EMOTET Dynamic Configuration Extraction

A tool for the dynamic extraction of EMOTET configurations based on emulation.

By

Remco Sprooten

28 October 2022

Key takeaways

- The EMOTET developers have changed the way they encode their configuration in the 64bit version of the malware.
- Using code emulation we can bypass multiple code obfuscation techniques.
- The use of code emulators in config extractors will become more prevalent in the future.

Additional EMOTET resources

To download the EMOTET configuration extractor, check out our post on the tool:

EMOTET configuration extractor

Preamble

The EMOTET family broke onto the malware scene as a modular banking trojan in 2014, focused on harvesting and exfiltrating bank account information by inspecting traffic. EMOTET has been adapted as an early-stage implant used to load other malware families, such as QAKBOT, TRICKBOT, and RYUK. While multiple EMOTET campaigns have been dismantled by international law enforcement entities, it has continued to operate as one of the most prolific cybercrime operations.

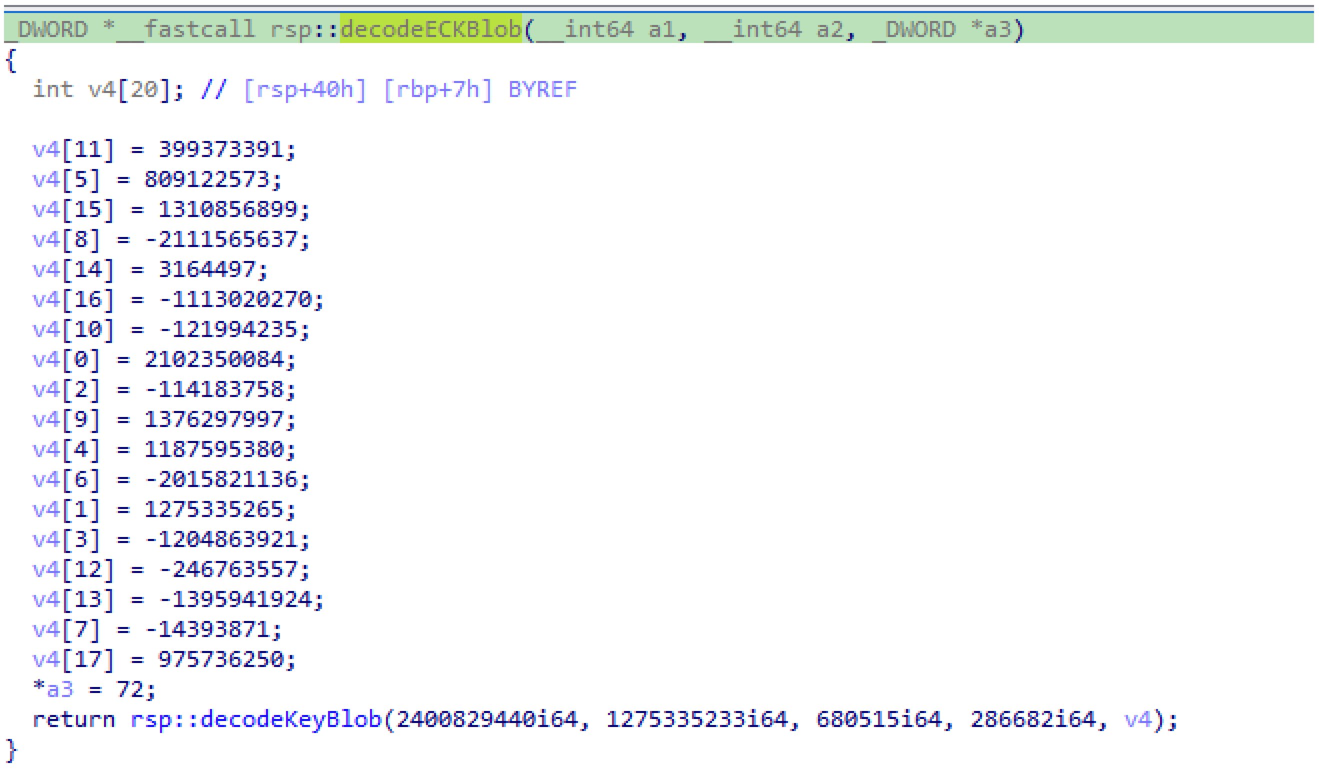
For the last several months, Elastic Security has observed the EMOTET developers transition to a 64-bit version of their malware. While this change does not seem to impact the core functionality of the samples we have witnessed, we did notice a change in how the configuration and strings are obfuscated. In earlier versions of EMOTET, the configuration was stored in an encrypted form in the **.data** section of the binary. In the newer versions the configuration is calculated at runtime. The information we need to extract the configuration from the binary is thus hidden within the actual code.

