

**Topic:-** Emotet Malware Analysis

**Malware Hash:-**


- sha256:- ae5de878deeb48308865377d6a71a769dbf74a06985fa7be19ebdb7a85ed316b

**Tools:-** x64dbg, Cutter, Process Hacker, Wireshark





**Overview:-** Emotet is a trojan primarily used in phishing attacks. It spreads mainly through email containing links or malicious attachments which act as an initial vector into the machine. While emerging in 2014 as a simple trojan with worm like capabilities, the malware has seen a resurgence in the past year and is mostly being used as as dropper or downloader of other malware. In this write-up we take a look at the various functionalities of the malware such as encryption, API address resolution and identify IOCs and IOAs.

## CHECKING MALWARE SANDBOXES

Taking a look at the Virus Total, we can see that it 3/4th of the vendors detect the the file as malicious. But if we take a look at the Intezer analysis we can see that it is not able to detect techniques or behaviour of the malware considering the malware has a lot of junk code and uses hashes for resolving API address dynamically.



ae5de878deeb48308865377d6a71a769dbf74a06985fa7be19ebdb7a85ed316b



55

/ 69

?

Community Score

55 security vendors and 2 sandboxes flagged this file as malicious

ae5de878deeb48308865377d6a71a769dbf74a06985fa7be19ebdb7a85ed316b

Sokoban.exe

pedll

1.02 MB  
Size


2022-04-04 10:59:36 UTC  
2 days ago

DETECTION	DETAILS	RELATIONS	BEHAVIOR	COMMUNITY <span>2</span>
Ad-Aware	<span>🚩</span> Trojan.GenericKD.48506126	AhnLab-V3	<span>🚩</span> Trojan.Win.BotX-gen.R476070	
Alibaba	<span>🚩</span> Trojan:Win32/Emotetcrypt.995c0cd5	ALYac	<span>🚩</span> Trojan.Agent.Emotet	
Antiy-AVL	<span>🚩</span> Trojan/Generic.ASMalwS.355402E	Arcabit	<span>🚩</span> Trojan.Generic.D2E4250E	
Avast	<span>🚩</span> Win32:BotX-gen [Trj]	AVG	<span>🚩</span> Win32:BotX-gen [Trj]	
Avira (no cloud)	<span>🚩</span> TR/AD.Nekark.tcmco	BitDefender	<span>🚩</span> Trojan.GenericKD.48506126	

INTEZER ANALYZE

HomeAPIDocsIntegrationsPluginsAnalysis ReportsSHA256 / SHA1 / MD5

Upgrade to the premium edition

 **Malicious**  
Main Family: Emotet

0110 SHA256  
ae5de878deeb48308865377d6a71a769dbf74a06985fa7be19ebdb7a85ed316b  
VIRUSTOTAL Report (32 / 69 Detections)  
pe dll i386 Armadillo

Malicious. This file contains code therefore it's very likely to be malicious.

Genetic Analysis

TTPs

IOCs

Behavior

Detect & Hunt BETA

MITRE ATT&CK Technique Detection

No matches found for MITRE ATT&CK

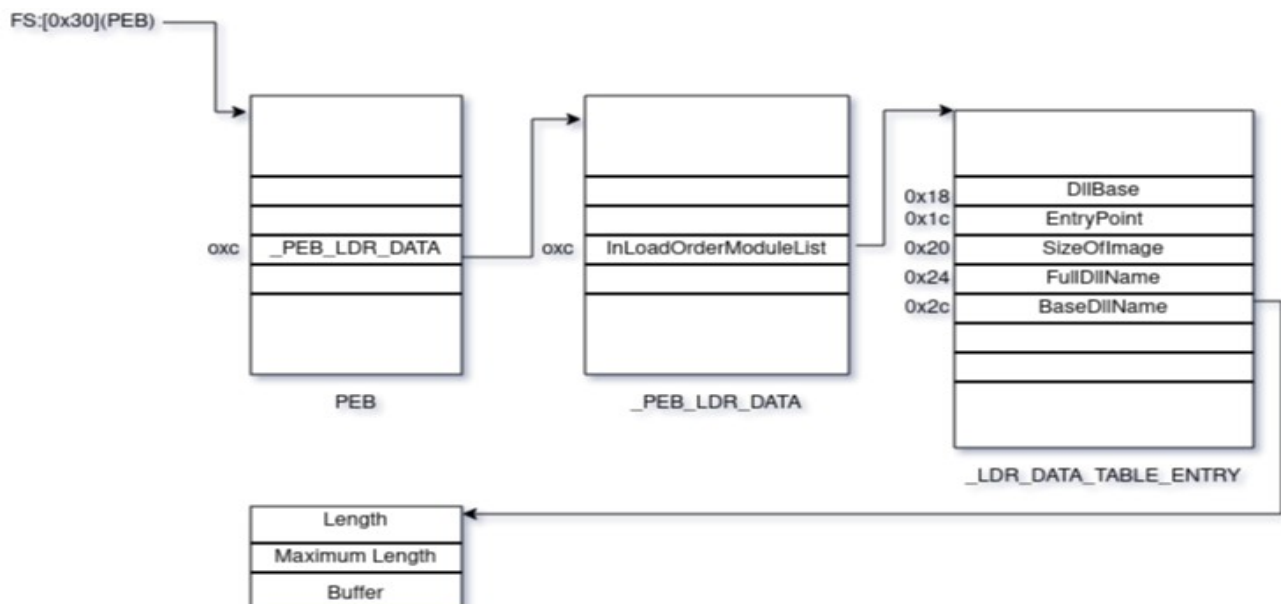
MITRE ATT&CK	Technique	Severity	Details
-	Behavioural detection: Executable code extraction - unpacking	Low	-

