# Emotet C2 Configuration Extraction and Analysis

By Oleg Boyarchuk, Jason Zhang

Published: 2022-03-29 · Archived: 2026-04-10 03:00:41 UTC

The Emotet actors have re-gained their power to launch attacks since the Emotet botnet was taken down in 2021. VMware's NSX Sandbox detected a series of attack waves of such attacks in January of this year. More details about the attacks can be found in our recent reports (Emotet Is Not Dead (Yet) part 1 and part 2.)

The Emotet botnet is known to use many command-and-control (C2) servers to keep communication open between the infected machines and the botnet's herders. Providing visibility into the C2 configuration of Emotet payloads can help in many ways, from detection to threat hunting. In this report, we first discuss the steps on how to extract the C2 configuration from Emotet payloads via a combination of dynamic and static analysis, including decrypting and dumping the embedded Emotet payload from the initial DLL payload dropped by documents, and the actual extraction process of C2 configuration contained in the decrypted Emotet payload. We then propose a process to automate the configuration extraction steps by leveraging the NSX Sandbox, which allows us to extract the C2 configuration from Emotet payloads at scale. In the final part, we provide an analysis of the most distinctive aspects of the C2 configurations extracted from recent Emotet campaigns.

## Executive Summary

This is a technical report containing our analysis on some Emotet attacks taking place in Q1 2022. In this report we detail:

- How to decrypt and dump the internal DLL from the initial Emotet DLL payload.
- How to extract C2 configuration contained in the internal DLL.
- Analysis of the C2 configuration data extracted from over 2000 DLL dropped payloads.
- Characterization of the network infrastructure of the botnets.

Emotet is a sophisticated botnet that comprises a few subgroups or sub-botnets, called "epochs." Each epoch has its own C2 infrastructure and distribution methods. As discussed in a report published in 2019, different epochs may be used to target different countries with different payloads. At the time of writing, there are five epochs (labeled as Epoch 1, Epoch 2, and so on). Epoch 1, Epoch 2, and Epoch 3 were mostly seen before the botnet was taken down in early 2020. Epoch 4 and Epoch 5 were introduced after Emotet resurfaced. The epoch number of a sample is typically identified by the public encryption key(s) contained in the C2 configuration of the sample. Though Emotet samples of different epochs keep their configuration data in different formats, they all share one common approach: their configuration is stored in an encrypted DLL (hereinafter referred to as "the internal DLL") that is embedded into the executable payload (called "the payload").

Figure 1 illustrates the C2 configuration extraction process for a given Emotet attack. The attacks are known to start with spam emails containing weaponized documents (e.g., Word or Excel files) leveraging embedded malicious VBA or Excel 4.0 (XL4) macros. The execution of macros typically leads to the delivery of a DLL file (the payload) that, when executed (often by calling rundll32.exe or regsvr32.exe), decrypts an internal DLL into memory and executes it (as highlighted in Figure 1).

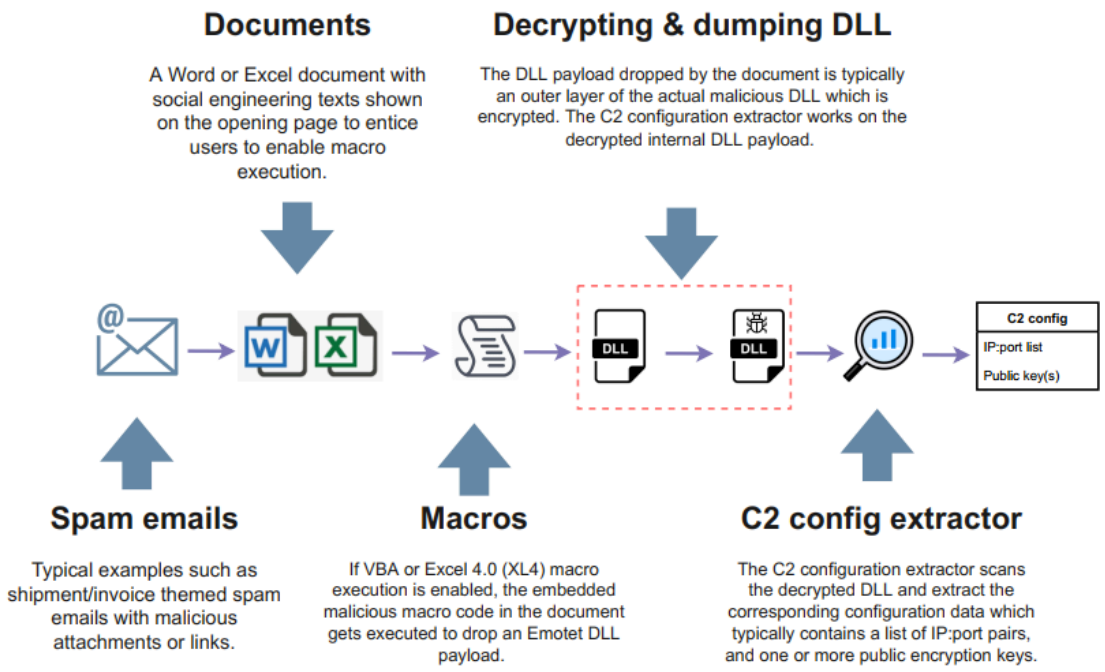


Figure 1: Emotet C2 configuration extraction pipeline.

Therefore, the process of extracting the Emotet C2 configuration has two main steps:

1. Decrypting and dumping the internal DLL from the initial DLL payload.
2. Scanning the decrypted internal DLL to extract key C2 configuration data, such as the C2 servers' IP:port pairs and the public encryption key(s) (see Figure 1).

In the remaining of this section, we will focus our discussion on these two steps via manual analysis.

### Decrypting and Dumping the Internal DLL

To demonstrate this step, we analyze one of the Emotet samples (63996a39755e84ee8b5d3f47296991362a17afaaccf2ac43207a424a366f4cc9).

The DllMain function of this sample (and many others) has the following algorithm:

1. Allocate ~ 100 MB of memory with malloc and fill it with random data. This stops the analysis of weak emulators not willing to allocate large amounts of memory.
2. Find the base address of kernel32.dll by PEB, PEB_LDR_DATA, etc. While normally this method is used to make the reverse engineering process more difficult, statically imported functions are still used later in the code; we speculate this to be a trick to also break emulation, as references to internal OS structures are rarely fully handled by emulators.
3. Find VirtualAlloc and VirtualAllocExNuma with the help of an ad hoc version of GetProcAddress.
4. Allocate memory with either VirtualAllocExNuma or VirtualAlloc, depending on which one is available. VirtualAlloc is supported starting with Windows XP whereas VirtualAllocExNuma is supported starting with Windows Vista. This looks like another trick to stop weak emulators that do not support the complete set of Windows APIs.
5. Copy the internal DLL into the allocated memory and then decrypt it.
6. Map the sections of the internal DLL in memory, fix relocations and imports. This is achieved with the help of the statically imported functions VirtualAlloc, LoadLibrary, and GetProcAddress.

To be able to decrypt and dump the internal DLL, it is first necessary to load the original DLL payload into a debugger, set breakpoints on the invocation of VirtualAllocExNuma and VirtualAlloc, and then start execution. When the execution

reaches the breakpoint, one needs to trace the code until it returns a pointer to the allocated memory (in the image below the address of the newly allocated memory is 0x00E7000).

