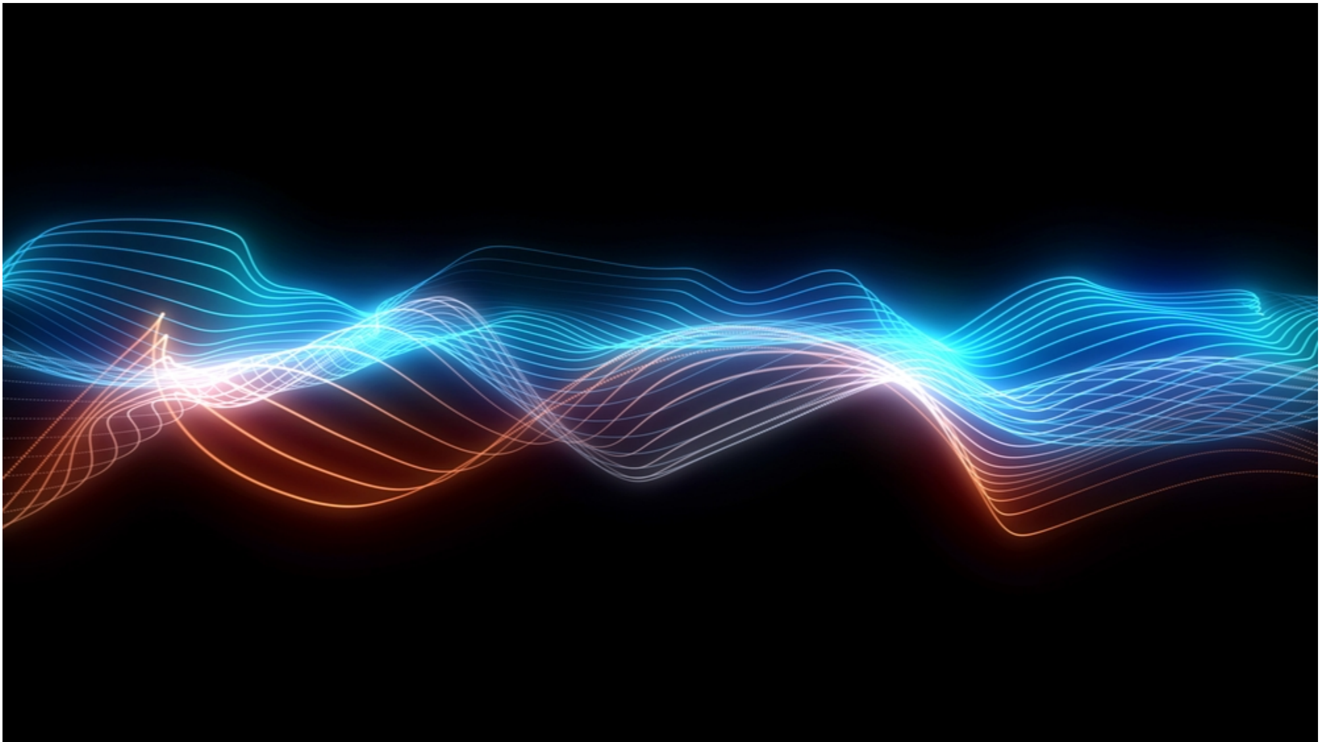
Why Emotet's Latest Wave is Harder to Catch than Ever Before

deepinstinct.com/2020/08/12/why-emotets-latest-wave-is-harder-to-catch-than-ever-before/