## RESOLVING API ADDRESSES DYNAMICALLY

The malware uses hashes for both DLL names and the APIs to be used.

```
Graph (fcn.1000400f)
fcn.1000400f(int32_t arg_8h, int32_t arg_ch, int32_t arg_10h, int32_t arg_14h, int32_t arg_18h, int32_t arg_1ch);
0x10004051 51      push ecx
0x10004052 6845020000 push 0x245 ; 581
0x10004057 b9369a1c3c mov ecx, 0x3c1c9a36 ; hash of the dll name
0x1000405c 8945fc    mov dword [var_4h], eax
0x1000405f 8175fc75f4afab xor dword [var_4h], 0xabaff475
0x10004066 8175fc8fdf291e xor dword [var_4h], 0x1e29df8f
0x1000406d c745f0d0dcee00 mov dword [var_10h], 0xeedcd0
0x10004074 8175f01f72f4c4 xor dword [var_10h], 0xc4f4721f
0x1000407b 8175f097ed18c4 xor dword [var_10h], 0xc418ed97
0x10004082 c745f465a3d000 mov dword [var_ch], 0xd0a365
0x10004089 8145f4aefd0000 add dword [var_ch], 0xfdae
0x10004090 8175f428acd200 xor dword [var_ch], 0xd2ac28
0x10004097 c745f80e327300 mov dword [var_8h], 0x73320e
0x1000409e 8175f86448ad2f xor dword [var_8h], 0x2fad4864
0x100040a5 8175f8b285de2f xor dword [var_8h], 0x2fde85b2
0x100040ac 8b45f8     mov eax, dword [var_8h]
0x100040af 8b45f4     mov eax, dword [var_ch]
0x100040b2 8b45f0     mov eax, dword [var_10h]
0x100040b5 8b45fc     mov eax, dword [var_4h]
0x100040b8 e8337f0100 call fcn.1001bff0 ; retrieve api address
0x100040bd 83c414     add esp, 0x14
0x100040c0 56        push esi
0x100040c1 ff7510     push dword [arg_10h]
0x100040c4 6aff      push 0xffffffffffffffff
0x100040c6 ff7518     push dword [arg_18h]
0x100040c9 57        push edi
0x100040ca ffd0      call eax
```

First the matching DLL is found. The below diagram shows how the malware retrieves the DLL names.

