Malware Analysis Spotlight: Emotet’s Use of Cryptography

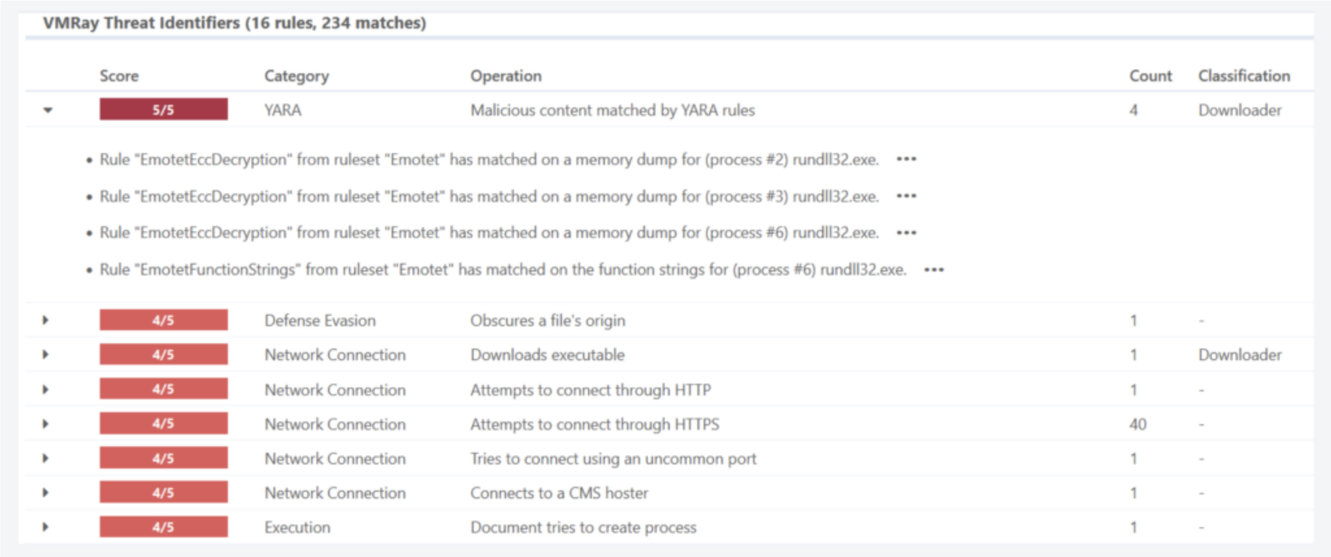
From the VMRay Labs Team
MALWARE ANALYSIS SPOTLIGHT: EMOTET
Examining Emotet’s Use of Cryptography

Emotet’s Use of Cryptography

Presented by the VMRay Labs Team

The group behind Emotet is the prime example of a very successful criminal enterprise. Emotet started out as a banking malware but over time evolved into a large botnet providing something akin to a malicious IaaS (Infrastructure-as-a-Service). It started providing access to its extensive list of infected devices to other threat actors and their malware (Trickbot, Dridex, IcedID). It started acting as their loader. Since the beginning of 2021, after a longer “break” which was the consequence of a coordinated take down of Emotet’s infrastructure by the law enforcement, Emotet resurfaced on the 14th of November 2021. Actively trying to rebuild its own infrastructure utilizing Trickbot. Many of the techniques stayed the same, but there are also some important differences.

The Emotet binaries, which were distributed starting from November 2021, come with two embedded elliptic-curve-based public keys of the server. The previous versions were using RSA as the primary asymmetric scheme. An RSA public key was embedded in the sample and used to encrypt the generated AES-128 key before sending it back to its C2. For message integrity, the packet was hashed with the SHA1 algorithm and the hash was appended to the request message. The new version comes with two public keys. One key is used for the **Elliptic Curve Diffie–Hellman (ECDH)** key exchange protocol while the other is used as part of the signature verification by the **Digital Signature Algorithm (DSA)**. In this blog post, we’ll be looking at how Emotet uses elliptic curve cryptography to protect the network communication and verify the authenticity and integrity of the commands received from its C2.



VMRay Threat Identifiers (16 rules, 234 matches)

	Score	Category	Operation	Count	Classification
▾	5/5	YARA	Malicious content matched by YARA rules	4	Downloader

- Rule "EmotetEccDecryption" from ruleset "Emotet" has matched on a memory dump for (process #2) rundll32.exe. •••
- Rule "EmotetEccDecryption" from ruleset "Emotet" has matched on a memory dump for (process #3) rundll32.exe. •••
- Rule "EmotetEccDecryption" from ruleset "Emotet" has matched on a memory dump for (process #6) rundll32.exe. •••
- Rule "EmotetFunctionStrings" from ruleset "Emotet" has matched on the function strings for (process #6) rundll32.exe. •••

▸	4/5	Defense Evasion	Obscures a file's origin	1	-
▸	4/5	Network Connection	Downloads executable	1	Downloader
▸	4/5	Network Connection	Attempts to connect through HTTP	1	-
▸	4/5	Network Connection	Attempts to connect through HTTPS	40	-
▸	4/5	Network Connection	Tries to connect using an uncommon port	1	-
▸	4/5	Network Connection	Connects to a CMS hoster	1	-
▸	4/5	Execution	Document tries to create process	1	-

View the Analysis

Background

Comparison: Past vs Present

Since the cryptographic part has changed in the newest version of Emotet we are providing a high level overview of the key steps taken by the older and new versions.

