000040	
.reloc	10C00	600	16000	4A4	42000040	- 1 -
Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	
.text	400	D400	1000	D395	60000020	
.rdata	D800	C00	F000	B00	40000040	
.data	E400	2800	10000	5608	C0000040	
.reloc	10C00	600	16000	4A4	42000040	- 2 -
Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	
.text	400	D400	1000	D395	60000020	
.rdata	D800	C00	F000	B00	40000040	
.data	E400	2800	10000	5608	C0000040	
.reloc	10C00	600	16000	4A4	42000040	- 3 -
Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	
.text	400	D400	1000	D395	60000020	
.rdata	D800	C00	F000	B00	40000040	
.data	E400	2800	10000	5608	C0000040	
.reloc	10C00	600	16000	4A4	42000040	- 4 -

Image 29: Difference by section. - Layer 2

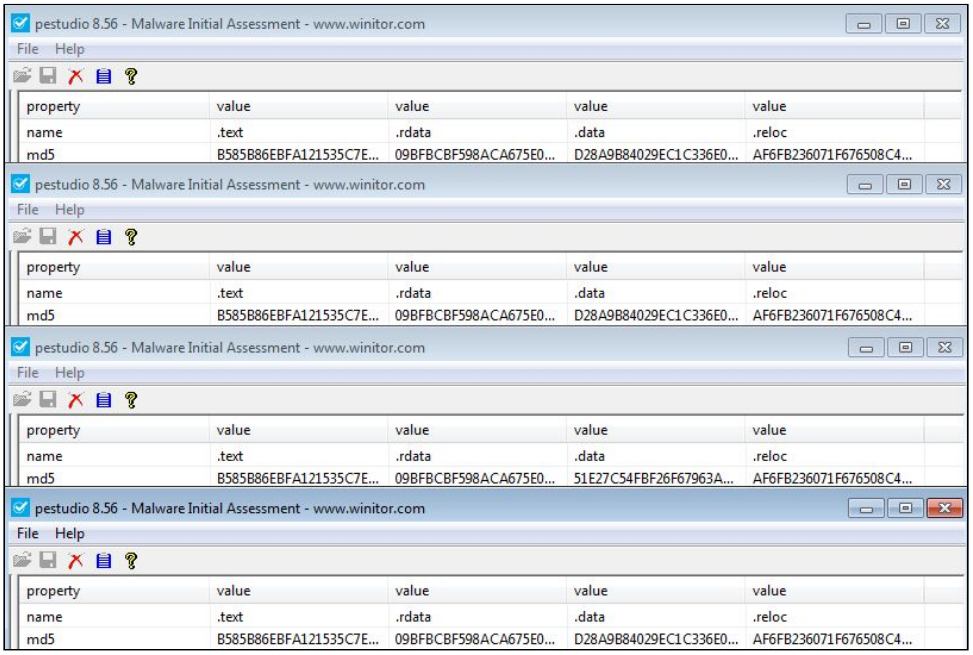

Image 30: Comparing md5 from each section. - Layer 2

In the image above you can see the MD5 of each section. Both the ".text", ".rdata" and ".reloc" sections are the same, with the exception of the ".data" section, where the static configuration is located.

At this point, it makes no sense to keep comparing its code as you can see that it is exactly the same.

6. TOOLS

During this research a number of tools have been developed that can help analysts to analyse, detect, unpack and extract control panels from most Emotet samples.

These tools have been tested with a few Emotet samples and for almost all, samples it has worked. This does not mean that samples dated after the publication of this article will work, as the criminal gang behind this malware is constantly developing.

6.1 UNPACKER

As we have already seen, obtaining the final sample of the emotet to be able to analyze it comfortably, is not an easy task. Because the code is constantly changing, it uses anti-dump techniques etc.

After a hard work of analysis of multiple binaries that protect the Emotet code, it has been discovered that all of them follow the same pattern, and thanks to this pattern it has been possible to make an unpacker, which is published in this same repository.