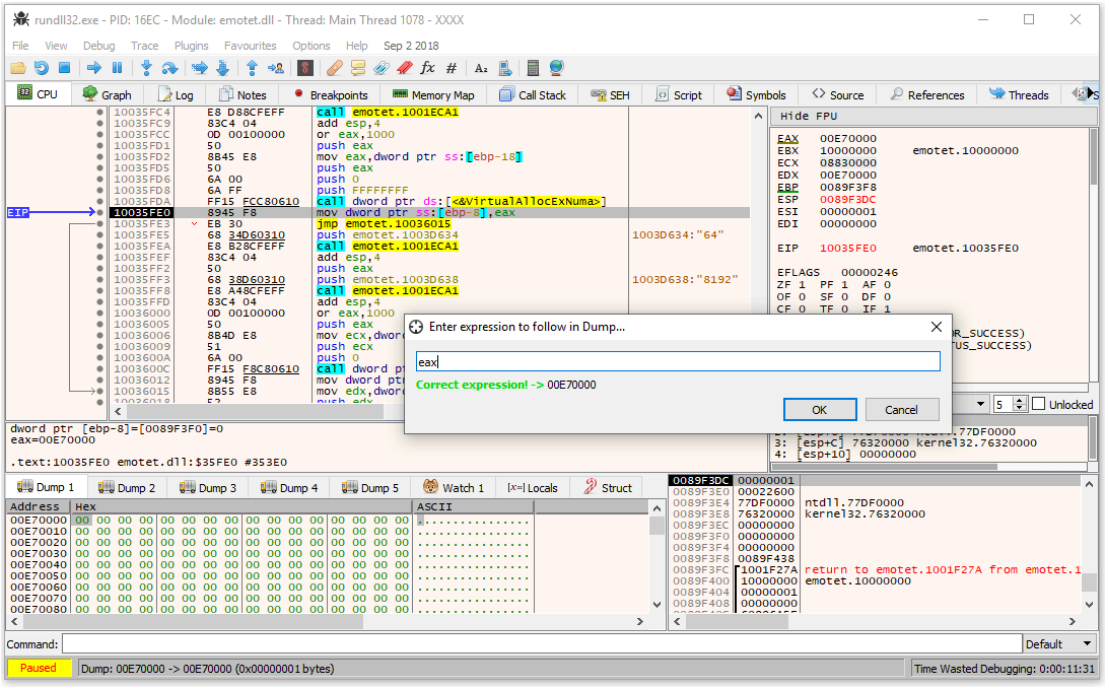


Figure 2: Memory, allocated with VirtualAllocExNuma.

The next step is to trace the code of DllMain until it copies the encrypted DLL into the allocated memory. In the figure below, one may see that the data of the Dump tab has changed from all zeros to random bytes.

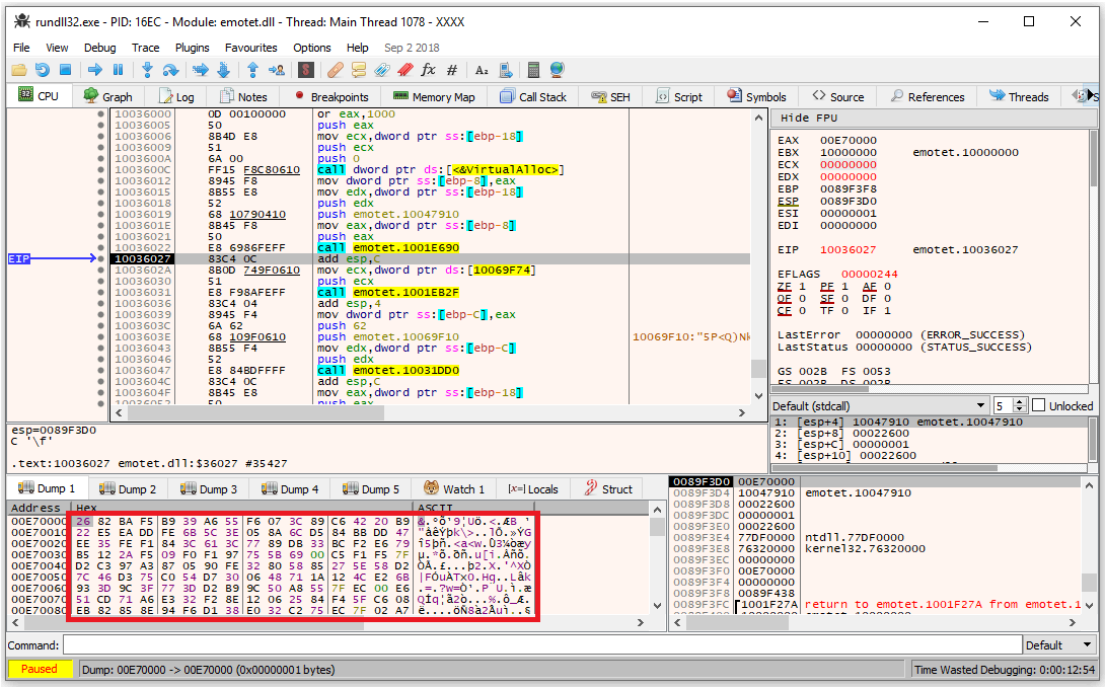


Figure 3: Embedded DLL copied into the allocated memory.

One can trace the code of DllMain a bit further until observing that the data of the Dump tab updates to a PE file with the "MZ" signature at the very beginning and the text "This program cannot be run in DOS mode", as shown in the figure below.

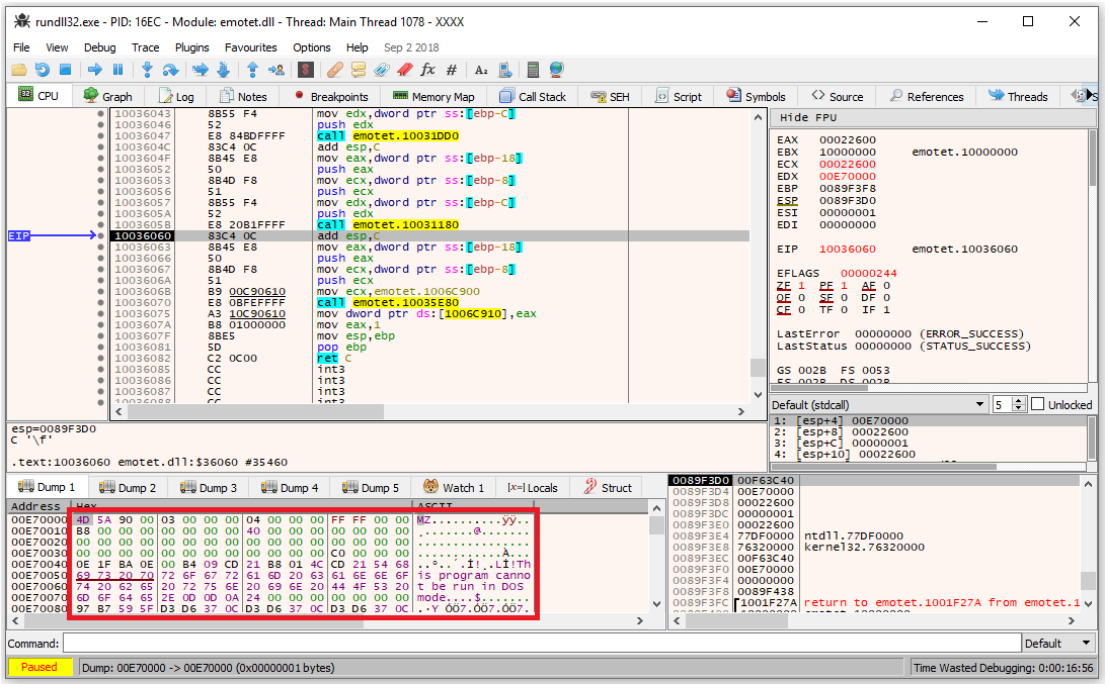


Figure 4: Decrypted embedded DLL in allocated memory.

At this point, it is possible to dump the decrypted internal DLL.

### C2 Configuration Extraction

The next step is to locate the configuration data. The DLL is supposed to be executed with the help of rundll32; use the following command line when debugging this artifact:

*"C:\Windows\system32\rundll32.exe" "Path\to\dumped.dll",DllRegisterServer*

The internal DLL does not import any function. Instead, it retrieves pointers to the Windows API functions dynamically. In addition, some of the core functionality is obfuscated, which makes static analysis challenging. Under the hood the code obfuscation includes mathematical operations performed multiple times as shown in the figure below.

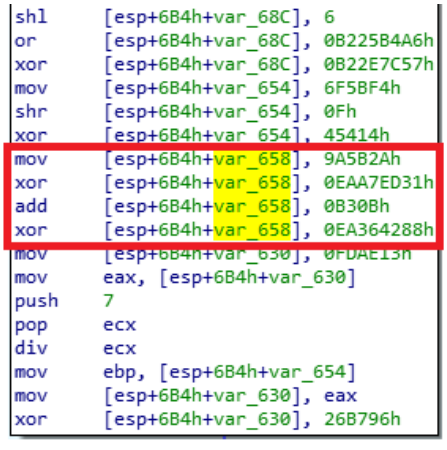


Figure 5: Mathematical operations on a number in the obfuscated code.