The previous version of Emotet that were using RSA roughly followed the following steps when encrypting a message:

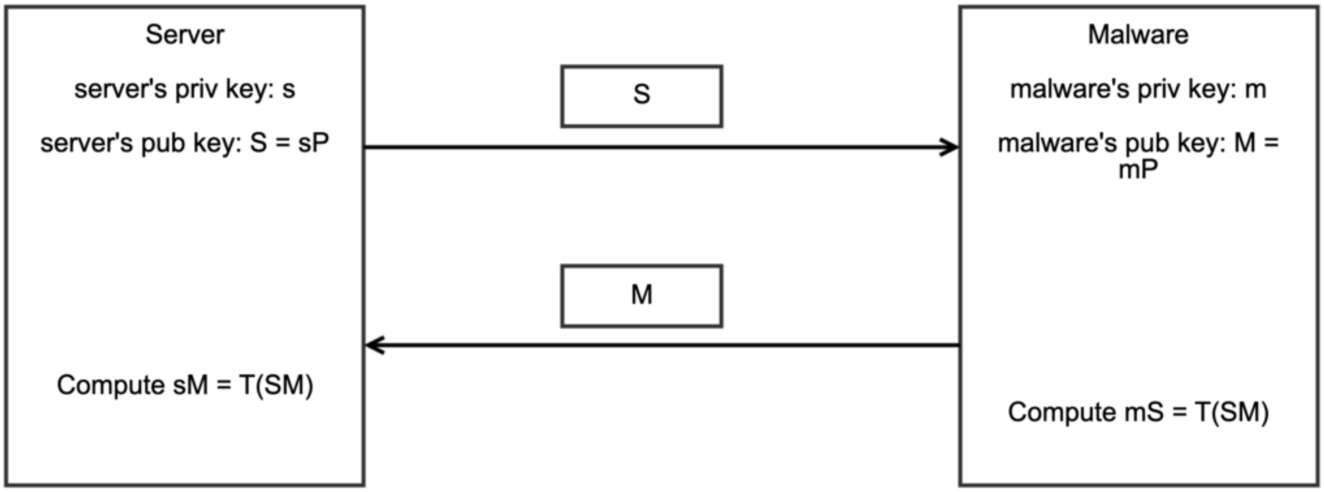
1. It generates a 128-bit AES key.
2. Encrypts it with the server's public key.
3. Constructs the message sent to the server.
4. Encrypts the message and hashes the message.C = SHA1(M) || AES128(M), where C is the resulting ciphertext and M is the plaintext message
5. This results in the following request packet.R = RSA(AESkey) || C

For the newest version the flow and the packets it generates are different as seen below:

1. It first generates its own ECDH public/private key pair.
2. Then it generate an AES key based on a secret agreement.
3. Constructs the message and hashes it.
4. Encrypts the resulting payload: *C = AES256(SHA256(M) || M)*
5. Request packet is then given by: *R = ECDHmal_pub_key* || *C* || *<random bytes>*

Elliptic Curve Diffie-Hellman (ECDH) Key Exchange

For the ECDH to work, the two communicating parties need to each have a key pair, a private and a public key. The public keys are points on an elliptic curve and are generated based on the private keys. The public keys are exchanged, i.e., known by both parties. For example, if s is a private key and P is a primitive element on the curve, then the public key S is calculated as sP=S, which is simply adding P to itself a times. The addition is a group operation. If both parties generate their public keys that way based on known domain parameters, they can calculate the same secret T(SM) (1).