August 12, 2020



August 12, 2020 | Ron Ben Yizhak

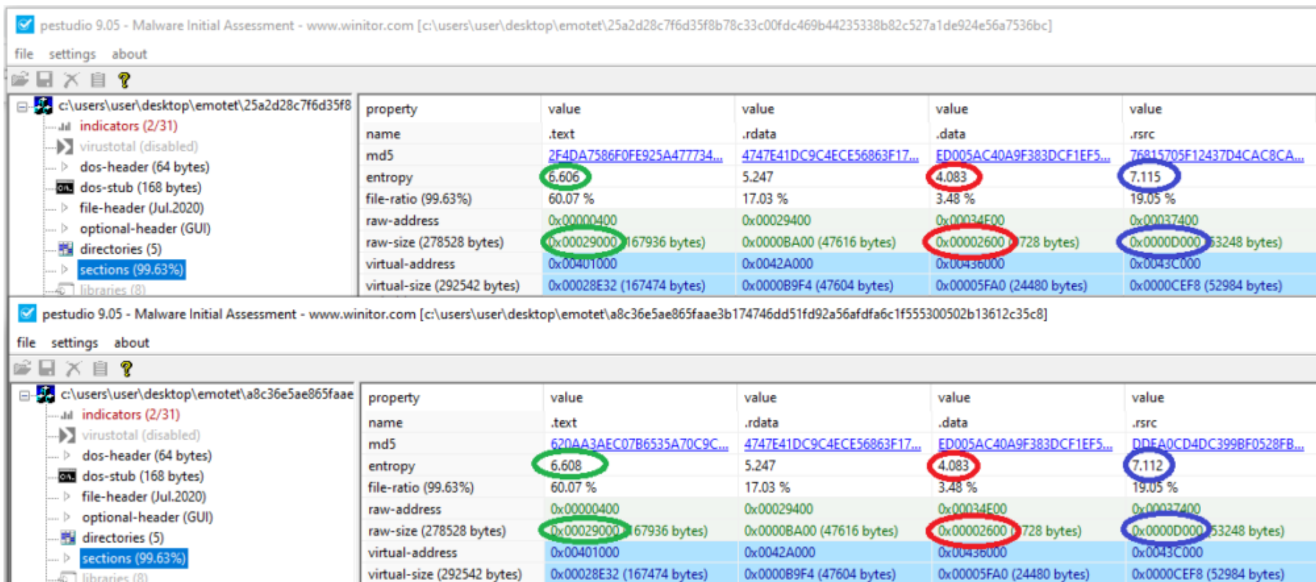
After five months of inactivity, the prolific and well-known Emotet botnet re-emerged on July 17th. The purpose of this botnet is to steal sensitive information from victims or provide an installation base for additional malware such as TrickBot, which then in many cases will drop ransomware or other malware. So far, in the current wave, it was observed delivering QakBot.

In this blog post, we reveal some novel evasion techniques which assist the new wave of Emotet to avoid detection. We discovered how its evasion techniques work and how to overcome them. The first part of the malware execution is the loader which is examined in this article, with an emphasis on the unpacking process. We also partition the current wave into several clusters, each cluster has some unique shared properties among the samples.

Clustering the samples

A dataset of 38 thousand samples was created using data collected by the Cryptolaemus group, from July 17th to July 28th. This group divides the samples into three epochs, which are separate botnets that operate independently from one another.

Static information was extracted from each sample in order to find consistent patterns across the entire data set. The information that is relevant to identify patterns includes the size and entropy of the .text, .data, .rsrc sections, and the size of the whole file, so this information was extracted from all the files. We started our analysis with two samples, an overview for which is provided in the image below. By looking at each file size we can see that the ratio of these sections remains the same, and the entropy differs very little between the files.