The result of such calculations is passed to a function and then never used (see Figure 6).

```
push    [esp+6B4h+var_630]
xor     ecx, ecx
push    [esp+6B8h+var_658]
push    ebx
push    ebp
push    [esp+6C4h+var_654]
push    ebx
push    [esp+6CCh+var_68C]
call    sub_6A3D0DCB
xor     ebx, ebx
add     esp, 1Ch
inc     ebx
```

Figure 6: Result of the mathematical operations passed as 6th parameter to a function.

As Figure 7 shows, the obfuscation also includes multiple conditional jumps that break the code flow of the decompiled code.

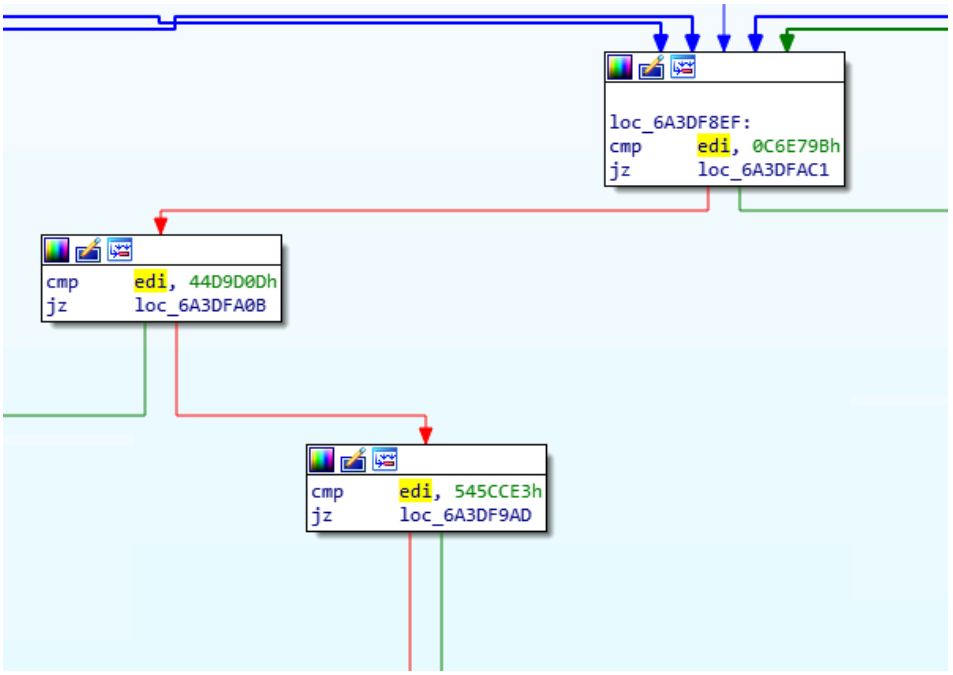


Figure 7: Conditional jumps in the obfuscated code.

The abundance of jumps is translated into nested while-loops in the decompiled obfuscated code (see Figure 8).

```
  v0 = 182879479;
  v1 = 10000;
  v2 = 10000;
  v3 = 125;
  result = 54767;
  v5 = 98;
  while ( 1 )
  {
    while ( 1 )
    {
      while ( 1 )
      {
        while ( 1 )
        {
          while ( 1 )
          {
            while ( v0 <= 141589151 )
            {
              if ( v0 == 141589151 )
              {
                result = sub_703E2610();
                v0 = 259674042;
              }
              else if ( v0 > 77303981 )
              {
                if ( v0 > 105252088 )
                {
                  switch ( v0 )
                  {
                    case 107854090:
                      result = sub_703E8104(v3, v5);
                      if ( !result )
                        return result;
                      v0 = 47140117;
                      break;
                    case 109770162:
                      result = sub_703FD4D8(v3, v5);
                      v7 = result != 0 ? 0x93815D9 : 0;
LABEL_17:
                      v0 = v7 + 77303981;
                      break;
                    case 129798139:
                      result = sub_703ED076(v3, v5);
                      v28 = result;
                      v0 = 195186940;
                      break;
                    case 135654200:
                      result = sub_703FC394(v3, v5);
                      if ( !result )
                        return result;
```

Figure 8: Nested while loops of the obfuscated code.

Each API function has a wrapper that is called by the core functionality. For example, Figure 9 shows how DllMain calls the wrapper around the ExitProcess API.

```
signed int __stdcall DllMain(int a1, int a2, int a3)
{
  void *v3; // ecx

  if ( a2 == 1 )
  {
    dword_70404218 = a1;
    if ( sub_703FF47C() )
      l_ExitProcess(v3);
  }
  return 1;
}
```

Figure 9: DllMain of the embedded DLL with highlighted wrapper over ExitProcess.

The figure below shows the implementation of the ExitProcess wrapper. It calls FindProcAddress which is also called by every other API wrapper to retrieve the API function address by hash.

```
int __thiscall l_ExitProcess(void *this)
{
  int (__stdcall *v1)(_DWORD); // eax

  v1 = (int (__stdcall *)(_DWORD))FindProcAddress(-1402801697, (int)this, (int)this, 185, -1660415793);
  return v1(0);
}
```

Figure 10: Wrapper over ExitProcess API in the embedded DLL.

By setting a breakpoint on FindProcAddress, one can extract all API wrappers and name them. We are particularly interested in the API functions that work with memory. They will help us find the key function responsible for memory allocation. This function is shown in the figure below.

```
int __fastcall AllocateMemory(int a1, int a2)
{
  int v2; // esi
  int v3; // eax

  v2 = a2;
  v3 = l_GetProcessHeap((void *)0x26);
  return l_RtlAllocateHeap(982485, 8, 199840, v3, 156988, v2);
}
```

Figure 11: Memory allocation function of the embedded DLL.

AllocateMemory is called by many functions, but we are particularly interested in its use within a function that resembles a decoding cycle. Using manual analysis, we identified the following function:

```
int __usercall sub_7511EEB9@<eax>(int a1@<edx>, int a2@<ecx>, int a3, int a4, int *a5)
{
  _DWORD *v5; // ecx
  char *v6; // esi
  int v7; // edx
  unsigned int v8; // edi
  int v9; // ebp
  char *v10; // ecx
  unsigned int v11; // edi
  unsigned int v12; // edx
  int v13; // ecx
  int v15; // [esp+18h] [ebp-8h]
  int v16; // [esp+1Ch] [ebp-4h]

  nullsub_1(a2, a1, a3, a4, a5);
  v6 = (char *)(v5 + 2);
  v7 = *v5 ^ v5[1];
  v15 = *v5;
  v16 = v7;
  v8 = v7;
  if ( (v7 & 3) != 0 )
    v8 = (v7 & 0xFFFFFFFC) + 4;
  v9 = AllocateMemory();
  if ( v9 )
  {
    v10 = &v6[4 * (v8 >> 2)];
    v11 = 0;
    v12 = (unsigned int)(v10 - v6 + 3) >> 2;
    if ( v6 > v10 )
      v12 = 0;
    if ( v12 )
    {
      v13 = v9 - (_DWORD)v6;
      do
      {
        ++v11;
        *(_DWORD *)&v6[v13] = v15 ^ *(_DWORD *)v6;
        v6 += 4;
      }
      while ( v11 < v12 );
    }
    if ( a5 )
      *a5 = v16;
  }
  return v9;
}
```

Figure 12: Config decryption function of the embedded DLL.