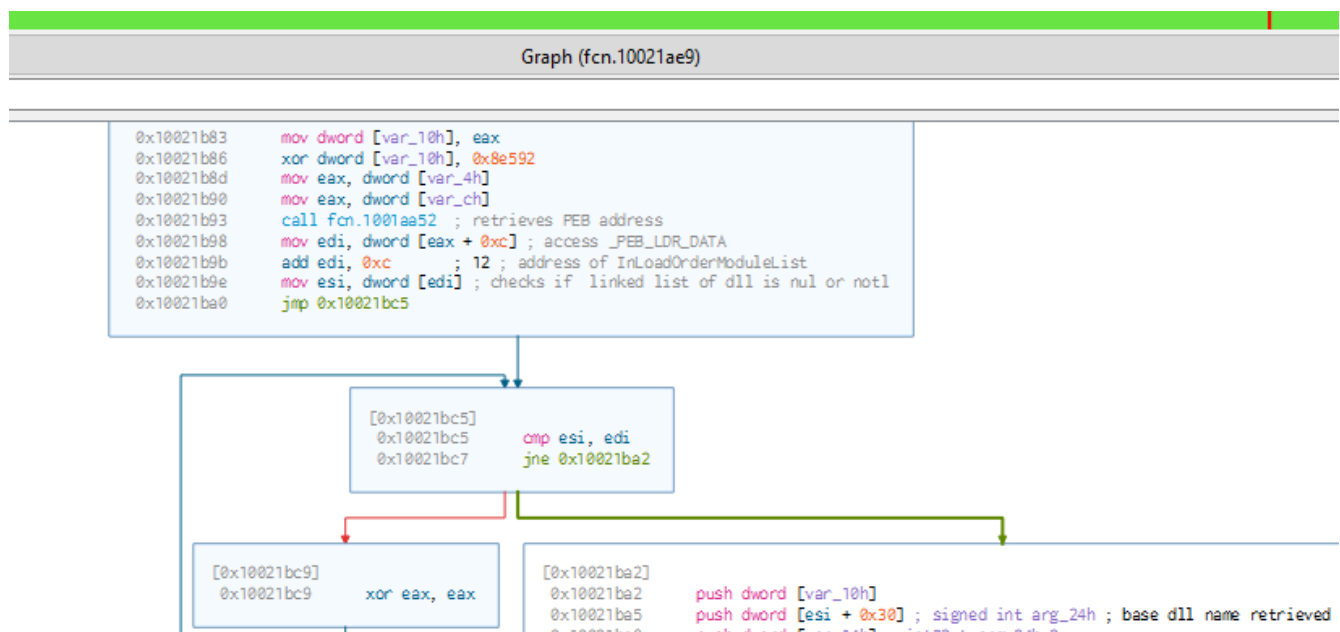


The malware iterates through `LDR_DATA_TABLE_ENTRY` structures for different DLLs to find the DLL name, calculates a hash of the name and matches the hash to the stored hash. If the hash matches the base address of the DLL is retrieved.

The malware then traverses the DLL to the **IMAGE\_EXPORT\_DIRECTORY** structure of the PE file which holds the fields **Address of Names**, **Address of Functions**, **Address of Name Ordinals**.

The Address of Names is an array of pointers, these pointers point to the names of the APIs. The malware calculates the hash based on the names of the APIs and matches it with the stored hash. If there is a match the corresponding address of the function name is retrieved from the Address of Functions array. For more detailed explanation on this check my article here [“How Malware Resolves API address dynamically”](#). The image below gives shows how this functionality is implemented.

```
[0x1001aa52]
7: fcn.1001aa52 ();
0x1001aa52 64a130000000 mov eax, dword fs:[0x30]
0x1001aa58 c3 ret
```



The algorithm used to create a hash of a particular API name and compare it later with the stored hash is given below. Since the malware code contains the lot of junk code, the code algorithm here is cleaned up for understanding.

```
int CalculateHashOfAPI(int PointerToAPIName, int junk_variable, int junk_variable)
{
    int temp1 = PointerToAPIName; //Pointer to API name
    PointerToAPIName = 0xeadf60;
    PointerToAPIName += 0x5fa9;
    PointerToAPIName = PointerToAPIName / 0x25;
    PointerToAPIName += 0xffffd8b5;
    PointerToAPIName ^= 0x6345b;
    if (*temp1 != 0)
```

```
{  
    do  
    {  
        int temp2 = PointerToAPIName;  
        int temp3 = PointerToAPIName;  
        temp3 <<= 0x6;  
        int temp4 = PointerToAPIName;  
        temp4 <<= 0x10;  
        PointerToAPIName = *temp1;           // store character  
        PointerToAPIName += temp3;  
        PointerToAPIName += temp4;  
        PointerToAPIName -= temp2;  
        temp1++;  
  
    }while(*temp1 != 0)  
  
    }  
    else  
    {  
        return PointerToAPIName;           // stores calculated hash  
    }  
}
```

## Acquiring Data and Encryption

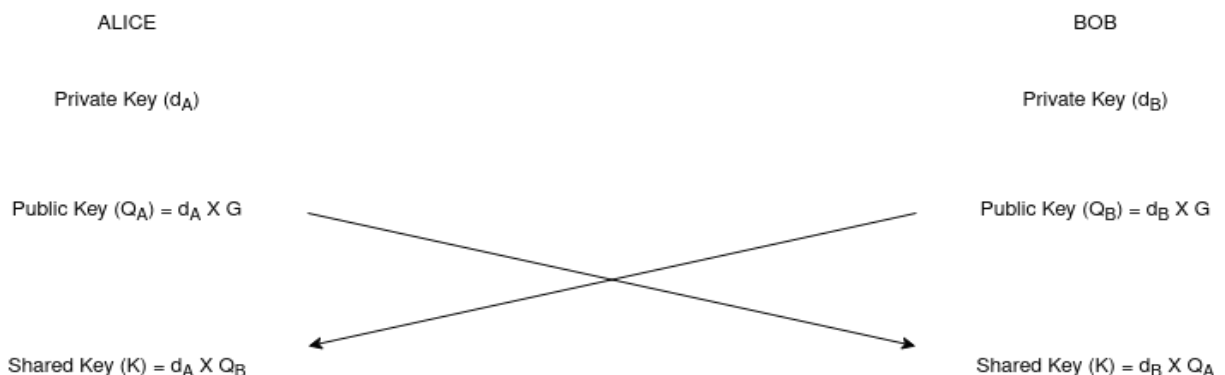
The malware obtains information using various APIs. It encrypts the data and tries to communicate it over to the C2 server. The APIs used for cryptographic operations are given below.