In the next sections, we'll discuss the following as it relates to 64-bit EMOTET samples:

- EMOTET encryption mechanisms
- Reviewing the EMOTET C2 list
- Interesting EMOTET strings
- The EMOTET configuration extractor utility

Encryption keys

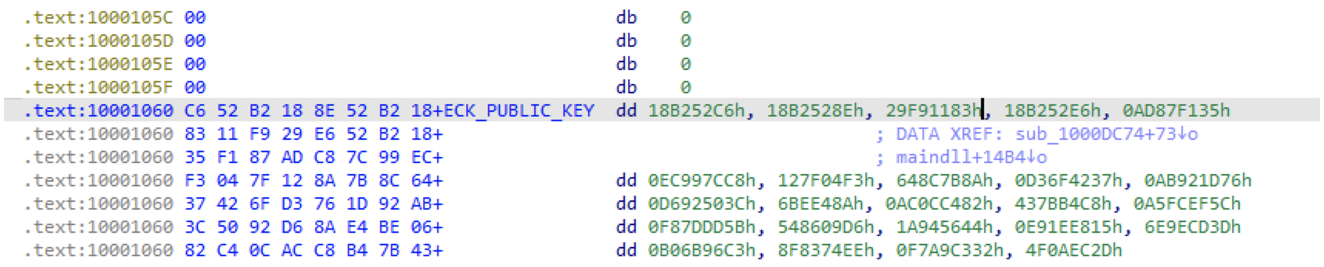
EMOTET uses embedded Elliptic Curve Cryptography (ECC) public keys to encrypt their network communication. While in previous versions, the keys would be stored in an XOR-encrypted blob, now the content is calculated at runtime.



```
_DWORD *__fastcall rsp::decodeECKBlob(__int64 a1, __int64 a2, _DWORD *a3)
{
  int v4[20]; // [rsp+40h] [rbp+7h] BYREF

  v4[11] = 399373391;
  v4[5] = 809122573;
  v4[15] = 1310856899;
  v4[8] = -2111565637;
  v4[14] = 3164497;
  v4[16] = -1113020270;
  v4[10] = -121994235;
  v4[0] = 2102350084;
  v4[2] = -114183758;
  v4[9] = 1376297997;
  v4[4] = 1187595380;
  v4[6] = -2015821136;
  v4[1] = 1275335265;
  v4[3] = -1204863921;
  v4[12] = -246763557;
  v4[13] = -1395941924;
  v4[7] = -14393871;
  v4[17] = 975736250;
  *a3 = 72;
  return rsp::decodeKeyBlob(2400829440i64, 1275335233i64, 680515i64, 286682i64, v4);
}
```