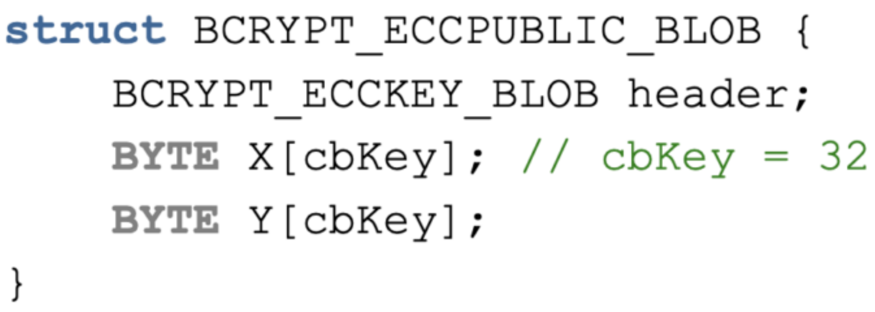
(Figure 1: An example of a DH key exchange algorithm)

The malware already has the ECDH public key of the server. Its own key pair is generated during the execution. Analogues to the example above, it can now generate a secret from the public key of the server and its own private key. Now it only needs to sends its public key to the server for the server to also be able to derive the same secret.

Implementation

Usage of ECDH

The Emotet's cryptographic components are now utilizing Microsoft's Cryptography API: Next Generation (CNG), most notably the BCrypt cryptographic primitive functions. Initially, the malware decrypts the two embedded public keys of the server (ECDH and ECDSA). It uses the same decryption method as with other strings. The keys are saved inside a BLOB structure which consists of a BCRYPT_ECCKEY_BLOB header immediately followed by the key data (Figure 2).



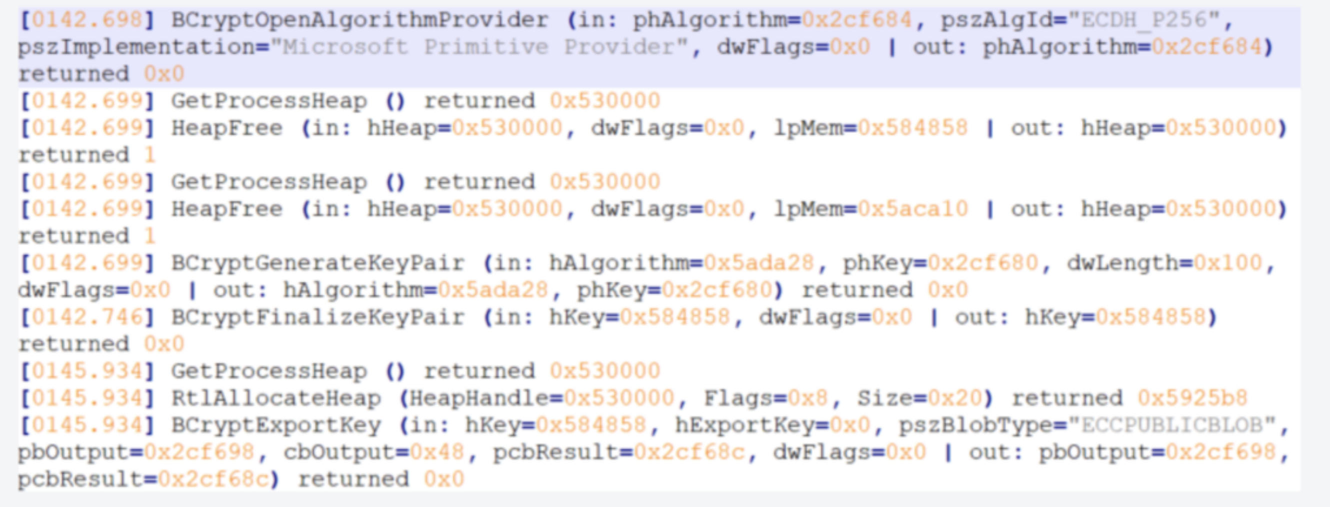
```
struct BCRYPT_ECCPUBLIC_BLOB {
    BCRYPT_ECCKEY_BLOB header;
    BYTE X[cbKey]; // cbKey = 32
    BYTE Y[cbKey];
}
```

(Figure 2: Structure that windows uses for the ECC public keys)

The ECDH public key of the server is passed to a function responsible for generating the symmetric key (256-bit AES key). On a higher-level it can be described by the following steps:

1. Generate a new ECDH key pair for the malware.
2. Generate a secret agreement based on the malware's private key and the server's public key.
3. Derive an AES key from the secret agreement using SHA256 as the key derivation function (KDF).