- <bcrypt.BCryptOpenAlgorithmProvider>
- <bcrypt.BCryptCloseAlgorithmProvider>
- <bcrypt.BCryptGenerateKeyPair>
- <bcrypt.BCryptFinalizeKeyPair>
- <bcrypt.BCryptExportKey>
- <bcrypt.BCryptImportKeyPair>
- <bcrypt.BCryptSecretAgreement>
- <bcrypt.BCryptDeriveKey>
- <bcrypt.BCryptImportKey>
- <bcrypt.BCryptDestroySecret>
- <bcrypt.BCryptDestroyKey>
- <bcrypt.BCryptHashData>
- <bcrypt.BCryptFinishHash>
- <bcrypt.BCryptDestroyHash>

The Algorithm used by the malware are:

- Elliptic-curve Diffie-hellman (ECDH) :- For key generation and shared secret key.
- Advanced Encryption Standard (AES) :- For encrypting communication with C2 server.
- SHA256 :- For Hashing of data to be send to C2 server.

Below diagram shows the working of the ECDH algorithm where G represents the generator, d the private keys, Q the Public Keys. The operation 'X' represents a scalar multiplication. The public key is an (x,y) point on the elliptic curve.



The article won't go into the mathematical details of the algorithm but get an idea in relation to the microsoft APIs we need to know how APIs accomplish the above steps.

Using the **BcryptGenerateKeyPair** API gives us the Public/Private Key pair. The shared key is then derived using the API **BcryptSecretAgreement**. The **BCryptDeriveKey** API is used to obtain the final key. The main task is to **create a shared secret that is common to both the Server and the infected machine in this case**. This

shared secret is then used as input to the AES algorithm, which then outputs the final symmetric key that will be used for encrypting messages.

## ENCRYPTION OPERATIONS

BCryptOpenAlgorithmProvider API initializes a CNG(Cryptography API: Next Generation) provider. The algorithm used by emotet malware in this case is ECDH\_P256 which is Elliptic-curve Diffie-hellman algorithm.

The screenshot shows the Immunity Debugger interface with the 'Memory Map' tab selected. The assembly window displays the following instructions:

```
1mul eax,dword ptr ss:[ebp-8],A
push F45DCDC9
push ecx
push ecx
push 346
mov dword ptr ss:[ebp-8],eax
mov ecx,45413672
add dword ptr ss:[ebp-8],FFFF6F2F
xor dword ptr ss:[ebp-8],2B73330
mov eax,dword ptr ss:[ebp-8]
mov eax,dword ptr ss:[ebp-8]
mov eax,dword ptr ss:[ebp-4]
mov eax,dword ptr ss:[ebp-4]
mov eax,dword ptr ss:[ebp-10]
call emotet_00f20000_pe_payload.71428FF0
add esp,14
push dword ptr ss:[ebp+8]
push esi
push dword ptr ss:[ebp+C]
push dword ptr ss:[ebp+14]
call eax
pop esi
```

The 'Registers' window shows the following values:

Register	Value	Comment
EAX	75F53D90	<bcrypt.BCryptOpenAlgorithmProvider>
EBX	0F04D6FD	
ECX	45413672	
EDX	0000004F	'o'
EBP	004AEDD4	
ESP	004AEDA8	'@ij'
ESI	028456F0	L"Microsoft Primitive Provider"
EDI	028456C0	L"ECDH_P256"
EIP	7141D753	emotet_00f20000_pe_payload.7141D753

The 'Stack' window shows the following stack frame:

Index	Address	Value
1	[esp] 004AEF40	
2	[esp+4] 028456C0	L"ECDH_P256"
3	[esp+8] 028456F0	L"Microsoft Primitive Provider"
4	[esp+C] 00000000	
5	[esp+10] 028456F0	L"Microsoft Primitive Provider"

The 'Registers' window also shows the following values:

Register	Value	Comment
EAX	75F53D90	<bcrypt.BCryptOpenAlgorithmProvider>
EBX	0F04D6FD	
ECX	45413672	
EDX	0000004F	'o'
EBP	004AEDD4	
ESP	004AEDA8	'@ij'
ESI	028456F0	L"Microsoft Primitive Provider"
EDI	028456C0	L"ECDH_P256"
EIP	7141D753	emotet_00f20000_pe_payload.7141D753

The handle of that is then passed to the BCRYPTGENERATEKEYPAIR which creates a public/private key pair. BCRYPTFINALIZEKEYPAIR API finalizes the key pair generated by BcryptGenerateKeyPair.