By setting a breakpoint on this function, we can identify all the encrypted configs, which are passed in ECX (see Figure 13):

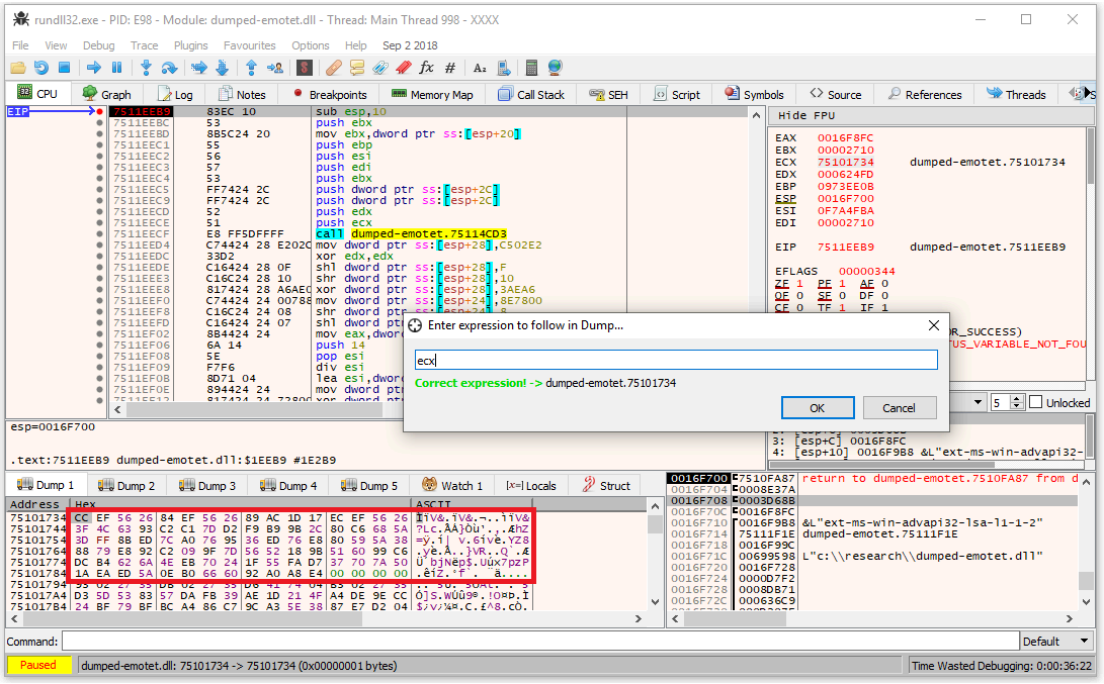


Figure 13: Pointer to the encrypted config passed to the decryption function in ECX.

Once the execution of this function ends, it returns a pointer to the decrypted config. In this case, we have the public key that is used in C2 communication, as highlighted in Figure 14.

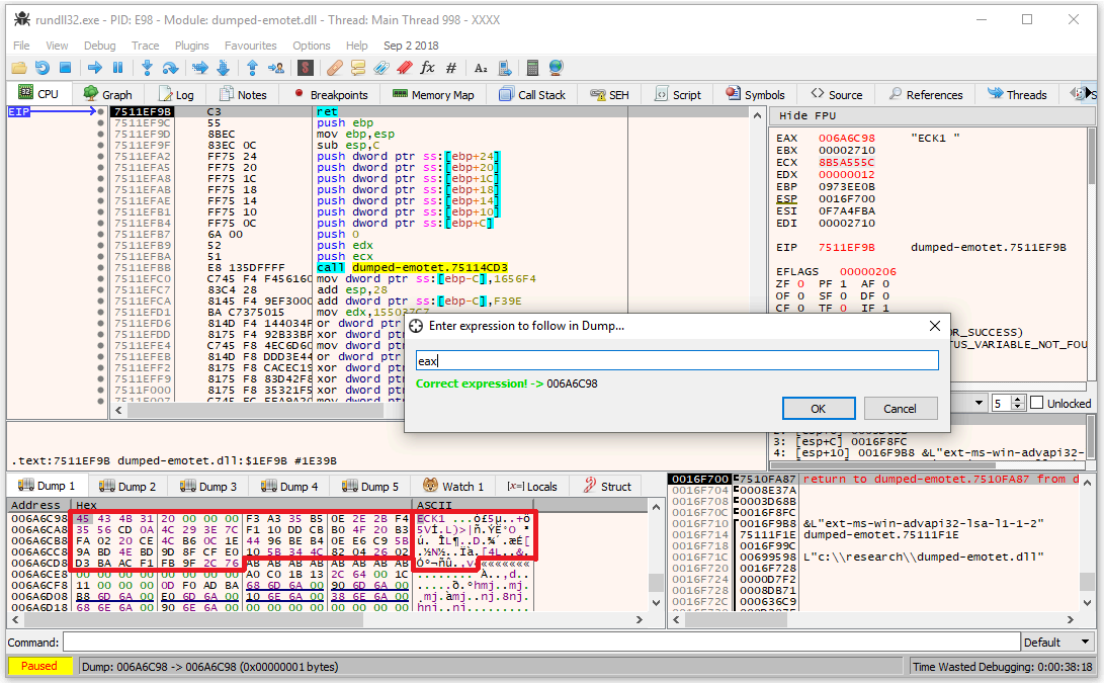


Figure 14: Decrypted C&C public key.

The encrypted data is stored in the following format (this format was found in samples belonging to both Epoch 4 and Epoch 5):

| Field | Offset | Size | Description |
|---|---|---|---|
| Key | 0 | 4 | Decryption key, little endian |
| Length | 4 | 4 | Length of data, little endian |

| Data | 8 | Variable | Encrypted data, split into DWORDs, little endian |
|---|---|---|---|

The length of the encrypted data can be retrieved by XORing the first DWORD of the encrypted data blob with the second one. To decrypt the data with the retrieved data length, one needs to split the data into DWORDs and then XOR them with the same first DWORD.

This way we decrypted the network keys, which are stored in the .text section of the extracted DLL. The keys extracted from the sample (63996a39755e84ee8b5d3f47296991362a17afaaccf2ac43207a424a366f4cc9) belong to Epoch 4:

- ECK1 (base64 encoded):
  *RUNLMSAAAADzozW1Di4r9DVWzQpMKT588RDdy7BPILP6AiDOTLYMHkSWvrQO5slbmr1OvZ2Pz+AQWzRMggQmAtO6rPH7ı*
- ECS1 (base64 encoded):
  *RUNTMSAAAABAX3S2xNjcDD0fBno33Ln5t71eii+mofIPoXkNFOX1MeiwCh48iz97kB0mJjGGZXwardnDXKxI8GCHGNl0PFj5*

The configuration containing C2 IP addresses and ports is normally stored at the very beginning of the .data section of the extracted DLL (see Figure 15). This configuration is encrypted with the same encryption method described previously.

```
.data:75124000
.data:75124000 ; Segment type: Pure data
.data:75124000 ; Segment permissions: Read/Write
.data:75124000 _data           segment para public 'DATA' use32
.data:75124000                 assume cs:_data
.data:75124000                 ;org 75124000h
.data:75124000 unk_75124000    db  39h ; 9             ; DATA XREF: sub_751134DB+3C1↑o
.data:75124001                 db  28h ; (
.data:75124002                 db 0DDh
.data:75124003                 db  1Ch
.data:75124004                 db  11h
.data:75124005                 db  29h ; )
.data:75124006                 db 0DDh
.data:75124007                 db  1Ch
.data:75124008                 db 0BAh
.data:75124009                 db  4Ch ; L
.data:7512400A                 db 0C5h
.data:7512400B                 db 0FBh
.data:7512400C                 db  39h ; 9
.data:7512400D                 db  78h ; x
.data:7512400E                 db 0DDh
.data:7512400F                 db  1Dh
.data:75124010                 db 0E8h
.data:75124011                 db  13h
.data:75124012                 db  57h ; W
.data:75124013                 db  57h ; W
.data:75124014                 db  22h ; "
.data:75124015                 db  80h
.data:75124016                 db 0DDh
.data:75124017                 db  1Dh
.data:75124018                 db  5Eh ; ^
.data:75124019                 db  20h
.data:7512401A                 db 0C7h
.data:7512401B                 db  7Bh ; {
.data:7512401C                 db  26h ; &
.data:7512401D                 db 0B8h
.data:7512401E                 db 0DDh
.data:7512401F                 db  1Dh
.data:75124020                 db  0Ah
.data:75124021                 db  0Eh
.data:75124022                 db  9Ah
.data:75124023                 db  1Ch
.data:75124024                 db  38h ; 8
.data:75124025                 db  93h
.data:75124026                 db 0DDh
.data:75124027                 db  1Dh
.data:75124028                 db 0EDh
.data:75124029                 db 0C5h
```

Figure 15: Encrypted list of IP:port pairs is stored at the beginning of .data section of the embedded DLL.