Encoded Encryption Key blob in 64-bit version

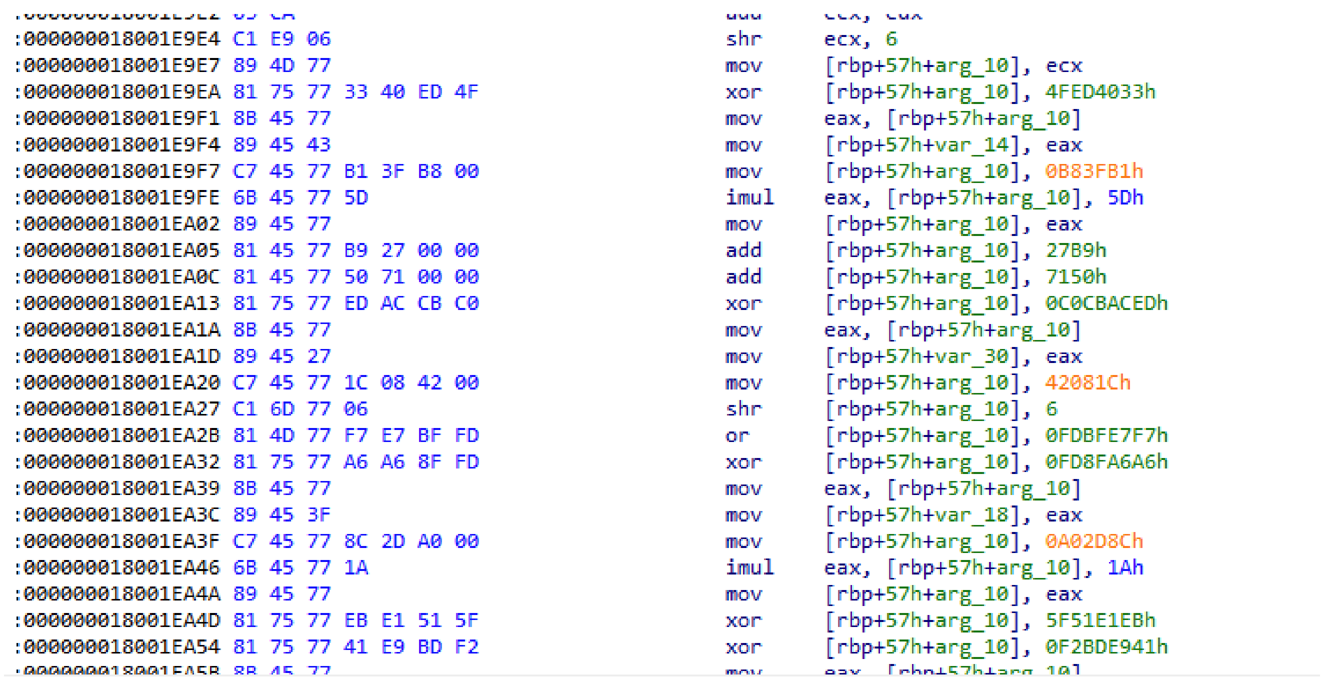
In comparison the previous versions of EMOTET would store an encrypted version of the key data in the **.text** section of the binary.



```
 .text:1000105C 00                                 db    0
 .text:1000105D 00                                 db    0
 .text:1000105E 00                                 db    0
 .text:1000105F 00                                 db    0
 .text:10001060 C6 52 B2 18 8E 52 B2 18+ECK_PUBLIC_KEY dd 18B252C6h, 18B2528Eh, 29F91183h, 18B252E6h, 0AD87F135h
 .text:10001060 83 11 F9 29 E6 52 B2 18+                                          ; DATA XREF: sub_1000DC74+73↓o
 .text:10001060 35 F1 87 AD C8 7C 99 EC+                                          ; maindll+14B4↓o
 .text:10001060 F3 04 7F 12 8A 7B 8C 64+           dd 0EC997CC8h, 127F04F3h, 648C7B8Ah, 0D36F4237h, 0AB921D76h
 .text:10001060 37 42 6F D3 76 1D 92 AB+           dd 0D692503Ch, 6BEE48Ah, 0AC0CC482h, 437BB4C8h, 0A5FCEF5Ch
 .text:10001060 3C 50 92 D6 8A E4 BE 06+           dd 0F87DDD5Bh, 548609D6h, 1A945644h, 0E91EE815h, 6E9ECD3Dh
 .text:10001060 82 C4 0C AC C8 B4 7B 43+           dd 0B06B96C3h, 8F8374EEh, 0F7A9C332h, 4F0AEC2Dh
```

Embedded key data in previous version of the malware

In order to make it harder for security researchers to find the given code the malware uses Mixed Boolean-Arithmetic (MBA) as one of its obfuscation techniques. It transforms constants and simple expressions into expressions that contain a mix of Boolean and arithmetic operations.