Basically, the unpacking pattern is as follows:

1. Layer 1 deciphers layer 2.
2. Layer 2 is mapped in memory.
3. Layer 1 passes the execution flow to layer 2 using one of the following instructions:
    a. call eax
    b. call edx
    c. call esi
    d. call edi
4. Layer 2 deciphers the final payload of the Emotet
5. The final payload mapped in memory and is hooked.
6. Layer 2 passes the execution flow to the payload using the following instruction:
    a. jmp eax
7. At that moment the Emotet is already unpacked.

This whole process has been automated using the python API of *TitanEngine*. (https://www.reversinglabs.com/open-source/titanengine.html).

The unpacker has two execution modes, one with graphical interface and one without graphical interface.

6.1.1 GUI

If you double-click on the main.py file or if you run it by command line without arguments, the graphical interface opens.

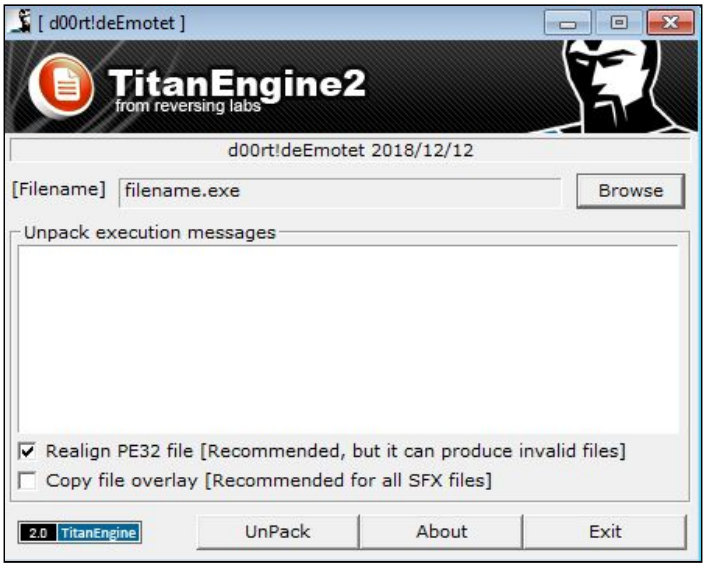

Image 31: Emotet unpacker GUI.

Selecting the file we want to unpack and clicking the "UnPack" button, the sample will be unpacked, if all goes well a folder called output is created containing the different files of each layer of the packer, including the fixed payload.

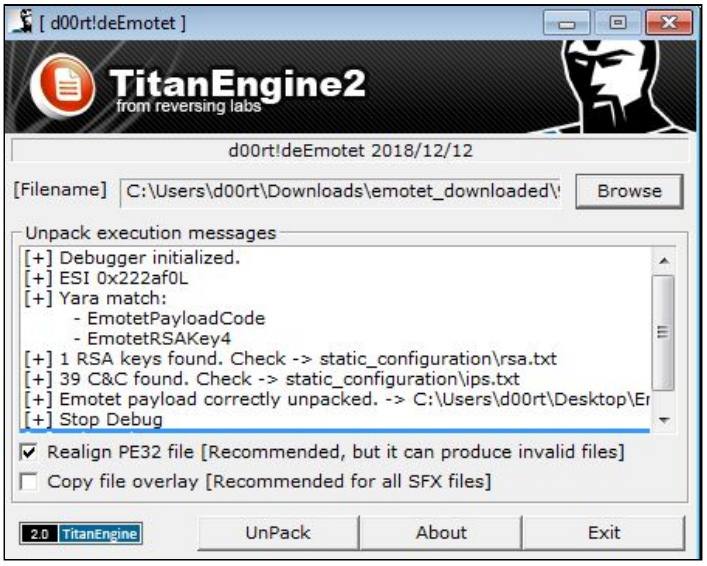

Image 32: Unpacker interface after unpacking a sample.

Image 33: Content of the folder where the unpacker leaves its results.