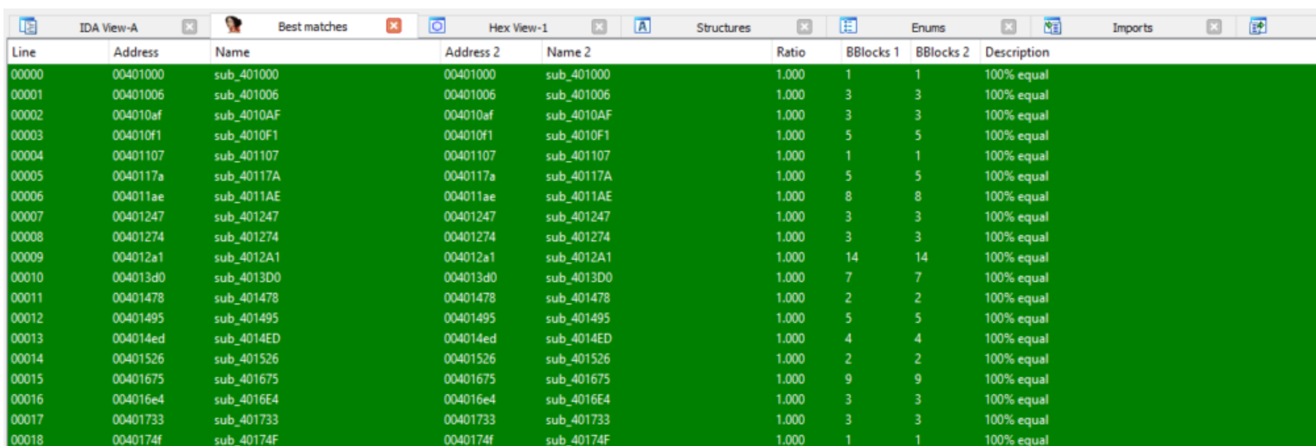


property	value	value	value	value
name	.text	.rdata	.data	.rsrc
md5	2F4DA7586F0FE925A477734...	4747E41DC9C4ECE56863F17...	ED005AC40A9F383DCF1EF5...	76815705F12437D4CAC8CA...
entropy	6.606	5.247	4.083	7.115
file-ratio (99.63%)	60.07 %	17.03 %	3.48 %	19.05 %
raw-address	0x00000400	0x00029400	0x00034E00	0x00037400
raw-size (278528 bytes)	0x00029000 167936 bytes)	0x0000BA00 (47616 bytes)	0x00002600 728 bytes)	0x0000D000 53248 bytes)
virtual-address	0x00401000	0x0042A000	0x00436000	0x0043C000
virtual-size (292542 bytes)	0x00028E32 (167474 bytes)	0x0000B9F4 (47604 bytes)	0x00005FA0 (24480 bytes)	0x0000CEF8 (52984 bytes)

property	value	value	value	value
name	.text	.rdata	.data	.rsrc
md5	620AA3AEC07B6535A70C9C...	4747E41DC9C4ECE56863F17...	ED005AC40A9F383DCF1EF5...	DDEA0CD4DC399BF0528FB...
entropy	6.608	5.247	4.083	7.112
file-ratio (99.63%)	60.07 %	17.03 %	3.48 %	19.05 %
raw-address	0x00000400	0x00029400	0x00034E00	0x00037400
raw-size (278528 bytes)	0x00029000 167936 bytes)	0x0000BA00 (47616 bytes)	0x00002600 728 bytes)	0x0000D000 53248 bytes)
virtual-address	0x00401000	0x0042A000	0x00436000	0x0043C000
virtual-size (292542 bytes)	0x00028E32 (167474 bytes)	0x0000B9F4 (47604 bytes)	0x00005FA0 (24480 bytes)	0x0000CEF8 (52984 bytes)

Image shows different Emotet samples having sections with the same size and entropy

This also results in completely identical code. The two samples presented above were compared using the diaphora plugin for IDA, and all the functions were identical.