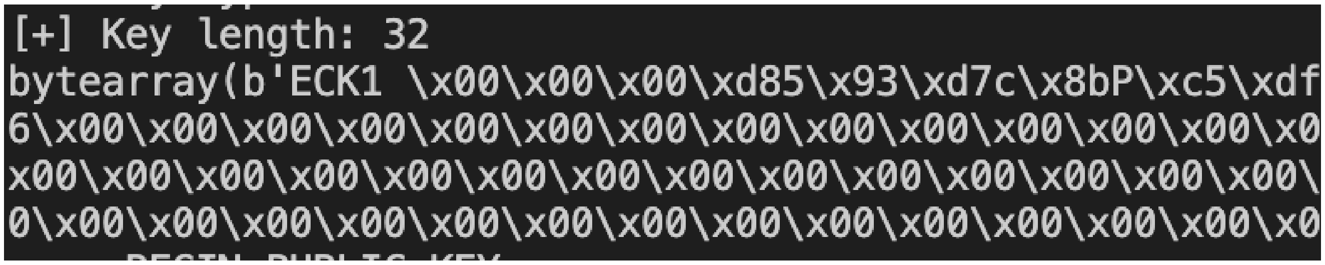
```
:000000018001E9E4 C1 E9 06              shr     ecx, 6
:000000018001E9E7 89 4D 77              mov     [rbp+57h+arg_10], ecx
:000000018001E9EA 81 75 77 33 40 ED 4F  xor     [rbp+57h+arg_10], 4FED4033h
:000000018001E9F1 8B 45 77              mov     eax, [rbp+57h+arg_10]
:000000018001E9F4 89 45 43              mov     [rbp+57h+var_14], eax
:000000018001E9F7 C7 45 77 B1 3F B8 00  mov     [rbp+57h+arg_10], 0B83FB1h
:000000018001E9FE 6B 45 77 5D           imul    eax, [rbp+57h+arg_10], 5Dh
:000000018001EA02 89 45 77              mov     [rbp+57h+arg_10], eax
:000000018001EA05 81 45 77 B9 27 00 00  add     [rbp+57h+arg_10], 27B9h
:000000018001EA0C 81 45 77 50 71 00 00  add     [rbp+57h+arg_10], 7150h
:000000018001EA13 81 75 77 ED AC CB C0  xor     [rbp+57h+arg_10], 0C0CBACEDh
:000000018001EA1A 8B 45 77              mov     eax, [rbp+57h+arg_10]
:000000018001EA1D 89 45 27              mov     [rbp+57h+var_30], eax
:000000018001EA20 C7 45 77 1C 08 42 00  mov     [rbp+57h+arg_10], 42081Ch
:000000018001EA27 C1 6D 77 06           shr     [rbp+57h+arg_10], 6
:000000018001EA2B 81 4D 77 F7 E7 BF FD  or      [rbp+57h+arg_10], 0FDBFE7F7h
:000000018001EA32 81 75 77 A6 A6 8F FD  xor     [rbp+57h+arg_10], 0FD8FA6A6h
:000000018001EA39 8B 45 77              mov     eax, [rbp+57h+arg_10]
:000000018001EA3C 89 45 3F              mov     [rbp+57h+var_18], eax
:000000018001EA3F C7 45 77 8C 2D A0 00  mov     [rbp+57h+arg_10], 0A02D8Ch
:000000018001EA46 6B 45 77 1A           imul    eax, [rbp+57h+arg_10], 1Ah
:000000018001EA4A 89 45 77              mov     [rbp+57h+arg_10], eax
:000000018001EA4D 81 75 77 EB E1 51 5F  xor     [rbp+57h+arg_10], 5F51E1EBh
:000000018001EA54 81 75 77 41 E9 BD F2  xor     [rbp+57h+arg_10], 0F2BDE941h
```

Example of Mixed Boolean-Arithmetic

In this example, an array of constants is instantiated, but looking at the assembly we see that every constant is calculated at runtime. This method makes it challenging to develop a signature to target this function.