6.1.2 Command Line

If the main.py is executed by command line passing it an argument, which can be either a folder or a file, the graphical interface will not be opened and the output will be printed on the console.

```
PS C:\Users\d00rt> C:\Python27\python.exe C:\Users\d00rt\Desktop\EmotetUnpacker\src\main.py C:\Users\d00rt\Downloads\emo
tet_downloaded
[+] MD5: 05e53052fddc653ce863195d23908f40 FILENAME: 03izhqgWA.exe
    UNPACKED: True, RSA_KEYS: 1, C&C: 38, MATCH1: [EmotetRSAKey3] MATCH2: [EmotetPayloadCode]
[+] MD5: 3adea9f1a9f12a577c257bc39f3b2be1 FILENAME: 06337328.exe
    UNPACKED: True, RSA_KEYS: 1, C&C: 39, MATCH1: [EmotetRSAKey4] MATCH2: [EmotetPayloadCode]
[+] MD5: 61fd6d472d3c6402c1eb5b6d7c121c06 FILENAME: 0790.exe
    UNPACKED: True, RSA_KEYS: 1, C&C: 38, MATCH1: [EmotetRSAKey4] MATCH2: [EmotetPayloadCode]
[+] MD5: 7e7317db5c6bb0ba824416aa9315349c FILENAME: 134383.exe
    UNPACKED: True, RSA_KEYS: 1, C&C: 38, MATCH1: [EmotetRSAKey4] MATCH2: [EmotetPayloadCode]
[+] MD5: 387926535ac3aa42fd091a724b898a08 FILENAME: 14.exe
    UNPACKED: True, RSA_KEYS: 1, C&C: 38, MATCH1: [EmotetRSAKey4] MATCH2: [EmotetPayloadCode]
[+] MD5: 61fd6d472d3c6402c1eb5b6d7c121c06 FILENAME: 192.exe
    UNPACKED: True, RSA_KEYS: 1, C&C: 38, MATCH1: [EmotetRSAKey4] MATCH2: [EmotetPayloadCode]
[+] MD5: df626567878cf99fc92102e82d2e35c9 FILENAME: 3XzgYvoG.exe
    UNPACKED: True, RSA_KEYS: 1, C&C: 37, MATCH1: [EmotetRSAKey3] MATCH2: [EmotetPayloadCode]
[+] MD5: 387926535ac3aa42fd091a724b898a08 FILENAME: 48028516.exe
    UNPACKED: True, RSA_KEYS: 1, C&C: 38, MATCH1: [EmotetRSAKey4] MATCH2: [EmotetPayloadCode]
[+] MD5: c4680e34726ad13296ae6a9fc7c5d540 FILENAME: 4b7Zv1hv7.exe
```

Image 34: Output of the unpacker when a folder with Emotet files is passed to it as a parameter.

6.2 CONFIGURATION EXTRACTOR

The unpacker also has the ability to extract the static configuration from the payload. This configuration consists of a list of control panels and an RSA key. The ip:port list is used by Emotet to try to download from there other binaries such as *PandaBanker*, *Trikcbot* or *IceID*. The RSA key is used to encrypt this communication.

It could be said that the RSA key represents the same Botnet. During this investigation 5 different RSA keys have been found. These keys are added to a .yar file in the same repository.

The static configuration is extracted and saved in the folder "static_configuration" inside the folder "output" generated by the unpacker.

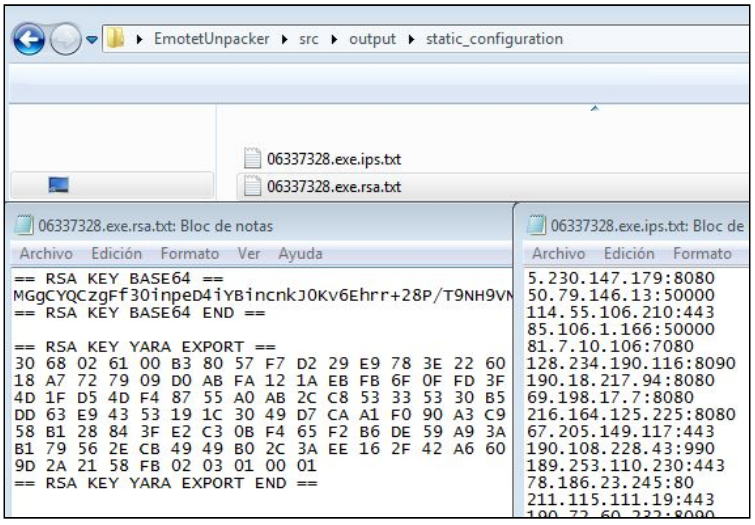

Image 35: Static configuration of an Emotet binary.