The screenshot shows the Immunity Debugger interface with the 'Memory Map' tab selected. The assembly window displays the following instructions:

```
8175 F4 C663FD2F xor dword ptr ss:[ebp-8],2FFD63C6
8175 F8 F4A78400 mov dword ptr ss:[ebp-8],84A7F4
8145 F8 EC130000 add dword ptr ss:[ebp-8],13EC
8145 F8 C8B3FFFF add dword ptr ss:[ebp-8],FFFFB3CB
8175 F8 4D9C8B00 xor dword ptr ss:[ebp-8],8B9C4D
8175 FC FE250500 mov dword ptr ss:[ebp-4],525FE
8175 FC 3195487A xor dword ptr ss:[ebp-4],7A489531
8175 FC 04 shl dword ptr ss:[ebp-4],4
8145 FC 6E610000 add dword ptr ss:[ebp-4],616E
8175 FC F930D8A4 xor dword ptr ss:[ebp-4],A4D830F9
8845 FC mov eax,dword ptr ss:[ebp-4]
8845 F8 mov eax,dword ptr ss:[ebp-8]
8845 F4 mov eax,dword ptr ss:[ebp-8]
8845 F0 mov eax,dword ptr ss:[ebp-10]
call emotet_00f20000_pe_payload.70A98FF0
add esp,14
push esi
push edi
push dword ptr ss:[ebp+18]
push dword ptr ss:[ebp+14]
call eax
pop edi
pop esi
mov eax,ah
```

The 'Registers' window shows the following values:

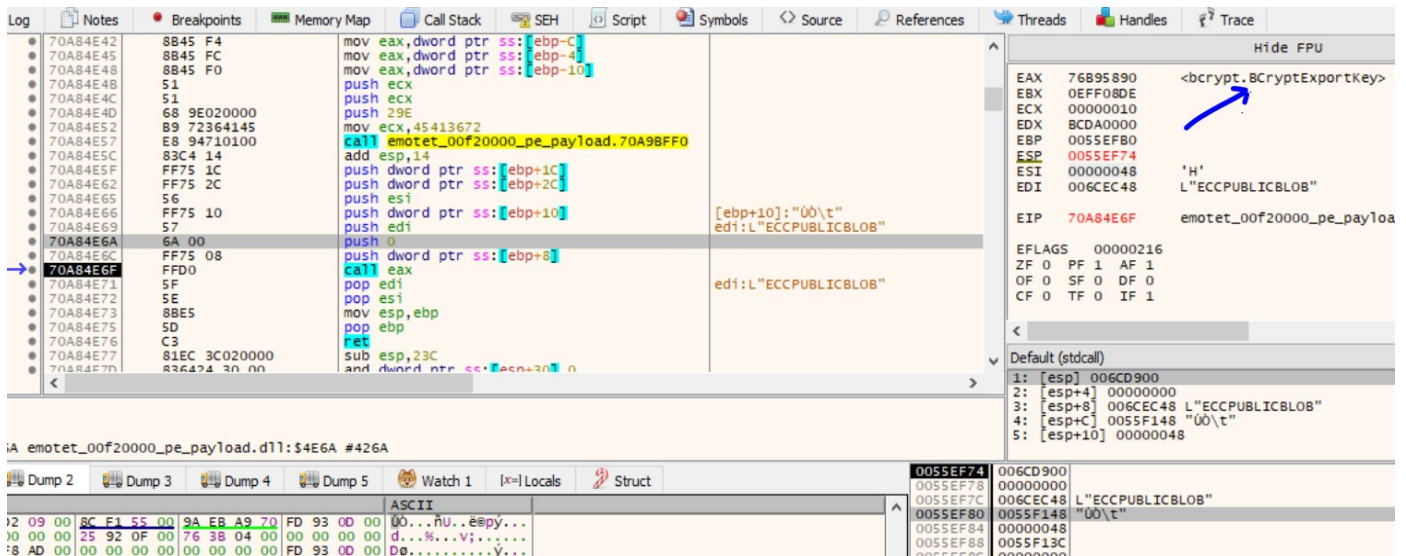
Register	Value	Comment
EAX	76895100	<bcrypt.BCryptGenerateKeyPair>
EBX	095D2264	
ECX	00000010	
EDX	89FA0000	
EBP	0055EFC4	
ESP	0055EF9C	'011'
ESI	00000000	
EDI	00000100	L'À'
EIP	70A8A014	emotet_00f20000_pe_payload.70A8A014

The 'Stack' window shows the following stack frame:

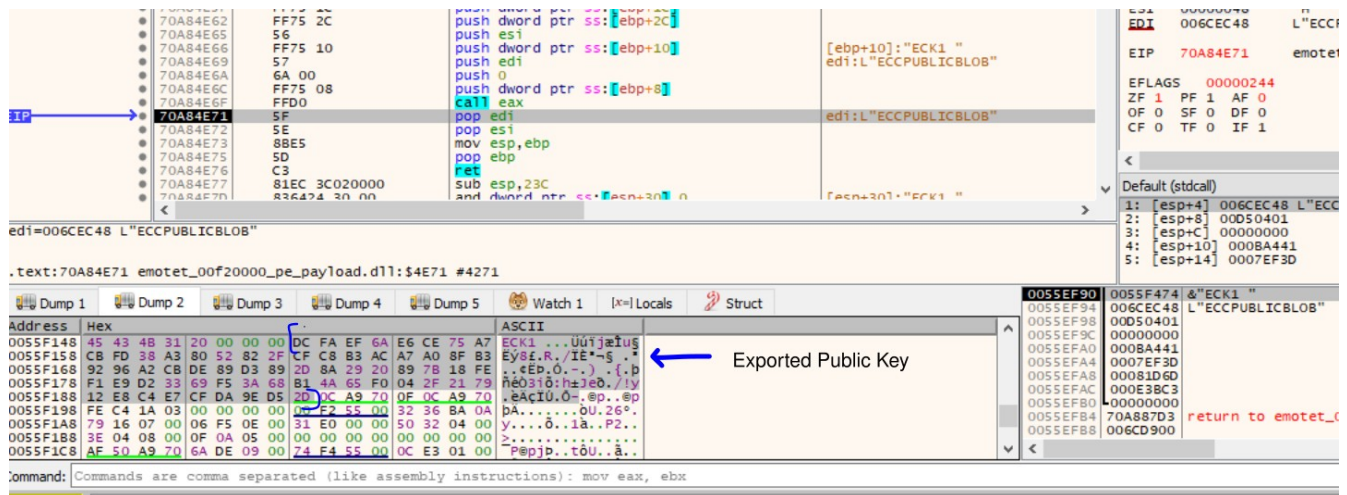
Index	Address	Value
1	[esp] 006CECF0	
2	[esp+4] 0055F12C	
3	[esp+8] 00000100	
4	[esp+C] 00000000	
5	[esp+10] 0055F474	&"ECK1 "



**BCryptExportKey** API is then used to export the Public key from the key pair to a **BCrypt\_ECCPUBLIC\_BLOB** structure. This public key can now be sent to the malicious server.



Exported Public Key inside the structure can be seen in the image below.



The Public Key of the Server is embedded in the malware inside a BCRYPT\_ECCPUBLIC\_BLOB structure. **BcryptImportKeyPair** API is used to obtain a handle to it.



The screenshot shows a debugger window with the following components:

- Assembly View:** Disassembled code starting at address 70A8B4C1. Instructions include `add dword ptr ss:[ebp-4],FFFFFFE83`, `xor dword ptr ss:[ebp-4],4FC1756`, `mov eax,dword ptr ss:[ebp-4]`, `mov eax,dword ptr ss:[ebp-C]`, `mov eax,dword ptr ss:[ebp-10]`, `mov eax,dword ptr ss:[ebp-8]`, `call emotet_00f20000_pe_payload.70A98FF0`, `add esp,14`, `push esi`, `push dword ptr ss:[ebp+28]`, `push dword ptr ss:[ebp+16]`, `push dword ptr ss:[ebp+20]`, `push dword ptr ss:[ebp+30]`, `push 0`, `push dword ptr ss:[ebp+2C]`, `call eax`, `pop esi`, `mov esp,ebp`, `pop ebp`, `ret`, `sub esp,58`, `mov dword ptr ss:[esp+48],9248BC`, `xor edx,edx`, and `mov dword ptr ss:[esp+4C],D1877`.
- Registers View:** Shows registers EAX, EBX, ECX, EDX, EBP, ESP, ESI, EDI, and EIP. EAX contains 76895FF0, EBX contains 06A948C9, ECX contains 00000010, EDX contains 842A0000, EBP contains 0055EFAC, ESP contains 0055EF7C, ESI contains 00000000, EDI contains 0055F474, and EIP contains 70A8B4F5.
- Memory View:** Shows a hex dump and an ASCII dump. The ASCII dump contains the text "ECK1 ...05.xc.PA" and "ECK1 ...05.xc.PA". A blue arrow points to the "Server Public Key" in the ASCII dump.
- Call Stack:** Shows the current function `emotet_00f20000_pe_payload.d11:$B4F5 #A8F5`.

The handle of the public/private key pair obtained using `BcryptGenerateKeyPair` and the handle of the public key of the Server obtained in the previous step is passed to the `BcryptSecretAgreement` API. This returns a handle to the shared secret.