Line	Address	Name	Address 2	Name 2	Ratio	BBlocks 1	BBlocks 2	Description
00000	00401000	sub_401000	00401000	sub_401000	1.000	1	1	100% equal
00001	00401006	sub_401006	00401006	sub_401006	1.000	3	3	100% equal
00002	004010af	sub_4010AF	004010af	sub_4010AF	1.000	3	3	100% equal
00003	004010f1	sub_4010F1	004010f1	sub_4010F1	1.000	5	5	100% equal
00004	00401107	sub_401107	00401107	sub_401107	1.000	1	1	100% equal
00005	0040117a	sub_40117A	0040117a	sub_40117A	1.000	5	5	100% equal
00006	004011ae	sub_4011AE	004011ae	sub_4011AE	1.000	8	8	100% equal
00007	00401247	sub_401247	00401247	sub_401247	1.000	3	3	100% equal
00008	00401274	sub_401274	00401274	sub_401274	1.000	3	3	100% equal
00009	004012a1	sub_4012A1	004012a1	sub_4012A1	1.000	14	14	100% equal
00010	004013d0	sub_4013D0	004013d0	sub_4013D0	1.000	7	7	100% equal
00011	00401478	sub_401478	00401478	sub_401478	1.000	2	2	100% equal
00012	00401495	sub_401495	00401495	sub_401495	1.000	5	5	100% equal
00013	004014ed	sub_4014ED	004014ed	sub_4014ED	1.000	4	4	100% equal
00014	00401526	sub_401526	00401526	sub_401526	1.000	2	2	100% equal
00015	00401675	sub_401675	00401675	sub_401675	1.000	9	9	100% equal
00016	004016e4	sub_4016E4	004016e4	sub_4016E4	1.000	3	3	100% equal
00017	00401733	sub_401733	00401733	sub_401733	1.000	3	3	100% equal
00018	0040174f	sub_40174F	0040174f	sub_40174F	1.000	1	1	100% equal

Comparing the code of the two samples shows they share the exact same subroutines

Grouping the entire dataset by the size of the files resulted in 272 unique sizes of files. The files matching each size were then checked to see if they have the same static information. This way we discovered 102 templates of Emotet samples.

Each sample in the dataset that matched a template was tagged with a template ID and with an epoch number, as indicated by the Cryptolaemus group. This means that the operators behind each epoch have their own Emotet loaders. The various templates might help reduce the detection rate of the samples used by the entire operation. If a specific template has a unique feature that can be signed, it won't affect samples belonging to other templates and epochs.

Most packers today provide features such as various encryption algorithms, integrity checks and evasion techniques. These templates are most likely the result of different combinations of flags and parameters in the packing software. Each epoch has different configurations for the packing software resulting in clusters of files that have the same static information.

Identifying benign code

The Emotet loader contains a lot of benign code as part of its evasion. A.I. based security products rely on both malicious and benign features when classifying a file. In order to bypass products like that, malware authors can insert benign code into their executable files to reduce the chance of them being detected. This is called an adversarial attack and its effectiveness is seen in security solutions based on machine learning.

By looking at the analysis done by Intezer on a specific Emotet sample from the new wave, we can see that the benign code might be taken from Microsoft DLL files that are part of the Visual C++ runtime environment. Alternatively, the benign software could be completely unrelated to the functioning of the sample.

The following screenshot shows the similarities between the previous sample and a benign Microsoft DLL file, using the diaphora plugin:

IDA View-A | Unmatched in secondary | Unmatched in primary | Unreliable matches | Partial matches | Best matches | Hex View-1 | Structures | Enums | Imports | Exports