The decrypted configuration consists of an array of 8-byte elements, each with the following format:

| Field | Offset | Size | Description |
|---|---|---|---|

| | | | |
|---|---|---|---|
| IP | 0 | 1 | 1st part of the IP address |
| | 1 | 1 | 2nd part of the IP address |
| | 2 | 1 | 3rd part of the IP address |
| | 3 | 1 | 4th part of the IP address |
| Port | 4 | 2 | Corresponding port, little endian |
| Valid | 6 | 2 | Always 1, presumably “valid” flag, little endian |

The screenshot in Figure 16 shows the decrypted IP:port pairs:

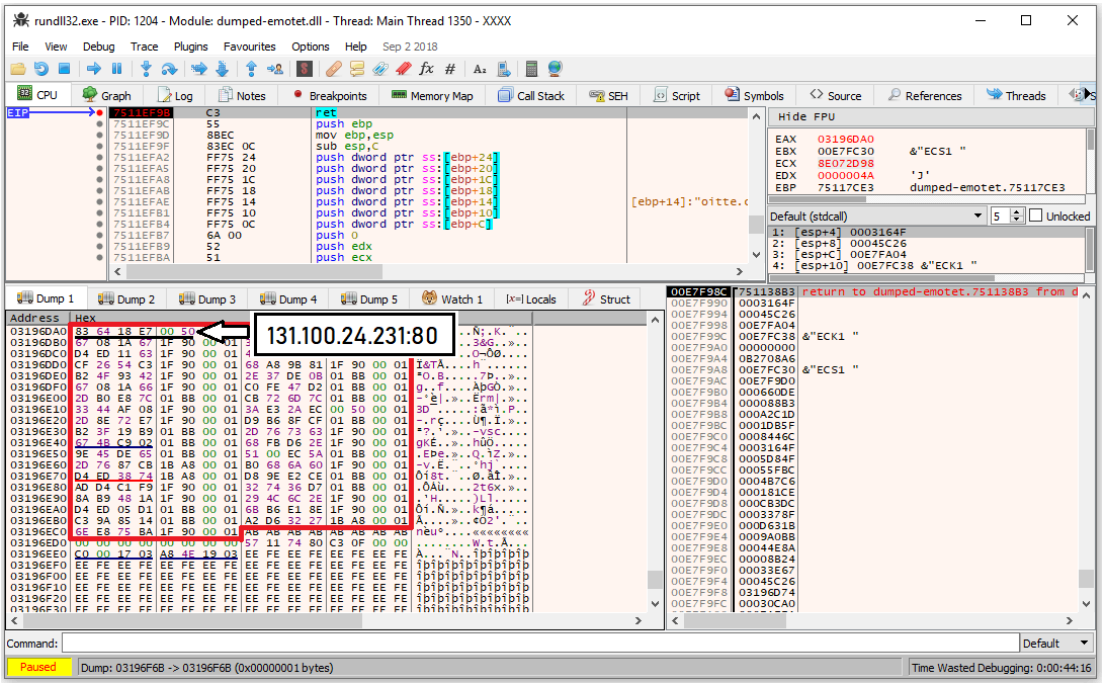


Figure 16: Decrypted list of IP:port pairs in binary format.

There are 37 IP:port pairs extracted from the sample, part of the list is shown below:

*131.100.24.231:80; 209.59.138.75:7080; 103.8.26.103:8080; 51.38.71.0:443; 212.237.17.99:8080;*

The steps of the extraction pipeline discussed above are based on manual analysis. As our analysis shows, even though it is possible to extract the decrypted payload and configuration data statically, this process is not efficient and does not scale. Therefore, we decided to fully automate the process for both steps. This is achieved by leveraging the NSX Sandbox analysis, which extracts and dumps the internal DLL artifact from the original Emotet DLL payload during execution (one can use other controlled environments as well.) The dumped DLL is then fed into the C2 configuration extractor for scanning. The extractor supports different configuration formats seen in various epochs.

With the automated extraction pipeline discussed above, we were able to extract C2 configurations from recent Emotet campaigns. In the following, we introduce the campaigns, and present our analysis of indicators of compromise (IoCs) contained in the C2 configurations.

## Recent Emotet Campaigns

Figure 17 shows a subset of VMware NSX telemetry data regarding Emotet attacks between January 1 and March 1, 2022. Following our earlier reports (Emotet Is Not Dead (Yet) part 1 and part 2) on the initial waves seen in the first four weeks of this year, more attacks were detected in the next four weeks.

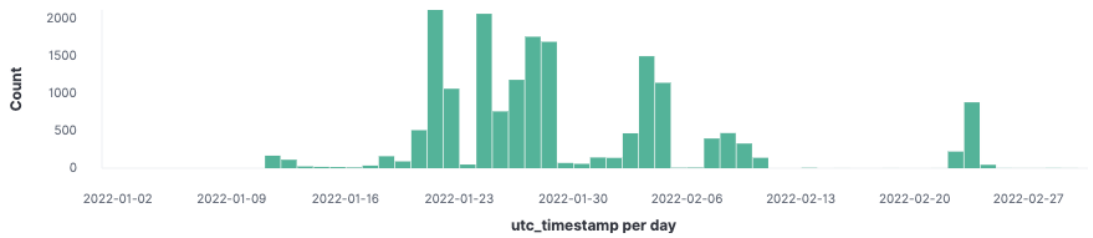


Figure 17: Excel document detection timeline of Emotet attacks.

Some key statistics of the telemetry data are shown below:

- **Total documents**: 17810.
- **Unique documents**: 9764. This number is smaller than the total document because a file could be submitted by various customers, resulting in multiple detections.
- **Unique documents that successfully dropped DLL payloads from remote hosts and executed DLL payloads within NSX Sandbox controlled environment**: 8888. If the remote hosts were taken down or not responding when the malicious documents were analyzed, the documents would fail to deliver the DLL payloads.
- **Unique DLL payloads**: 3132. This implies that every unique DLL payload was dropped by 2.8 different documents on average. Figure 18 shows the distribution of the top 20 DLL payloads. The most common DLL payload (with file sha256 starting with *22004*) appeared in 174 (or 1.96%) of all 8888 documents.

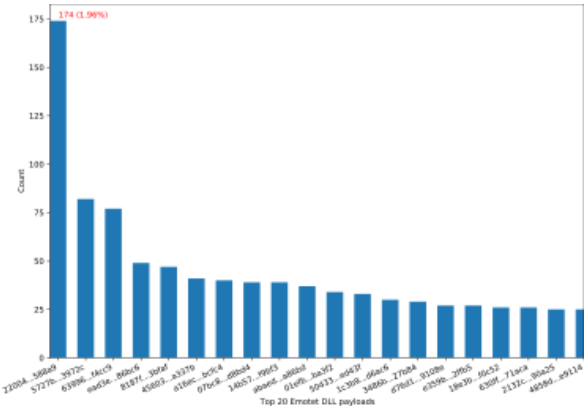


Figure 18: The distribution of top 20 Emotet DLL payloads.

There are various reasons that affect the availability of some DLLs for analysis. For instance, the NSX Sandbox detonation failed to extract the internal DLLs in a limited number of cases. So, we decided to skip these documents and the corresponding DLL payloads. The evaluation dataset used for the analysis in the next section contains 5941 unique documents which dropped the 2181 DLL payloads (representing 69.6% of the 3132 DLL payloads.)

## C2 Configuration Analysis

Using the C2 configuration extraction tool described above, we successfully extracted the C2 configuration data from all 2181 Emotet DLL payloads. The C2 configuration data extracted from each DLL payload sample comprises a pair of encryption keys and a list of IP:port pairs. In this section, we analyze the configuration data and discuss some interesting findings.