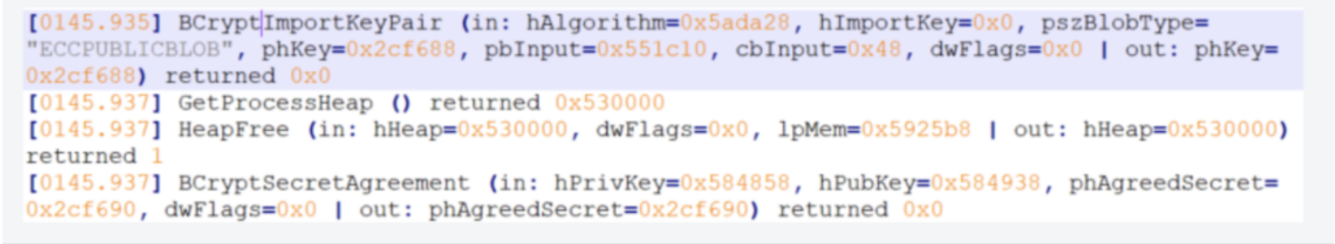
In more detail, this function's first step is to generate an ECDH key pair that is unique to the malware sample. It does so by calling BCryptOpenAlgorithmProvider to initialize a CNG provider with the AlgId ECDH_P256 which corresponds to the prime256v1 or P-256 elliptic curve. Next, it generates a new key pair using the combination of BCryptGenerateKeyPair and BCryptFinalizeKeyPair. The keys are then exported into a BLOB using BCryptExportKey for later use (Figure 3).



```
[0142.698] BCryptOpenAlgorithmProvider (in: phAlgorithm=0x2cf684, pszAlgId="ECDH_P256",
pszImplementation="Microsoft Primitive Provider", dwFlags=0x0 | out: phAlgorithm=0x2cf684)
returned 0x0
[0142.699] GetProcessHeap () returned 0x530000
[0142.699] HeapFree (in: hHeap=0x530000, dwFlags=0x0, lpMem=0x584858 | out: hHeap=0x530000)
returned 1
[0142.699] GetProcessHeap () returned 0x530000
[0142.699] HeapFree (in: hHeap=0x530000, dwFlags=0x0, lpMem=0x5aca10 | out: hHeap=0x530000)
returned 1
[0142.699] BCryptGenerateKeyPair (in: hAlgorithm=0x5ada28, phKey=0x2cf680, dwLength=0x100,
dwFlags=0x0 | out: hAlgorithm=0x5ada28, phKey=0x2cf680) returned 0x0
[0142.746] BCryptFinalizeKeyPair (in: hKey=0x584858, dwFlags=0x0 | out: hKey=0x584858)
returned 0x0
[0145.934] GetProcessHeap () returned 0x530000
[0145.934] RtlAllocateHeap (HeapHandle=0x530000, Flags=0x8, Size=0x20) returned 0x5925b8
[0145.934] BCryptExportKey (in: hKey=0x584858, hExportKey=0x0, pszBlobType="ECCPUBLICBLOB",
pbOutput=0x2cf698, cbOutput=0x48, pcbResult=0x2cf68c, dwFlags=0x0 | out: pbOutput=0x2cf698,
pcbResult=0x2cf68c) returned 0x0
```

(Figure 3: VMRay function log – series of function calls responsible for creating a new EC key pair)

Having finalized its key pair, it now imports the servers public key to be able to use it in the generation of a shared secret. It's using BCryptImportKeyPair that gets the public key as one of the arguments and returns a handle to it. This handle can then be passed to BCryptSecretAgreement together with a handle to it's own key which it got in the previous step from calling BCryptExportKey (Figure 4). At this stage the secret agreement is equal to the T(SM) value from Figure 1 and Emotet can start deriving a symmetric key.

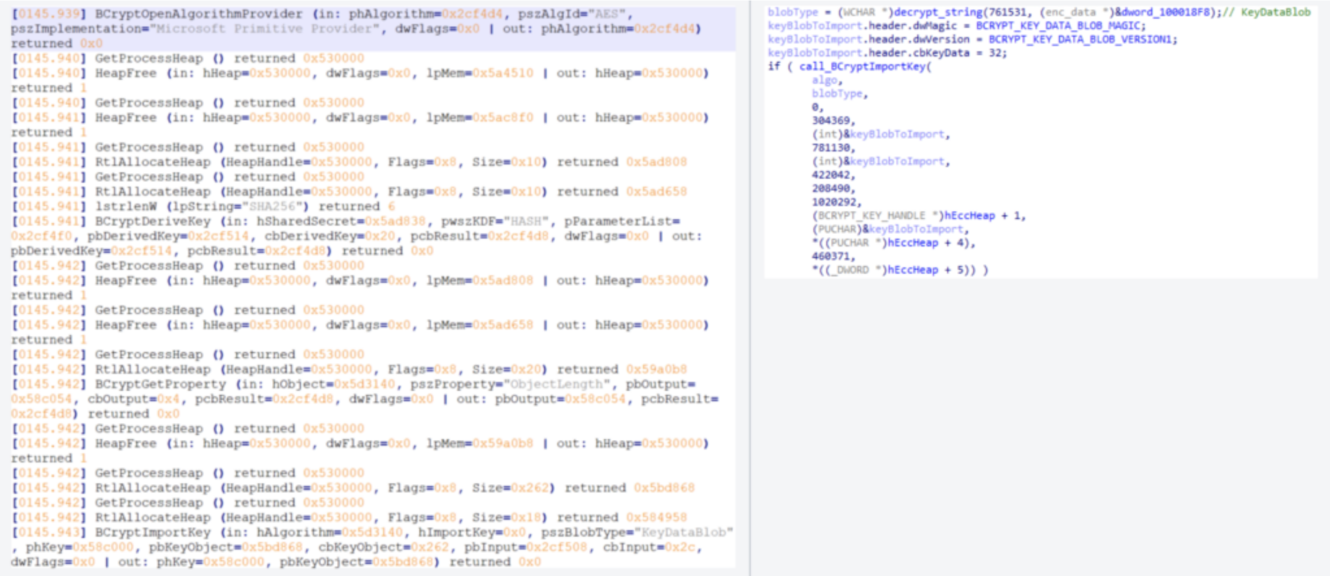


```
[0145.935] BCryptImportKeyPair (in: hAlgorithm=0x5ada28, hImportKey=0x0, pszBlobType=
"ECCPUBLICBLOB", phKey=0x2cf688, pbInput=0x551c10, cbInput=0x48, dwFlags=0x0 | out: phKey=
0x2cf688) returned 0x0
[0145.937] GetProcessHeap () returned 0x530000
[0145.937] HeapFree (in: hHeap=0x530000, dwFlags=0x0, lpMem=0x5925b8 | out: hHeap=0x530000)
returned 1
[0145.937] BCryptSecretAgreement (in: hPrivKey=0x584858, hPubKey=0x584938, phAgreedSecret=
0x2cf690, dwFlags=0x0 | out: phAgreedSecret=0x2cf690) returned 0x0
```