The screenshot shows a debugger window with the following components:

- Assembly View:** Disassembled code starting at address 70A8BDD2. Instructions include `mov dword ptr ss:[ebp-10],7125E5`, `shl dword ptr ss:[ebp-10],6`, `xor dword ptr ss:[ebp-10],1C40AC4E`, `mov eax,dword ptr ss:[ebp-10]`, `mov eax,dword ptr ss:[ebp-4]`, `mov eax,dword ptr ss:[ebp-8]`, `mov eax,dword ptr ss:[ebp-C]`, `call emotet_00f20000_pe_payload.70A98FF0`, `add esp,14`, `push dword ptr ss:[ebp+C]`, `push esi`, `push dword ptr ss:[ebp+14]`, `push dword ptr ss:[ebp+18]`, `call eax`, `pop esi`, `mov esp,ebp`, `pop ebp`, `ret`, `sub esp,2C4`, `mov dword ptr ss:[esp+80],14AA91`, `xor edx,edx`, `push ebx`, `xor ebx,ebx`, and `mov ecx,dword ptr ss:[esp+8C]`.
- Registers View:** Shows registers EAX, EBX, ECX, EDX, EBP, ESP, ESI, EDI, and EIP. EAX contains 76896540, EBX contains 07A2A4C8, ECX contains 00000010, EDX contains 77660000, EBP contains 0055EFC0, ESP contains 0055EF9C, ESI contains 0055F138, EDI contains 0055F474, and EIP contains 70A8BE02.
- Memory View:** Shows a hex dump and an ASCII dump. The ASCII dump contains the text "ECK1 ...05.xc.PA" and "ECK1 ...05.xc.PA".
- Call Stack:** Shows the current function `emotet_00f20000_pe_payload.d11:$B8E02 #B202`.

The `BCryptOpenAlgorithmProvider` is then called to initialize the AES CNG provider. The handle obtained previously of the shared secret is passed to the `BCryptDeriveKey` API to obtain a final symmetric(same for both malware and server) key of size 32 bytes ie. 256 bit key.

**BCryptImportKey** is then used to obtain a handle to this key. This key will later be used to encrypt the messages sent to the server. All the other intermediate keys are destroyed using **BCryptDestroySecret** and **BCryptDestroyKey**.

Before the Data is to be sent. The hash of the data is calculated, in this case the data is the **Computer name** obtained using the API **GetComputerNameA** and the **serial number of the C Drive**, that is retrieved using the **GetVolumeInformation** API. Both are concatenated and sent to the server later after encryption.

The **BCryptAlgorithmProvider** is called to initialize the SHA256 CNG provider. **BCryptCreateHash** API is called to create a hash object and **BCryptHashData** API is used to hash the data as seen below.

The ComputerName and the hash are then encrypted using the **BCryptEncrypt** API.

The screenshot displays a debugger interface with the following components:

- Assembly Window:** Shows instructions starting at address 70A830B5. The instruction at 70A830B5 is `add esp, 20`. Other instructions include `push edi`, `call eax`, `pop edi`, `pop esi`, `mov esp, ebp`, `pop ebp`, `ret`, `push ebp`, `mov ebp, esp`, `sub esp, 20`, and several `and` and `mov` instructions for pointers and registers.
- Register Window:** Shows the state of registers: EAX (006D05F0), EBX (0055F47C), ECX (00000000), EDX (00000000), EBP (0055EF14), ESP (0055EEDC), ESI (006D0528), and EDI (006D05F0). The EIP register is highlighted at 70A830B5.
- Memory Dump Window:** Shows a dump of memory starting at address 006D0580. The dump is organized into columns for Address, Hex, and ASCII. Annotations with arrows point to specific data in the ASCII column:
  - Public key:** Points to the ASCII string `UuijæIu$Ey8f.R./` at address 006D0580.
  - Encrypted Data:** Points to the ASCII string `ÏE*-$. .cEb.0.` at address 006D0580.



## COMMUNICATION WITH C2 SERVER

The data after being converted is prepared to be sent to the server. The sequence of API's to establish connection to the server are:-

1. wininet.InternetOpenW
2. wininet.InternetConnectW
3. wininet.HttpOpenRequestW
4. wininet.InternetSetOptionW
5. wininet.InternetQueryOptionW
6. wininet.HttpSendRequestW

The malware first uses the **InternetOpenW** api to get a handle that is later used by other wininet apis.

The screenshot shows the Immunity Debugger interface with the disassembly window open. The CPU window displays the instruction list for the module `emotet_00f20000_pe_payload.dll`. The instruction at address `70A865CD` is `call emote_00f20000_pe_payload.70A98BF0`, which is highlighted. The right-hand pane shows the register values and the current instruction pointer (EIP) at `70A865CD`. The register values are: EAX: 74694850, EBX: 00000000, ECX: 00000010, EDI: 00000000, ESI: 00000000, EDI: 00000000. The instruction pointer (EIP) is `70A865CD`. The register values for the arguments of the `InternetOpenW` function are: EAX: 74694850, EBX: 00000000, ECX: 00000010, EDI: 00000000. The instruction pointer (EIP) is `70A865CD`.