### Encryption Keys and Epoch Distribution

Prior to the takedown of 2020, Emotet had three sub-botnets: Epoch 1, Epoch 2 and Epoch 3. All of them leveraged a single hard-coded RSA public key to encrypt an AES encryption key generated on-the-fly to encrypt network traffic between an infected machine and the associated C2 servers. Differently, the samples from recent attacks use two keys in the communication protocols, labelled as ECK1 and ECS1, respectively. According to an early report, they are two Elliptic Curve Cryptography (ECC) public keys used for asymmetric encryption. ECK1 is a hard-coded Elliptic-curve Diffie-Hellman (ECDH) public key for encryption, and ECS1 a hard-coded Elliptic-curve Digital Signature Algorithm (ECDSA)

public key for data validation. There are two distinct pairs of such public keys extracted from our dataset, which correspond to Epoch 4 and Epoch 5 botnets, respectively:

- Epoch4
  - ECK1: *RUNLMSAAAADzozW1Di4r9DVWzQpMKT588RDdy7BPILP6AiDOTLYMHkSWvrQO5slbmr1OvZ2Pz+AQWzRMggQmAtO*
  - ECS1: *RUNTMSAAAABAX3S2xNjcDD0fBno33Ln5t71eii+mofIPoXkNFOX1MeiwCh48iz97kB0mJjGGZXwardnDXKxI8GCHGNl0P*
- Epoch5
  - ECK1: *RUNLMSAAAADYNZPXY4tQxd/N4Wn5sTYAm5tUOxY2ol1ELrI4MNhHNi640vSLasjYTHpFRBoG+o84vtr7AJachCzOHjaAJ*
  - ECS1: *RUNTMSAAAAD0LxqDNhonUYwk8sqo7IWuUllRdUiUBnACc6romsQoe1YJD7wIe4AheqYofpZFucPDXCZ0z9i+ooUffqeoLZ*

Figure 19 shows the breakdown in terms of IP addresses, DLL payloads, and the corresponding documents for Epoch 4 and Epoch 5. There are 134 unique IP addresses extracted from all 2181 DLL payloads. 57% of them were assigned to the Epoch 4 botnet, while 41.5% of the IP addresses belonged to the Epoch 5 botnet. There is only one IP (217.182.143[.]207, with port 443) that appeared in both botnets (see Figure 19 (A)). As a comparison, we also checked the distribution of IP:port pairs, which is identical to the distribution of IP addresses. For this reason, we only show the results from IP addresses. This largely confirms the findings in a BeepingComputer's report that each epoch has different C2 servers. The distinct C2 infrastructure by each epoch not only greatly increases the continuality of communication between bots and C2 servers, but it also makes it challenging in threat tracking. For instance, if one epoch is taken down or under maintenance, the Emotet actors can keep other epochs running. They can even move bots from one epoch to another according to the report from BeepingComputer.

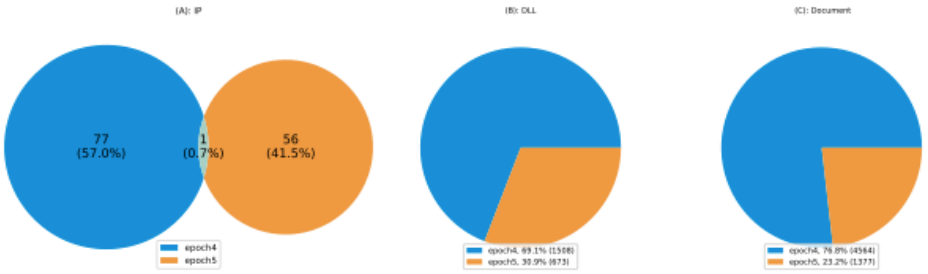


Figure 19: Epoch distribution.

The epoch distribution of IP addresses implies that Epoch 4 has been the main botnet of the recent attacks. Similar observations can be seen from DLL and document epoch distributions, with 69.1% of DLLs and nearly 77% of documents associated with Epoch 4 (see Figure 19 (B) and (C)).

## IP Address-Port Analysis

### IP count distribution

Our analysis shows that the number of IP addresses or IP:port pairs extracted from C2 configuration data of a DLL payload varies from 27 up to 49, which leads to 40 IP addresses or IP:port pairs per DLL on average. In terms of IP address count distribution among all DLL payloads, the top count goes to IP address 217.182.143[.]207, which appeared 2181 times, meaning all DLL payloads contain this IP. According to the IP address lookup, currently there are no hostnames resolving to this IP address. Though we don't know the underlying reason why this IP address has been included in all DLLs (both Epoch 4 and Epoch 5), it could be possible that this host remained compromised all the time during the attacks, or it was added by accident to both epoch botnets. Other IP addresses were used between 14 and 1508 times among the DLL payloads. Figure 20 shows the full distribution of the 134 IP addresses contained in the 2181 DLL payloads.

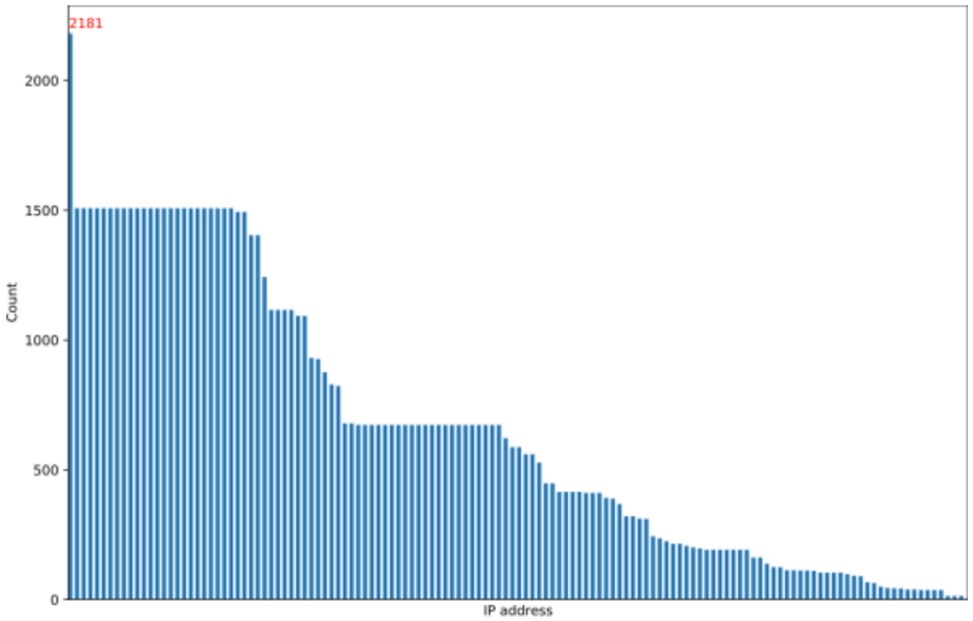


Figure 20: IP address distribution.

**IP Address Set Distribution**

Apart from analyzing the distribution of individual IP addresses, we also examined how often a full set of C2 server IP addresses (or IP:port pairs) within a DLL payload appeared across all DLL payloads. This is achieved by concatenating the sorted IP addresses extracted from the DLL payload as a string and hashing the string. As a result, the distribution of IP list becomes the distribution of hashes of the concatenated IP strings. There are 27 unique hashes of the IP strings, and most of them appeared in the range of 50 to 150 times (see Figure 21).

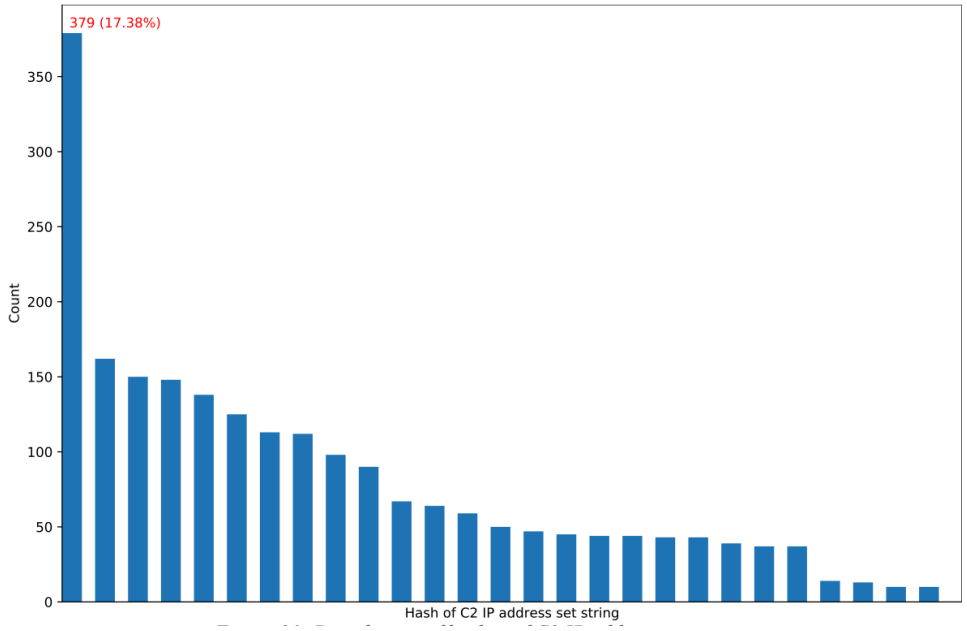


Figure 21: Distribution of hashes of C2 IP address set strings.