(Figure 4: VMRay function log – series of function calls responsible for creating the secret agreement)

The secret generated from the public key of the server and the private key of the malware sample is then used to generate an AES key. A new CNG provider is initialized with the AlgId = AES. The key is then derived using BCryptDeriveKey. This function takes the secret agreement as input and generates a key based on a key derivation function (KDF) and its parameters which are passed in the BCryptBufferDesc structure. For that Emotet uses HASH as the KDF and passes the SHA256 as the actual algorithm. This key is then imported using BCryptImportKey (for symmetric keys) so that it can also be later used when encrypting data. The KeyDataBlob passed as argument to BCryptImportKey describes the key. Based on the

BCRYPT_KEY_DATA_BLOB_HEADER the key data size is 32 bytes, i.e., 256 bits (Figure 5). To generate the same symmetric key, the server needs the public key of the malware which it prepends to the request sent to the server.



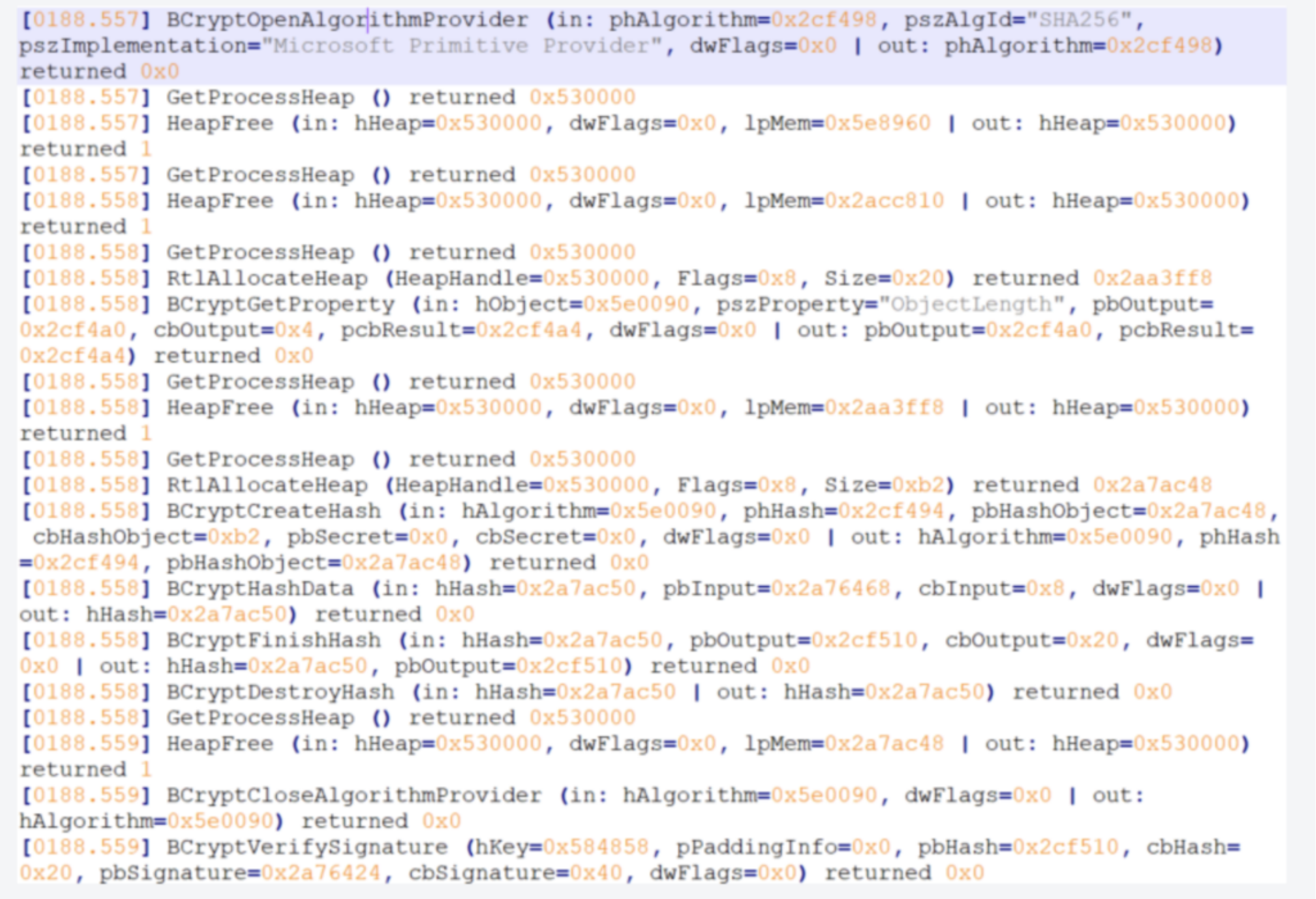
```
[0145.939] BCryptOpenAlgorithmProvider (in: phAlgorithm=0x2cf4d4, pszAlgId="AES",
pszImplementation="Microsoft Primitive Provider", dwFlags=0x0 | out: phAlgorithm=0x2cf4d4)
returned 0x0
[0145.940] GetProcessHeap () returned 0x530000
[0145.940] HeapFree (in: hHeap=0x530000, dwFlags=0x0, lpMem=0x5a4510 | out: hHeap=0x530000)
returned 1
[0145.940] GetProcessHeap () returned 0x530000
[0145.941] HeapFree (in: hHeap=0x530000, dwFlags=0x0, lpMem=0x5ac8f0 | out: hHeap=0x530000)
returned 1
[0145.941] GetProcessHeap () returned 0x530000
[0145.941] RtlAllocateHeap (HeapHandle=0x530000, Flags=0x8, Size=0x10) returned 0x5ad808
[0145.941] GetProcessHeap () returned 0x530000
[0145.941] RtlAllocateHeap (HeapHandle=0x530000, Flags=0x8, Size=0x10) returned 0x5ad658
[0145.941] lstrlenW (lpString="SHA256") returned 6
[0145.941] BCryptDeriveKey (in: hSharedSecret=0x5ad838, pwszKDF="HASH", pParameterList=
0x2cf4f0, pbDerivedKey=0x2cf514, cbDerivedKey=0x20, pcbResult=0x2cf4d8, dwFlags=0x0 | out:
pbDerivedKey=0x2cf514, pcbResult=0x2cf4d8) returned 0x0
[0145.942] GetProcessHeap () returned 0x530000
[0145.942] HeapFree (in: hHeap=0x530000, dwFlags=0x0, lpMem=0x5ad808 | out: hHeap=0x530000)
returned 1
[0145.942] GetProcessHeap () returned 0x530000
[0145.942] HeapFree (in: hHeap=0x530000, dwFlags=0x0, lpMem=0x5ad658 | out: hHeap=0x530000)
returned 1
[0145.942] GetProcessHeap () returned 0x530000
[0145.942] RtlAllocateHeap (HeapHandle=0x530000, Flags=0x8, Size=0x20) returned 0x59a0b8
[0145.942] BCryptGetProperty (in: hObject=0x5d3140, pszProperty="ObjectLength", pbOutput=
0x58c054, cbOutput=0x4, pcbResult=0x2cf4d8, dwFlags=0x0 | out: pbOutput=0x58c054, pcbResult=
0x2cf4d8) returned 0x0
[0145.942] GetProcessHeap () returned 0x530000
[0145.942] HeapFree (in: hHeap=0x530000, dwFlags=0x0, lpMem=0x59a0b8 | out: hHeap=0x530000)
returned 1
[0145.942] GetProcessHeap () returned 0x530000
[0145.942] RtlAllocateHeap (HeapHandle=0x530000, Flags=0x8, Size=0x262) returned 0x5bd868
[0145.942] GetProcessHeap () returned 0x530000
[0145.942] RtlAllocateHeap (HeapHandle=0x530000, Flags=0x8, Size=0x18) returned 0x584958
[0145.943] BCryptImportKey (in: hAlgorithm=0x5d3140, hImportKey=0x0, pszBlobType="KeyDataBlob"
, phKey=0x58c000, pbKeyObject=0x5bd868, cbKeyObject=0x262, pbInput=0x2cf508, cbInput=0x2c,
dwFlags=0x0 | out: phKey=0x58c000, pbKeyObject=0x5bd868) returned 0x0
```

```
blobType = (WCHAR *)decrypt_string(761531, (enc_data *)&dword_100018F8);// KeyDataBlob
keyBlobToImport.header.dwMagic = BCRYPT_KEY_DATA_BLOB_MAGIC;
keyBlobToImport.header.dwVersion = BCRYPT_KEY_DATA_BLOB_VERSION1;
keyBlobToImport.header.cbKeyData = 32;
if ( call_BCryptImportKey(
       algo,
       blobType,
       0,
       304369,
       (int)&keyBlobToImport,
       781130,
       (int)&keyBlobToImport,
       422042,
       208490,
       1020292,
       (BCRYPT_KEY_HANDLE *)hEccHeap + 1,
       (PUCHAR)&keyBlobToImport,
       *((PUCHAR *)hEccHeap + 4),
       460371,
       *((_DWORD *)hEccHeap + 5)) )
```