Line	Address	Name	Address 2	Name 2	Ratio	BBlocks 1	BBlocks 2	Description
00046	0041255b	sub_41255B	7863620e	public: int CMapPtrToPtr::Lookup(void near *,void near * near &)const	0.950	3	3	Mnemonics small-primes-product
00033	0040d45c	sub_40D45C	78644f73	public: virtual void near * CMDIFrameWnd::`vector deleting destructor'(uns...	0.940	3	3	Mnemonics small-primes-product
00034	0040d54a	sub_40D54A	786a23f2	public: virtual void near * ATL::CFixedStringT<class ATL::CStringT<wchar_t,...	0.940	3	3	Mnemonics small-primes-product
00035	0040de61	sub_40DE61	786a2418	public: virtual void near * ATL::CFixedStringT<class ATL::CStringT<char,clas...	0.940	3	3	Mnemonics small-primes-product
00036	0040ef33	sub_40EF33	786a4c5a	public: virtual void near * COleObjectFactory::`vector deleting destructor'(u...	0.940	3	3	Mnemonics small-primes-product
00037	0041070f	sub_41070F	786ba221	public: virtual void near * COleControlSite::`scalar deleting destructor'(unsi...	0.940	3	3	Mnemonics small-primes-product
00038	00414de7	sub_414DE7	786c78b0	public: virtual void near * COleControl::XPropConnPt::`scalar deleting destr...	0.940	3	3	Mnemonics small-primes-product
00039	00415001	sub_415001	786c7fe1	public: virtual void near * COleControl::CControlDataSource::`scalar deletin...	0.940	3	3	Mnemonics small-primes-product
00040	0041527f	sub_41527F	786d745f	public: virtual void near * CWindowlessDC::`scalar deleting destructor'(unsi...	0.940	3	3	Mnemonics small-primes-product
00041	0041615b	sub_41615B	786e3ebe	public: virtual void near * CMFCTabDropTarget::`scalar deleting destructor'(...	0.940	3	3	Mnemonics small-primes-product
00042	00428286	sub_428286	786f1ed9	public: virtual void near * CMap<unsigned long,unsigned long,class ATL::C...	0.940	3	3	Mnemonics small-primes-product
00001	0040b7b9	sub_40B7B9	7863b7ea	public: virtual void CWnd::WinHelpA(unsigned long,unsigned int)	0.920	9	9	Same constants
00043	004282cf	sub_4282CF	786438db	public: virtual void near * CList<struct HWND__ near *,struct HWND__ near ...	0.890	3	3	Mnemonics small-primes-product
00044	0041b2e0	sub_41B2E0	7864b52d	public: virtual void near * CArray<int,int const near &>::`scalar deleting des...	0.890	3	3	Mnemonics small-primes-product
00045	0042831e	sub_42831E	7865cf6e	public: virtual void near * CCommonDialog::`vector deleting destructor'(un...	0.890	3	3	Mnemonics small-primes-product
00000	00428508	sub_428508	787f179d	public: ATL::CAtlBaseModule::CAtlBaseModule(void)	0.850	3	3	Same constants
00005	00403495	sub_403495	7862f7f1	public: class ATL::CSimpleStringT<char,1> near & ATL::CSimpleStringT<cha...	0.850	6	6	Same rare KOKA hash
00002	0040b829	sub_40B829	7863b85d	public: virtual void CWnd::HtmlHelpA(unsigned long,unsigned int)	0.840	9	9	Same constants
00006	00410f56	sub_410F56	78657199	class ATL::CStringT<char,class StrTraitMFC_DLL<char,class ATL::ChTraitsCRT...	0.830	7	7	Same rare KOKA hash
00003	004131d2	sub_4131D2	78630797	public: virtual int CArchiveException::GetErrorMessage(char near *,unsigne...	0.810	17	14	Same constants
00004	00428579	sub_428579	787f0925	__delayLoadHelper2(x,x)	0.800	37	37	Same constants
00007	00403336	sub_403336	786379c9	private: static struct ATL::CStringData near * ATL::CSimpleStringT<wchar_t,1...	0.800	7	7	Same rare KOKA hash
00028	00409691	sub_409691	78699a29	public: static class CNoTrackObject near * CProcessLocal<class _AFX_OLE_S...	0.800	3	3	Mnemonics small-primes-product
00023	004146f0	sub_4146F0	78631f07	public: ATL::CStringT<char,class StrTraitMFC_DLL<char,class ATL::ChTraitsC...	0.780	7	7	Same MD Index and constants

The Emotet loader contains benign code taken from a Microsoft DLL

Under the column “Name” there are functions from the malware, and under the column “Name 2” there are functions from the benign file. As we can see, the malware contains much benign code which isn’t necessarily needed.