The malware then calls the **InternetConnectW** api to start a session for the address 186.250.48.5 over port 50.

The screenshot shows the Immunity Debugger interface with the disassembly window open. The CPU window displays the instruction list for the module `emotet_00f20000_pe_payload.dll`. The instruction at address `70A85883` is `call emote_00f20000_pe_payload.70A98BF0`, which is highlighted. The right-hand pane shows the register values and the current instruction pointer (EIP) at `70A85883`. The register values are: EAX: 746C67A0, EBX: 00C00004, ECX: 00000010, EDI: 00000000, ESI: 00000000, EDI: 00000000. The instruction pointer (EIP) is `70A85883`. The register values for the arguments of the `InternetConnectW` function are: EAX: 746C67A0, EBX: 00C00004, ECX: 00000010, EDI: 00000000. The instruction pointer (EIP) is `70A85883`.

HttpOpenRequestW api is then used to create an Http request handle. The Get method is used in this case and the object name is **hugJqjMWzCo**. **HttpSendRequestW** method is used to send the request to the Malicious server. The Http additional header is where the malware stores and sends the data. In this case it is the data previously seen i.e **the Public key of the machine + Encrypted Data**. Everything is then converted to strings using the API **CryptBinaryToStringW**. Which is then appended to the header as the field named **Cookie** in this case.

The connection request from the malware to the C2 server can be seen below.

File Machine View Input Devices Help

Activities Wireshark Apr 1 03:15

Capturing from enp0s8

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
2437	2935.2594467...	10.0.1.6	186.250.48.5	TCP	66	[TCP Retransmission] 1463 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 W
2438	2937.2770541...	10.0.1.6	186.250.48.5	TCP	66	[TCP Retransmission] 1463 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 W
2439	2941.2906634...	10.0.1.6	186.250.48.5	TCP	66	[TCP Retransmission] 1463 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 W
2440	2941.9321650...	PcsCompu_3a:27:fa	PcsCompu_ed:eb:6e	ARP	60	Who has 10.0.1.2? Tell 10.0.1.6
2441	2941.9321784...	PcsCompu_ed:eb:6e	PcsCompu_3a:27:fa	ARP	42	10.0.1.2 is at 08:00:27:ed:eb:6e
2442	2949.3068261...	10.0.1.6	186.250.48.5	TCP	66	[TCP Retransmission] 1463 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 W
2443	2955.3268512...	10.0.1.6	168.119.39.118	TCP	66	1464 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM=1
2444	2956.3382900...	10.0.1.6	168.119.39.118	TCP	66	[TCP Retransmission] 1464 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=1460
2445	2958.3383746...	10.0.1.6	168.119.39.118	TCP	66	[TCP Retransmission] 1464 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=1460

Frame 2442: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface enp0s8, id 0

- Interface id: 0 (enp0s8)
- Encapsulation type: Ethernet (1)
- Arrival Time: Apr 1, 2022 03:15:25.103236764 EDT

0000	08 00 27 ed eb 6e 08 00 27 3a 27 fa 08 00 45 00	..'.n..':'.E.
0010	00 34 c9 31 40 00 80 06 3b 8d 0a 00 01 06 ba fa	.4.1@...;.....
0020	30 05 05 b7 00 50 e0 52 58 91 00 00 00 00 80 02	0....P.R X.....
0030	ff ff 3a 20 00 00 02 04 05 b4 01 03 03 08 01 01	...: ....
0040	04 02	..



## **INDICATORS OF COMPROMISE**

- **Host Based IOCs**
  - File:- ae5de878deeb48308865377d6a71a769dbf74a06985fa7be19ebdb7a85ed316b
- **Network Based IOCs**
  - 186.250.48.5
  - 168.119.39.118
  - 185.168.130.138
  - 190.90.233.66
  - 159.69.237.188
  - 54.37.228.122
  - 93.104.209.107
  - 185.148.168.15
  - 198.199.98.78
  - 87.106.97.83
  - 195.77.239.39
  - 37.44.244.177
  - 54.38.242.185
  - 185.184.25.78
  - 116.124.128.206
  - 139.196.72.155
  - 128.199.192.135
  - 103.41.204.169
  - 78.47.204.80
  - 68.183.93.250
  - 194.9.172.107

## **INDICATORS OF ATTACK**

- **Data exfiltration**
  - Acquires drive serial number, Computer name and Operating System Version etc to be sent to the C2 server.
- **C2 Communication**
  - Tries to connect to C2 server with periodic connection request.
- **Stealth**
  - Almost all APIs address are resolved dynamically
  - Junk code usage