(Figure 5: VMRay function log – sequence of BCrypt calls responsible for key derivation (left) and pseudo-code of the data structure describing the key (right).)

Usage of the Elliptic Curve Digital Signature Algorithm (ECDSA)

The server's ECDSA public key is used to verify the response messages the malware receives. The server's DSA public key is imported just like ECDH public key was. When an encrypted response from the server arrives, it is first decrypted with BCryptDecrypt (no padding is used). It then calculates the SHA256 hash of the decrypted data and uses BCryptVerifySignature to verify the integrity and authenticity, i.e., that it matches with the embedded signed hash – signature (Figure 6).



```
[0188.557] BCryptOpenAlgorithmProvider (in: phAlgorithm=0x2cf498, pszAlgId="SHA256",
pszImplementation="Microsoft Primitive Provider", dwFlags=0x0 | out: phAlgorithm=0x2cf498)
returned 0x0
[0188.557] GetProcessHeap () returned 0x530000
[0188.557] HeapFree (in: hHeap=0x530000, dwFlags=0x0, lpMem=0x5e8960 | out: hHeap=0x530000)
returned 1
[0188.557] GetProcessHeap () returned 0x530000
[0188.558] HeapFree (in: hHeap=0x530000, dwFlags=0x0, lpMem=0x2acc810 | out: hHeap=0x530000)
returned 1
[0188.558] GetProcessHeap () returned 0x530000
[0188.558] RtlAllocateHeap (HeapHandle=0x530000, Flags=0x8, Size=0x20) returned 0x2aa3ff8
[0188.558] BCryptGetProperty (in: hObject=0x5e0090, pszProperty="ObjectLength", pbOutput=
0x2cf4a0, cbOutput=0x4, pcbResult=0x2cf4a4, dwFlags=0x0 | out: pbOutput=0x2cf4a0, pcbResult=
0x2cf4a4) returned 0x0
[0188.558] GetProcessHeap () returned 0x530000
[0188.558] HeapFree (in: hHeap=0x530000, dwFlags=0x0, lpMem=0x2aa3ff8 | out: hHeap=0x530000)
returned 1
[0188.558] GetProcessHeap () returned 0x530000
[0188.558] RtlAllocateHeap (HeapHandle=0x530000, Flags=0x8, Size=0xb2) returned 0x2a7ac48
[0188.558] BCryptCreateHash (in: hAlgorithm=0x5e0090, phHash=0x2cf494, pbHashObject=0x2a7ac48,
 cbHashObject=0xb2, pbSecret=0x0, cbSecret=0x0, dwFlags=0x0 | out: hAlgorithm=0x5e0090, phHash
=0x2cf494, pbHashObject=0x2a7ac48) returned 0x0
[0188.558] BCryptHashData (in: hHash=0x2a7ac50, pbInput=0x2a76468, cbInput=0x8, dwFlags=0x0 |
out: hHash=0x2a7ac50) returned 0x0
[0188.558] BCryptFinishHash (in: hHash=0x2a7ac50, pbOutput=0x2cf510, cbOutput=0x20, dwFlags=
0x0 | out: hHash=0x2a7ac50, pbOutput=0x2cf510) returned 0x0
[0188.558] BCryptDestroyHash (in: hHash=0x2a7ac50 | out: hHash=0x2a7ac50) returned 0x0
[0188.558] GetProcessHeap () returned 0x530000
[0188.559] HeapFree (in: hHeap=0x530000, dwFlags=0x0, lpMem=0x2a7ac48 | out: hHeap=0x530000)
returned 1
[0188.559] BCryptCloseAlgorithmProvider (in: hAlgorithm=0x5e0090, dwFlags=0x0 | out:
hAlgorithm=0x5e0090) returned 0x0
[0188.559] BCryptVerifySignature (hKey=0x584858, pPaddingInfo=0x0, pbHash=0x2cf510, cbHash=
0x20, pbSignature=0x2a76424, cbSignature=0x40, dwFlags=0x0) returned 0x0
```