**IP Geographic Distribution**

We analyzed the geographic distribution of the 134 IP addresses (see Figure 22) to understand which countries were used to host the Emotet infrastructure. The analysis shows that most of the IP addresses were in the US (over 18%), followed by Germany and France. Other popular regions included South Asia, Brazil, Canada, and the United Kingdom.

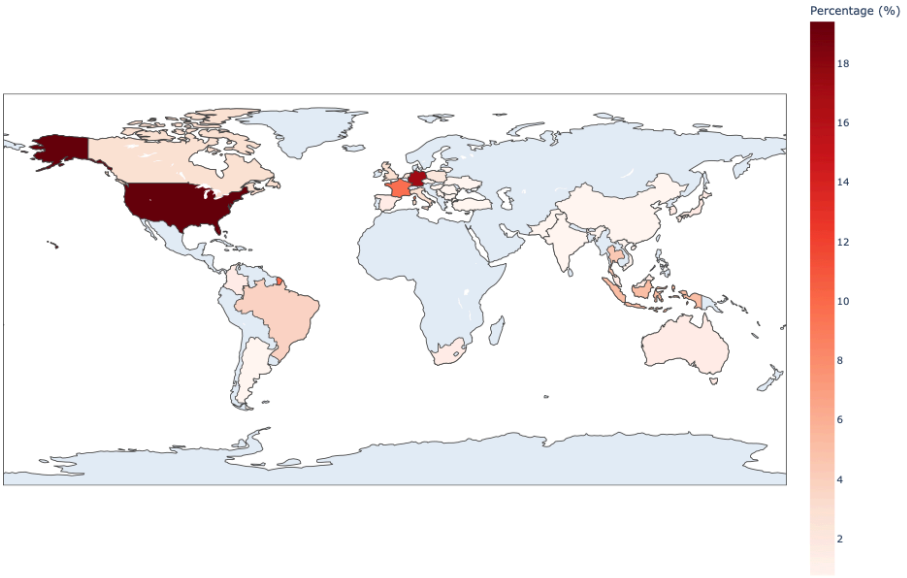


Figure 22: IP distribution map.

**Port Distribution**

Every C2 server IP address comes with a specific port number, which forms a collection of hard-coded IP:port pairs in the C2 configuration data, as previously discussed. There were four commonly used ports found in the 134 IP:port pairs (see Figure 23). The most common port is 8080, which accounted for over 50% of the total count of all ports, followed by the 443 (HTTPS) port. Port 8080 is commonly used as a proxy port, suggesting that most of the C2 servers associated with the IP addresses were likely to be compromised legitimate servers used to proxy the real C2 servers. Using proxies to hide actual C2 servers is common in Emotet attacks. According to the findings of a report published in 2017, Emotet actors run an Nginx reverse proxy on a secondary port (e.g., 8080) of a compromised server, which then relays requests to the actual sever.

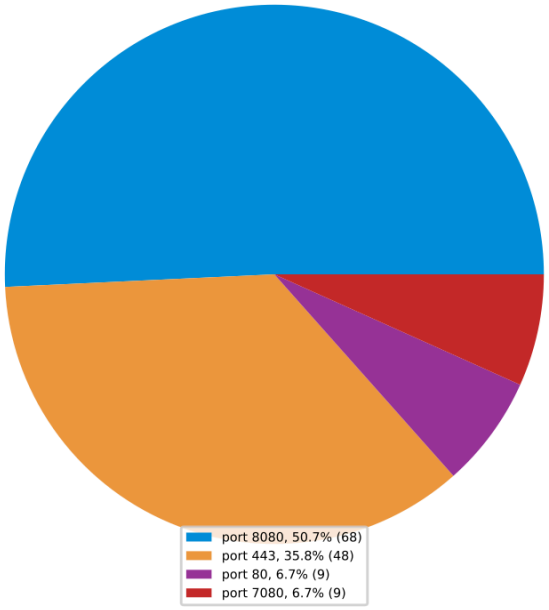


Figure 23: Port distribution.

**JARM Fingerprint Distribution**

JARM is an active Transport Layer Security (TLS) server fingerprinting tool to identify and cluster servers based on their TLS configuration.

In this section, we examine the distribution of JARM fingerprint hashes for the Emotet C2 server IP addresses. At the time of this analysis, most of the servers were offline, but we were able to obtain JARM fingerprints for 118 of the 134 IP addresses by querying the IP addresses on VirusTotal. The remaining 16 IP addresses were missing the JARM fingerprint. The likely reason is that those C2 servers were offline at the time when VirusTotal checked their JARM fingerprints. We assume that the JARM fingerprint hashes obtained from VirusTotal were based on the C2 servers' default standard port 443. To verify this assumption, we scanned one of the C2 IP address-port pairs, 135.148.121[.]246:8080, with the JARM fingerprinting tool (see Figure 24). The tool allows one to specify a specific port (with option *-p*) when fingerprinting a server. If a port is not specified, it uses the default port of the server.

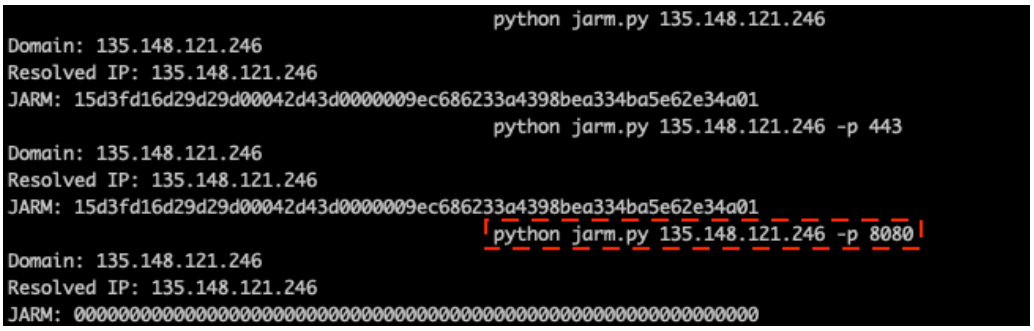


Figure 24: JARM fingerprinting IP address 135.148.121[.]246 with different ports.

As we can see from the figure above, the fingerprint hash (15d3fd16d29d29d00042d43d0000009ec686233a4398bea334ba5e62e34a01) is the same when scanning with the default port and port 443. This is the same JARM hash returned from VirusTotal when querying for the IP address. However, when scanning the IP address with port 8080 (as highlighted in the figure), JARM failed to fingerprint the server (the fingerprint hash string has all zeros). This means the server refused to respond to JARM fingerprinting messages on port 8080, as the port typically used for proxy service was closed (confirmed with Nmap port scanning, see Figure 25).


Figure 25: Port scanning with Nmap.

By using the RiskIQ Community tool, we determined that the IP address belongs to OVH (highlighted in Figure 26). OVH is not an abuse well-known European Internet service provider (ISP) that provides server rental services. As we can see from Figure 26, there are a few domains currently pointing to the IP address since July 2021, which existed before Emotet resurfaced. So, we have a good reason to believe that this is probably a legitimate (compromised) web server.