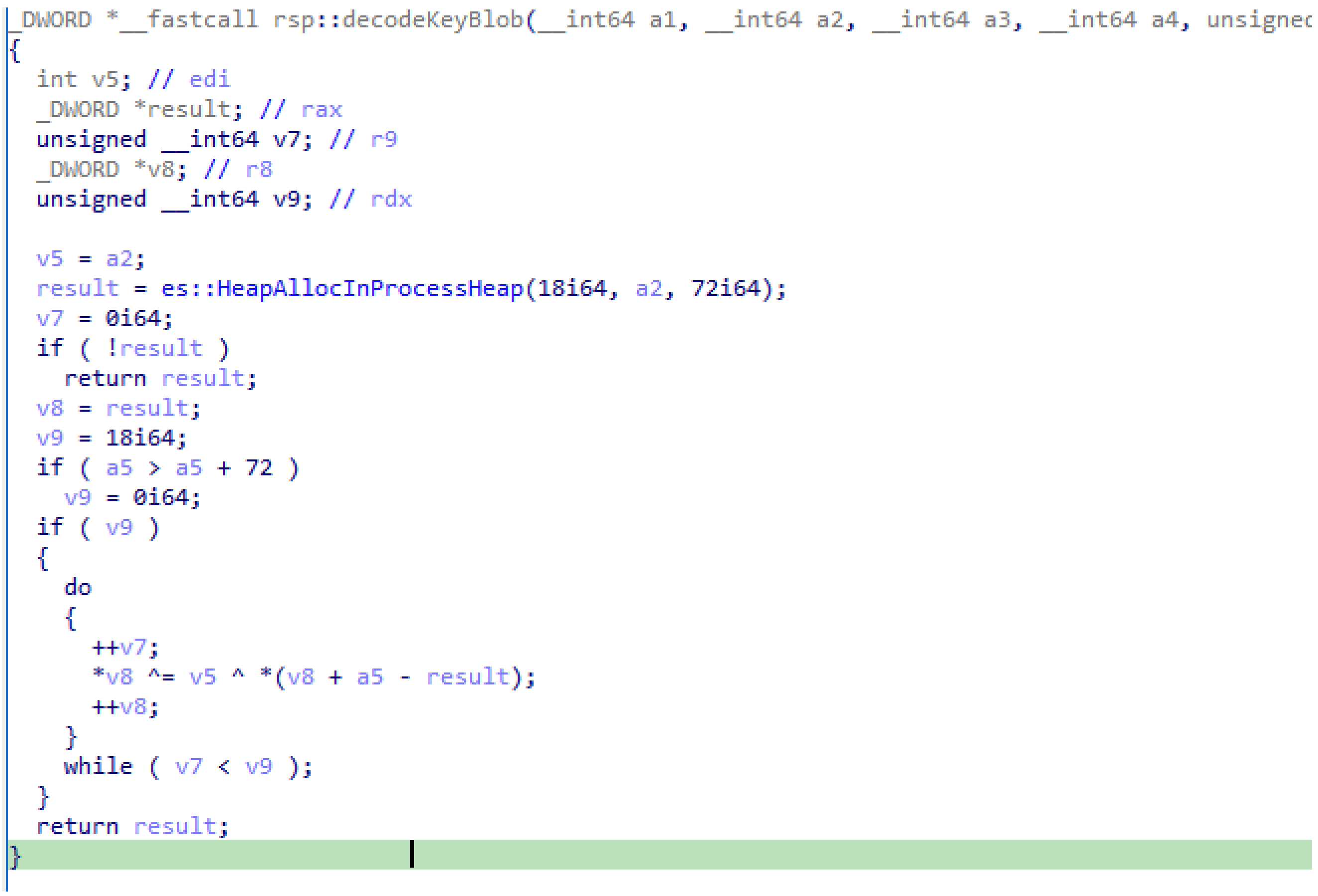
We noticed that both the Elliptic Curve Diffie-Hellman (ECDH) and Elliptic Curve Digital Signature Algorithm (ECDSA) keys use the same function to decode the contents.