(Figure 6: VMRay function log – BCrypt functions used when verifying the response)

Conclusion

We have looked at one of the updated components of Emotet which involves the usage of cryptography. The most obvious element is that the malware developers switched from the RSA algorithm to using elliptic curves. Emotet has been encrypting its communication for a long time, but the recent change might be due to a lot of factors like, e.g., smaller key sizes and better security. The C2's response is now checked for its integrity and authenticity by using ECDSA with a separate key. While using ECDH the symmetric key is never transmitted over the wire and instead the server generates the key from the public key of the malware. We have also observed the switch from CryptoAPI to CNG, which might be due to the fact that the CryptoAPI has been officially deprecated or that it simply didn't support elliptic curve cryptography.

IOCs

Initial Sample 7443d5335a207cca176825bd774a412e72882c815206c7f59ace1feb111bb4e9

Server's ECC keys

ECDH:
86M1tQ4uK/Q1Vs0KTCk+fPEQ3cuwTyCz+gIgzky2DB5EIr60DubJW5q9Tr2dj8/gEFs0TIIEJgLTuqzx+58sdg==

ECDSA:
QF90tsTY3Aw9HwZ6N9y5+be9XoovpqHyD6F5DRTl9THosAoePIs/e5AdJiYxhmV8Gq3Zw1ysSPBghxjZdDxY+Q==

Emotet C&Cs

hXXps://131[.]100.24.231:80
hXXps://209[.]59.138.75:7080
hXXps://103[.]8.26.102:8080
hXXps://178[.]79.147.66:8080
hXXps://51[.]38.71.0:443
hXXps://79[.]172.212.216:8080
hXXps://162[.]214.50.39:7080
hXXps://203[.]114.109.124:443
hXXps://45[.]142.114.231:8080
hXXps://212[.]237.5.209:443
hXXps://104[.]251.214.46:8080
hXXps://212[.]237.56.116:7080
hXXps://107[.]182.225.142:8080
hXXps://104[.]168.155.129:8080
hXXps://138[.]185.72.26:8080
hXXps://45[.]118.135.203:7080
hXXps://216[.]158.226.206:443
hXXps://103[.]75.201.2:443
hXXps://158[.]69.222.101:443
hXXps://178[.]63.25.185:443
hXXps://45[.]118.115.99:8080
hXXps://46[.]55.222.11:443
hXXps://192[.]254.71.210:443
hXXps://217[.]182.143.207:443
hXXps://110[.]232.117.186:8080
hXXps://81[.]0.236.90:443
hXXps://176[.]104.106.96:8080
hXXps://103[.]8.26.103:8080
hXXps://50[.]116.54.215:443
hXXps://195[.]154.133.20:443
hXXps://51[.]68.175.8:8080
hXXps://58[.]227.42.236:80
hXXps://173[.]212.193.249:8080
hXXps://212[.]237.17.99:8080
hXXps://41[.]76.108.46:8080
hXXps://45[.]176.232.124:443
hXXps://207[.]38.84.195:8080

References

https://www.cert.ssi.gouv.fr/uploads/CERTFR-2021-CTI-003.pdf

https://www.europol.europa.eu/media-press/newsroom/news/world%e2%80%99s-most-dangerous-malware-emotet-disrupted-through-global-action

https://cyber.wtf/2021/11/15/guess-whos-back/

https://blog.malwarebytes.com/threat-intelligence/2021/11/trickbot-helps-emotet-come-back-from-the-dead/

https://link.springer.com/book/10.1007/978-3-642-04101-3

https://securityintelligence.com/new-banking-trojan-icedid-discovered-by-ibm-x-force-research/

https://nakedsecurity.sophos.com/2017/08/10/watch-out-for-emotet-the-trojan-thats-nearly-a-worm/

https://unit42.paloaltonetworks.com/unit42-malware-team-malspam-pushing-emotet-trickbot/

https://www.virusbulletin.com/virusbulletin/2019/10/vb2019-paper-exploring-emotet-elaborate-everyday-enigma/

https://docs.microsoft.com/en-us/windows/win32/seccng/about-cng

https://docs.microsoft.com/en-us/windows/win32/seccng/cryptographic-primitives

https://docs.microsoft.com/de-de/windows/win32/api/bcrypt/ns-bcrypt-bcrypt_ecckey_blob