6.3 PAYLOAD DOCUMENTATION

In the folder "unpacked_sample_idc" there is an unpacked Emotet binary (emotet_unpacked.bin), next to an IDC file (emotet_unpacked.idc).

If you open the binary with the IDA, and then execute the IDC script, automatically all functions will be renamed.

The IDC has been generated from an IDB that has been created trying to document much of the functionality of the Emotet and its features. In this way it is easier for the community of researchers to analyze this family and its capabilities.

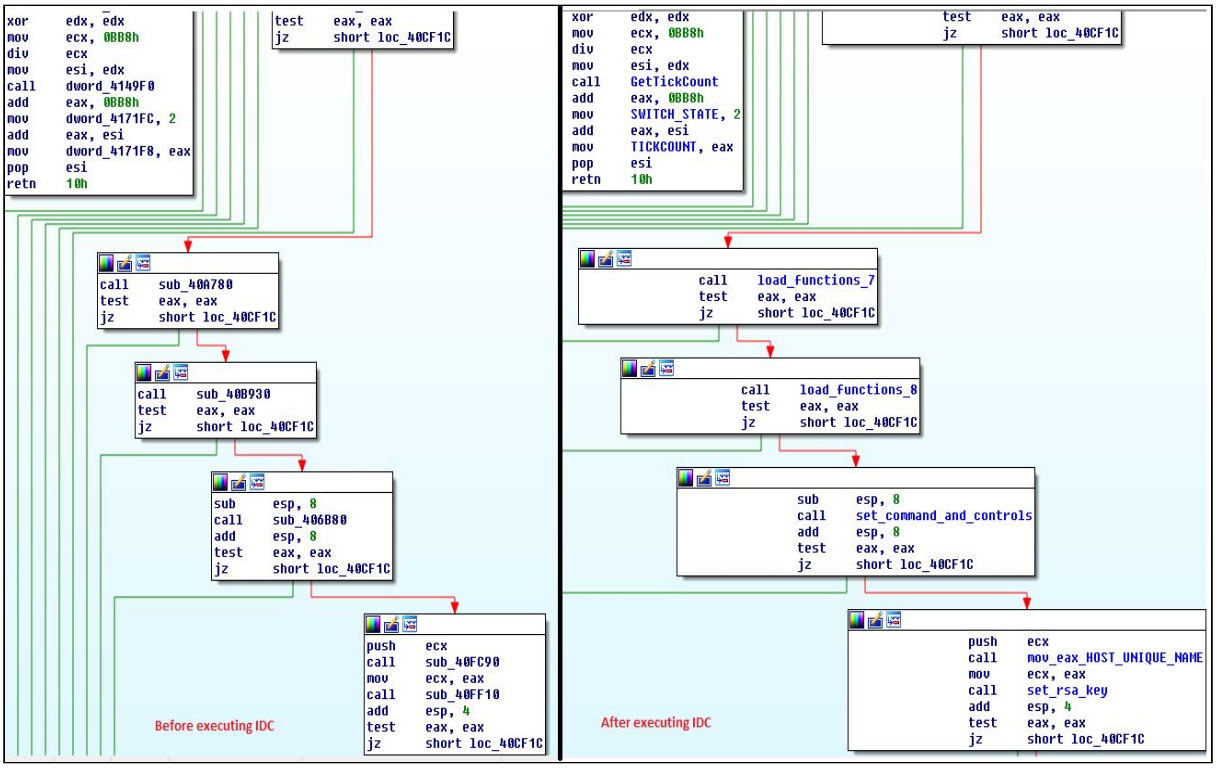

Image 36: Code fragment before and after running the script.

7. CONCLUSIONS

The Emotet packer consists of 3 components, the protector (layer 1), the Emotet loader (layer 2), and the Emotet itself (payload).

As seen during the article, Layer 1 changes its code constantly and this makes detection of it by traditional methods such as signatures practically impossible.

Also 2-3 different types of layer 2 have been detected, which also makes it difficult to unpack it.