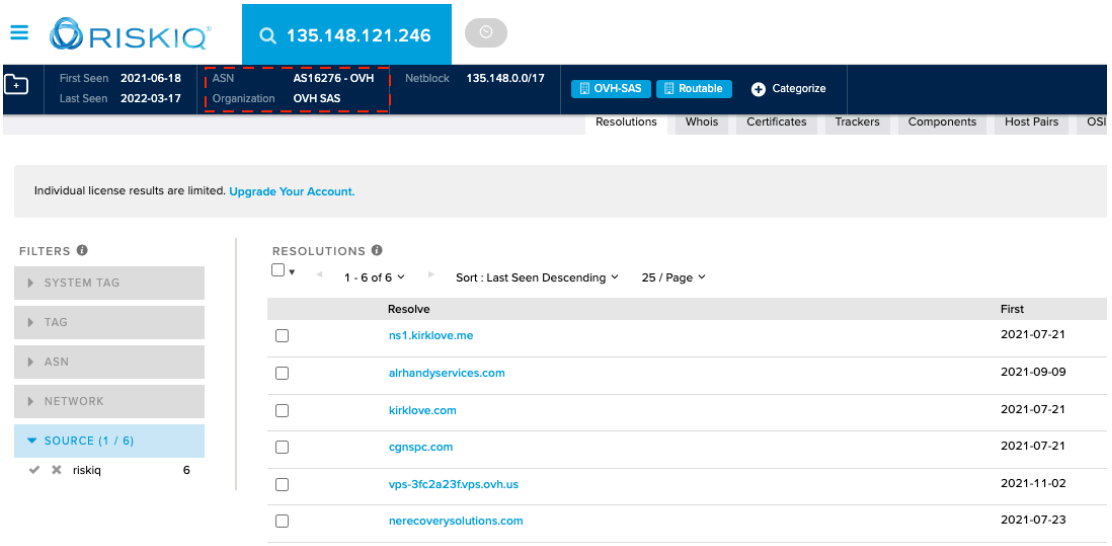


Figure 26: RISKIQ lookup on IP: 135.148.121[.]246.

The findings from the investigation above suggest that if one uses JARM to fingerprint a server without specifying a port number, the resulting fingerprint could be misleading. Different services running on the same server but with different ports can lead to different JARM fingerprints. This also means that when using JARM in threat hunting (such as hunting for C2 servers) one should always specify the corresponding port numbers identified from the C2 configuration.

As a result, we only use the subset of C2 IP:port pairs that reference port 443 when analyzing the JARM fingerprints obtained from VirusTotal. According to the port distribution discussed above (see Figure 23), there are 48 such IP addresses, with 40 of them having JARM fingerprints from VirusTotal. As Figure 27 shows, there are 7 unique JARM fingerprint hashes in total, and 31 IP addresses with port 443 share the same hash (2ad2ad0002ad2ad0002ad2ad2ad2ade1a3c0d7ca6ad8388057924be83dfc6a).

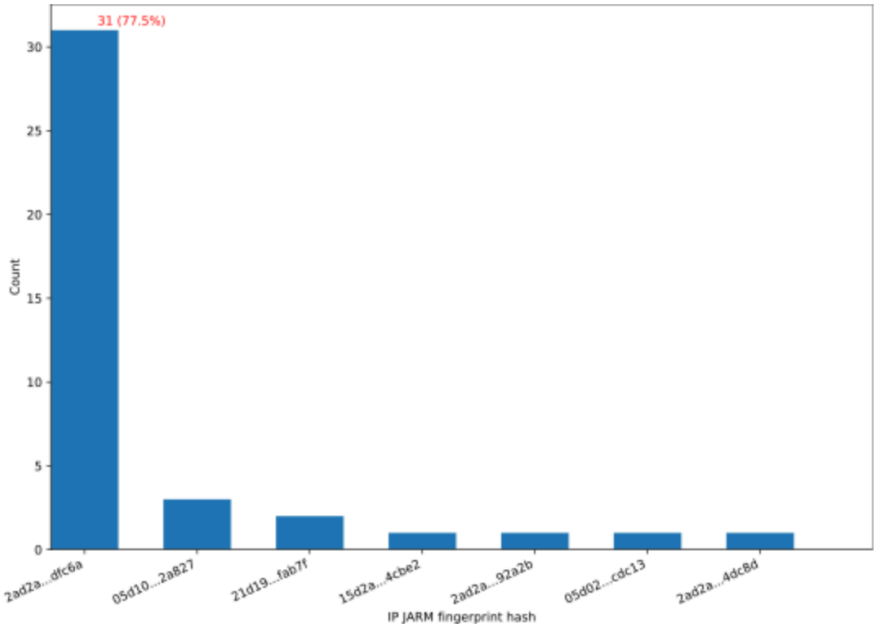


Figure 27: JARM fingerprint hash distribution for IP addresses with port 443.

It is worth noting that, although JARM can be used to identify and cluster servers including malware C2 servers, it can lead to false positives (FPs) if not combined with other intelligence, such as IP address/domain history and reputation. For

instance, a report from Cobalt Strike found that the JARM fingerprint of a Cobalt Strike server was the same as a Java server. In addition, one can evade JARM fingerprinting by changing server-side configuration using a proxy.

### AS Number Distribution

An autonomous system (AS), which is identified by a unique number (e.g., OVH's AS number is 16276, as shown in Figure 26), refers to a large network or group of networks typically operated by a single large organization, such as an ISP or a large company. Therefore, by examining the distribution of AS numbers of the C2 IP addresses we can reveal the organizations that own or operate the corresponding servers.

There are 75 unique AS numbers associated with the 134 IP addresses (see Figure 28). As the distribution shows, the most common AS number (139943 – located in Indonesia) is related to 16 IP addresses, and most of the AS numbers only have one IP address each. The detailed AS numbers for all IP addresses can be found in the IoCs appendix section.

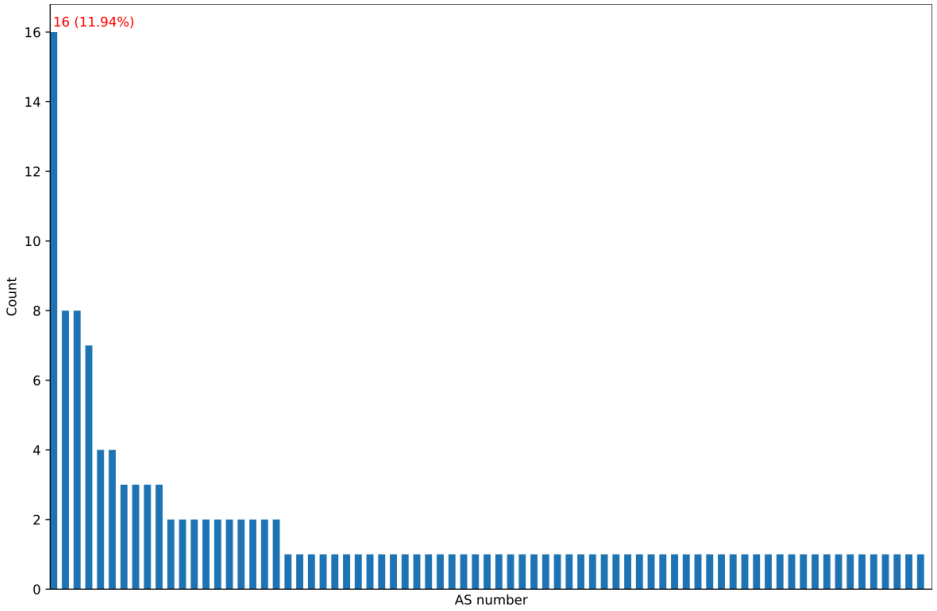


Figure 28: IP address AS number distribution.

## Conclusions

In this report, we first discussed the procedure to statically extract the internal DLL payload embedded in the original Emotet DLL payload delivered by a weaponized document. We then provided the detailed steps to extract the C2 configuration contained in the decrypted DLL. To automate the process, we leverage the NSX Sandbox to dump the decrypted internal DLL from memory during execution and feed it into our C2 configuration extractor for data extraction. Thanks to our fully automated pipeline, we were able to successfully extract the C2 configuration data from all available 2181 DLL payloads dropped by recent Emotet attacks that we observed in our telemetry. We then provided a comprehensive analysis of the configuration data from epoch breakdown to the distributions of IP addresses, JARM fingerprints, ports and AS numbers, with interesting findings. For example, the epoch distribution shows that each epoch has almost distinct C2 servers and Epoch 4 was the main botnet used in recent Emotet campaigns. In terms of port distribution, the secondary port 8080 (commonly used by proxies) was the most used port. The JARM fingerprint distribution of the IP addresses with port 443 implies that over 75% of the 40 evaluated servers shared the same fingerprint.

## Appendix: IoCs

The indicators of compromise identified from this report can be found on VMware TAU's GitHub IoCs repository.