The next step is to check how much of the code is actually used. This can be done using the tool drcov by DynamoRIO. This tool executes binary file and tracks which parts of the code are used. The log produced by this tool can later be processed by the lighthouse plugin for IDA. This plugin integrates the execution log into the IDA database in order to visualize which functions are used. The analysis was performed on the sample shown so far, the result is that just 16.22% of the code is executed

Lighthouse Coverage Report

- **Target Binary:** 25a2d28c7f6d35f8b78c33c00fdc469b44235338b82c527a1de924e56a7536bc
- **Coverage Name:** drcov.04416.0000.proc.log
- **Coverage File:** C:/Users/user/Desktop/emotet/drcov.04416.0000.proc.log
- **Database Coverage:** 16.22%
- **Table Coverage:** 16.22%
- **Timestamp:** Mon Aug 3 14:23:51 2020

Cov %	Func Name	Address	Blocks Hit	Instr. Hit	Func Size
0.00	sub_401000	0x401000	0 / 1	0 / 2	6
0.00	sub_401006	0x401006	0 / 3	0 / 6	14
0.00	unknown_libname_11	0x401014	0 / 3	0 / 5	24
100.00	ATL::CStringData::Release(void)	0x40102D	3 / 3	12 / 12	26
0.00	std::allocator::allocate(uint)	0x401047	0 / 1	0 / 4	13
0.00	CDC::Rectangle(tagRECT const *)	0x401054	0 / 1	0 / 8	27
0.00	CWnd::SendMessageA(uint,uint,long)	0x40106F	0 / 1	0 / 6	24
100.00	CWnd::GetWindowRect(tagRECT *)	0x401087	1 / 1	4 / 4	16
0.00	CWnd::BeginModalState(void)	0x401097	0 / 1	0 / 4	12
0.00	CWnd::EndModalState(void)	0x4010A3	0 / 1	0 / 4	12
0.00	sub_4010AF	0x4010AF	0 / 3	0 / 11	28
83.33	ATL::CSimpleStringT::SetLength(int)	0x4010CB	3 / 4	10 / 12	37

The report produced by the lighthouse plugin showing which functions were executed

After we filtered out benign code that was injected into the executable, we can compare the code of different variants to locate the malicious functions which exist in every sample.

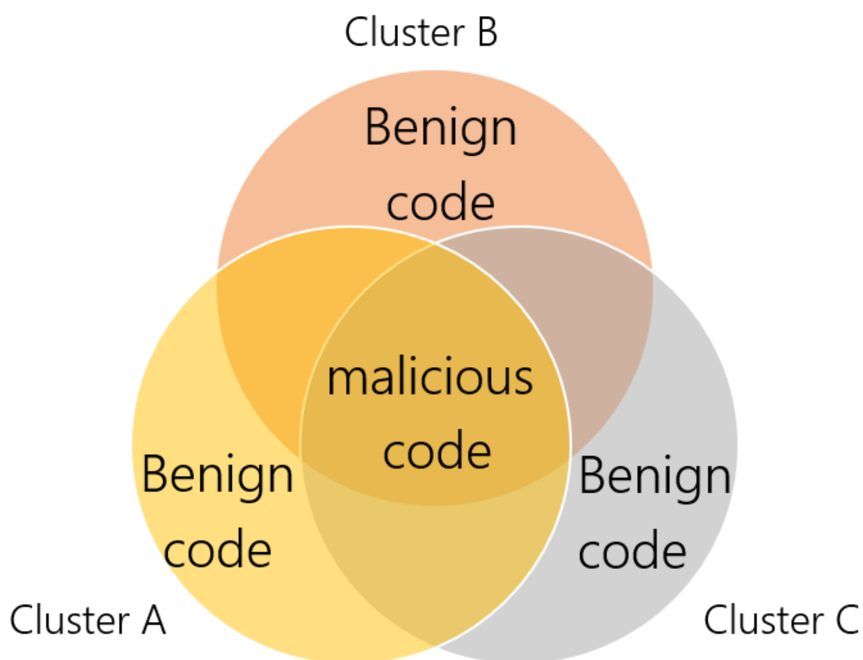


Diagram shows code from different clusters sharing the same malicious functionality.

Filtering out benign functions helps to reveal the malicious code, but it can also be found using dynamic analysis, which will be discussed in the next section.