Although not detailed, the difference between the different Layer 1, seem to be related to opaque predicates based on tautologies, with which in compilation time the actors are able to generate completely different code and with different genes/graphs, which are not easy to normalize.

The array used by Layer 1 to decipher Layer 2, in each binary has been randomized, so to find out in which position of the binary is the array, to be able to reconstruct Layer 2 in a static way is a very complicated task.

The fact that in less than a year there are 2 different versions of Emotet and that all the techniques used are complex and advanced, shows that this malware is in constant development by highly qualified people.

5 RSA keys were detected during the investigation. This RSA key is used to communicate with the C&C so it could relate the RSA key to the Botnet ID.

The botnets seen in the last samples analyzed correspond to EmotetRSAKey3 and EmotetRSAKey4. (Names given to the yara rule that you can find in this repository)

The fact that 5 different RSA keys have been detected (there could be more), but that the ones that are always more active are only 2, suggests two things to the author:

1) That there are 5 different botnets.
2) That there are only 2 botnets and that over time this RSA key has been updated in the bots, because although the communication protocol of the Emotet has not been described, it would easily have this capacity.

Everything points to a criminal gang which has the power of the malware code and change it according to their needs. And they are using the botnets they have to distribute malware from other actors who contract their services.

As a curiosity, during the investigation, all Emotet files that are downloaded with a name consisting only of numbers used the EmotetRSAKey3 key while those that also use letters in their names use the EmotetRSAKey4 key.

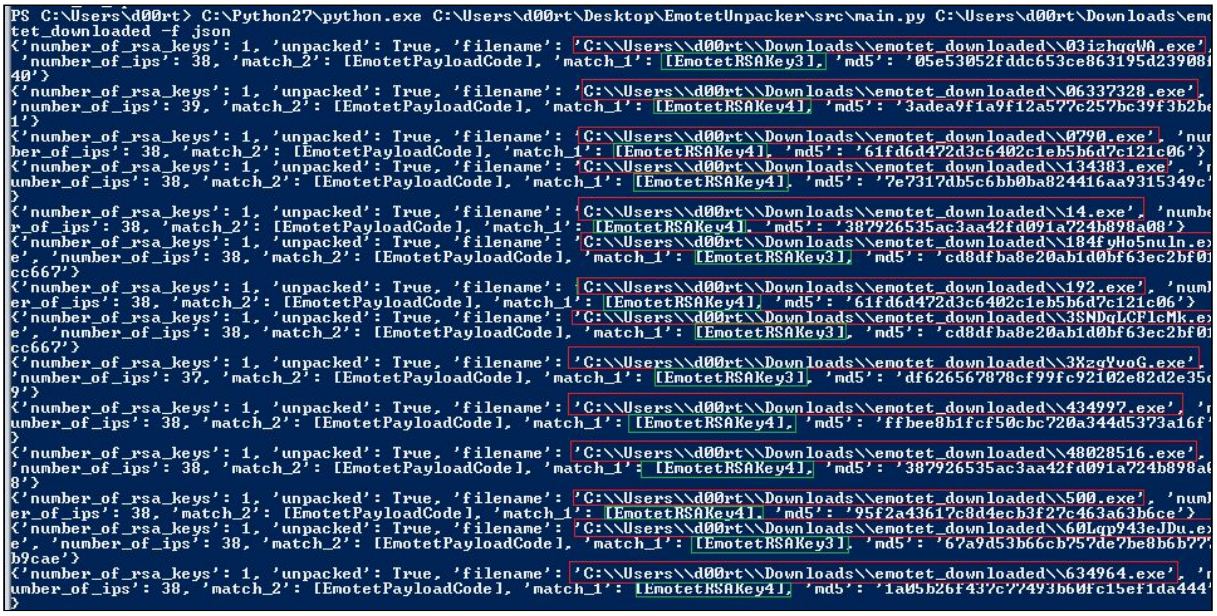

Image 37: Output of the unpacker executed by command line.

Finally, I invite anyone to continue studying this family and try to improve the unpacker. The twitter [@D00RT_RM](https://twitter.com/D00RT_RM) account is also available for any suggestion or contribution.