Finding the encrypted payload

The executable previously shown is an Emotet loader. The main purpose of the loader is to decrypt the payload hidden in the sample and execute it. The payload consists of a PE file and a shellcode that loads it. The encrypted payload will cause the section it resides in to have high entropy. Based on the dataset we collected, 87% of the files had the payload in the .data section and 13% of the files had the payload in the .rsrc section. Tools like pestudio show the entropy of each section and each resource. For example, the resources of a sample with the payload lies encrypted in the “9248” resource:

In order to find the code that decrypts the resource, we can put a hardware breakpoint at the start of the resource.

type (10)	name	file-offset (57)	signature	non-standard	size (49968 bytes)	file-ratio (17.87%)	md5	entropy
rcdata	9248	0x00040004	unknown	-	35140	12.57 %	386C8263468884C2C4B3B316B28830D9	7.985
manifest	1	0x00048D9C	manifest	-	346	0.12 %	24D3B502E1846356B0263F945DDD5529	4.796
cursor	15	0x0003D9A0	cursor	-	308	0.11 %	FF43EAAB521694D0356618A92CD83B55	3.815
dialog	100	0x0003E47C	dialog	-	294	0.11 %	7CEE385DDA07AD97A5D832557D2409BB	3.430
version	1	0x00048AA8	version	-	756	0.27 %	545CE9C33E67114715846E4837B82F10	3.349

Pestudio showing the resources with the highest entropy in the Emotet loader

In cases where the payload is inside the .data section, it’s unclear where it starts. We can approximate it by calculating the entropy for small bulks of the section and find where the score starts to rise.

```
<span style="color: #0000ff;">import</span> sys
<span style="color: #0000ff;">import</span> math
<span style="color: #0000ff;">import</span> pefile

BULK_SIZE = 256

<span style="color: #0000ff;">def</span> <span style="color: #00ccff;">entropy</span>
(byteArr):
       arrSize = <span style="color: #0000ff;">len</span>(byteArr)
       freqList = []
       <span style="color: #0000ff;">for</span> b in <span style="color:
#0000ff;">range</span>(256):
             ctr = 0
            <span style="color: #0000ff;"> for</span> byte in byteArr:
                    <span style="color: #0000ff;">if</span> byte == b:
                           ctr += 1
             freqList.append(<span style="color: #00ccff;">float</span>(ctr) /
arrSize)
       ent = 0.0
       <span style="color: #0000ff;">for</span> freq in freqList:
           <span style="color: #0000ff;">if</span> freq > 0:
               ent = ent + freq * math.log(freq, 2)
       ent = -ent
       <span style="color: #0000ff;">return</span> ent
pe_file = pefile.PE(sys.argv[1])
data_section = <span style="color: #0000ff;">next</span>(section <span style="color:
#0000ff;">for</span> section in pe_file.sections <span style="color:
#0000ff;">if</span> section.Name == b<span style="color:
#993300;">'.data\x00\x00\x00'</span>)
data_section_buffer = data_section.get_data()
data_section_va = pe_file.OPTIONAL_HEADER.ImageBase + data_section.VirtualAddress
buffer_size = <span style="color: #0000ff;">len</span>(data_section_buffer)
bulks = [data_section_buffer[i:i+BULK_SIZE] <span style="color: #0000ff;">for</span>
i in <span style="color: #0000ff;">range</span>(0, buffer_size, BULK_SIZE)]
<span style="color: #0000ff;">for</span> i, bulk in <span style="color:
#0000ff;">enumerate</span>(bulks):
       <span style="color: #0000ff;">print</span>(<span style="color:
#0000ff;">hex</span>(data_section_va + i*BULK_SIZE), entropy(bulk))
```

Python script that calculates the entropy of small portions of the .data section

Once we know where the payload is located, we’ll be able to find the code that decrypts it, and that is where the malicious action starts.

Analyzing the malicious code

For this part, we’ll look at the sample:

249269aae1e8a9c52f7f6ae93eb0466a5069870b14bf50ac22dc14099c2655db.

In this sample, the script indicates that the beginning of the data section contains the payload although it may vary in other samples. We will put the breakpoint at the address 0x406100. The breakpoint was hit at the address 0x40218C which is in the function sub_401F80. After looking at this function, we notice a few suspicious things:

1. This function builds strings on the stack in order to hide its intents. It uses GetProcAddress to find the address of VirtualAllocExNuma and calls it to allocate memory for the payload.

```
mov     [esp+68h+ProcName], 56h ; 'V'
mov     [esp+68h+var_1F], 69h ; 'i'
mov     [esp+68h+var_1E], 72h ; 'r'
mov     [esp+68h+var_1D], 74h ; 't'
mov     [esp+68h+var_1C], cl
mov     [esp+68h+var_1B], 61h ; 'a'
mov     [esp+68h+var_19], 41h ; 'A'
mov     [esp+68h+var_16], 6Fh ; 'o'
mov     [esp+68h+var_15], 63h ; 'c'
mov     byte ptr [esp+68h+var_14], 45h ; 'E'
mov     byte ptr [esp+68h+var_14+1], 78h ; 'x'
mov     byte ptr [esp+68h+var_14+2], 4Eh ; 'N'
mov     byte ptr [esp+68h+var_14+3], cl
mov     byte ptr [esp+68h+var_10], 6Dh ; 'm'
mov     byte ptr [esp+68h+var_10+1], 61h ; 'a'
mov     byte ptr [esp+68h+var_10+2], bl
mov     [esp+68h+LibFileName], 6Bh ; 'k'
mov     [esp+68h+var_3F], 65h ; 'e'
mov     [esp+68h+var_3E], 72h ; 'r'
mov     [esp+68h+var_3D], 6Eh ; 'n'
mov     [esp+68h+var_3C], 65h ; 'e'
mov     [esp+68h+var_3A], 33h ; '3'
mov     [esp+68h+var_39], 32h ; '2'
mov     [esp+68h+var_38], 2Eh ; '.'
mov     [esp+68h+var_37], 64h ; 'd'
mov     [esp+68h+var_34], bl
call    esi ; LoadLibraryExA
lea     ecx, [esp+5Ch+ProcName]
push    ecx             ; lpProcName
push    eax             ; hModule
call    ds:GetProcAddress
```

The loader conceals suspicious API calls

2. It calculates the parameters for VirtualAllocExNuma during runtime, to hide the allocation of RWX memory. The function atoi is being used to convert the string “64” to int, which is PAGE_EXECUTE_READWRITE. Also, the string “8192” is converted to 0x3000 which means the memory is allocated with the flags MEM_COMMIT and MEM_RESERVE.

```
mov     esi, ds:atoi
push    ebx
push    offset String   ; "64"
call    esi ; atoi
add     esp, 4
push    eax
push    offset a8192    ; "8192"
call    esi ; atoi
add     esp, 4
or      ah, 10h
push    eax
push    8544h
push    ebx
call    ds:GetCurrentProcess
push    eax
call    edi
```

The parameters were saved as strings to obfuscate the API call

The payload is then copied from the .data section to the RWX memory (that is where our breakpoint hit). The decryption routine is being called and then the shellcode is being executed.


```
mov     esi, ds:atoi
push    ebx
push    offset String   ; "64"
call    esi ; atoi
add     esp, 4
push    eax
push    offset a8192    ; "8192"
call    esi ; atoi
add     esp, 4
or      ah, 10h
push    eax
push    8544h
push    ebx
call    ds:GetCurrentProcess
push    eax
call    edi
```


The loader decrypts the payload and continues to the next step in the execution of the malware

In this blog post we looked at the static information of the new Emotet loader, revealed how to cluster similar samples, and found how to locate the malicious code and its payload. In addition, we exposed how the loader evades detection; primarily the loader hides the malicious API calls using obfuscation, but it also injects benign code to manipulate the algorithms of AI-based security products. Both processes have been shown to reduce the chance of the file being detected. The cumulative effect of these techniques makes the Emotet group one of the most advanced campaigns in the threat landscape.

Update

Read more about the hidden payload in the Emotet loader that is decrypted and then executed to successfully avoid being detected.