The ECDH key (which you can recognize by its magic ECK1 bytes) is used for encryption purposes while the ECDSA key (ECC1) is used for verifying the C2 server's responses.



```
[+] Key length: 32
bytearray(b'ECK1 \x00\x00\x00\xd85\x93\xd7c\x8bP\xc5\xdf
6\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
```

ECK1 magic bytes at the start of the key data



```
_DWORD *__fastcall rsp::decodeKeyBlob(__int64 a1, __int64 a2, __int64 a3, __int64 a4, unsignec
{
  int v5; // edi
  _DWORD *result; // rax
  unsigned __int64 v7; // r9
  _DWORD *v8; // r8
  unsigned __int64 v9; // rdx

  v5 = a2;
  result = es::HeapAllocInProcessHeap(18i64, a2, 72i64);
  v7 = 0i64;
  if ( !result )
    return result;
  v8 = result;
  v9 = 18i64;
  if ( a5 > a5 + 72 )
    v9 = 0i64;
  if ( v9 )
  {
    do
    {
      ++v7;
      *v8 ^= v5 ^ *(v8 + a5 - result);
      ++v8;
    }
    while ( v7 < v9 );
  }
  return result;
}
```

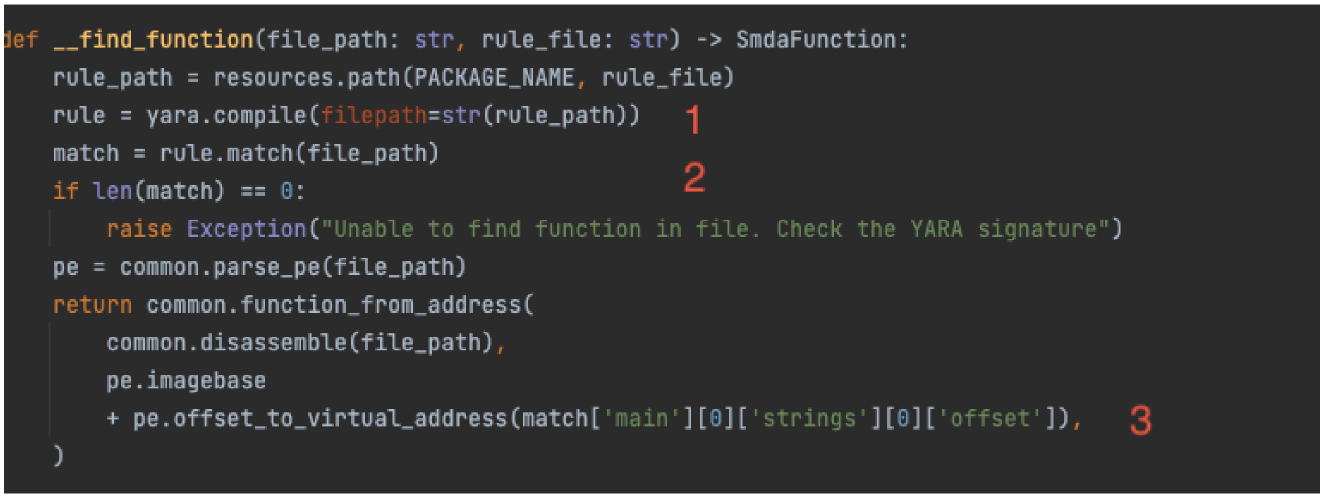
Decoding algorithm for the key material

By leveraging a YARA signature to find the location of this decode function within the EMOTET binary we can observe the following process:

1. Find the decoding algorithm within the binary.
2. Locate any Cross References (Xrefs) to the decoding function.
3. Emulate the function that calls the decoding function.
4. Read the resulting data from memory.

As we mentioned, we first find the function in the binary by using YARA. The signature is provided at the end of this article. It is worth pointing out that these yara signatures are used to identify locations in the binary but are, in their current form, not usable to identify EMOTET samples.

In order to automatically retrieve the data from multiple samples, we created a configuration extractor. In the snippets below, we will demonstrate, in a high level fashion, how we collect the configuration information from the malware samples.

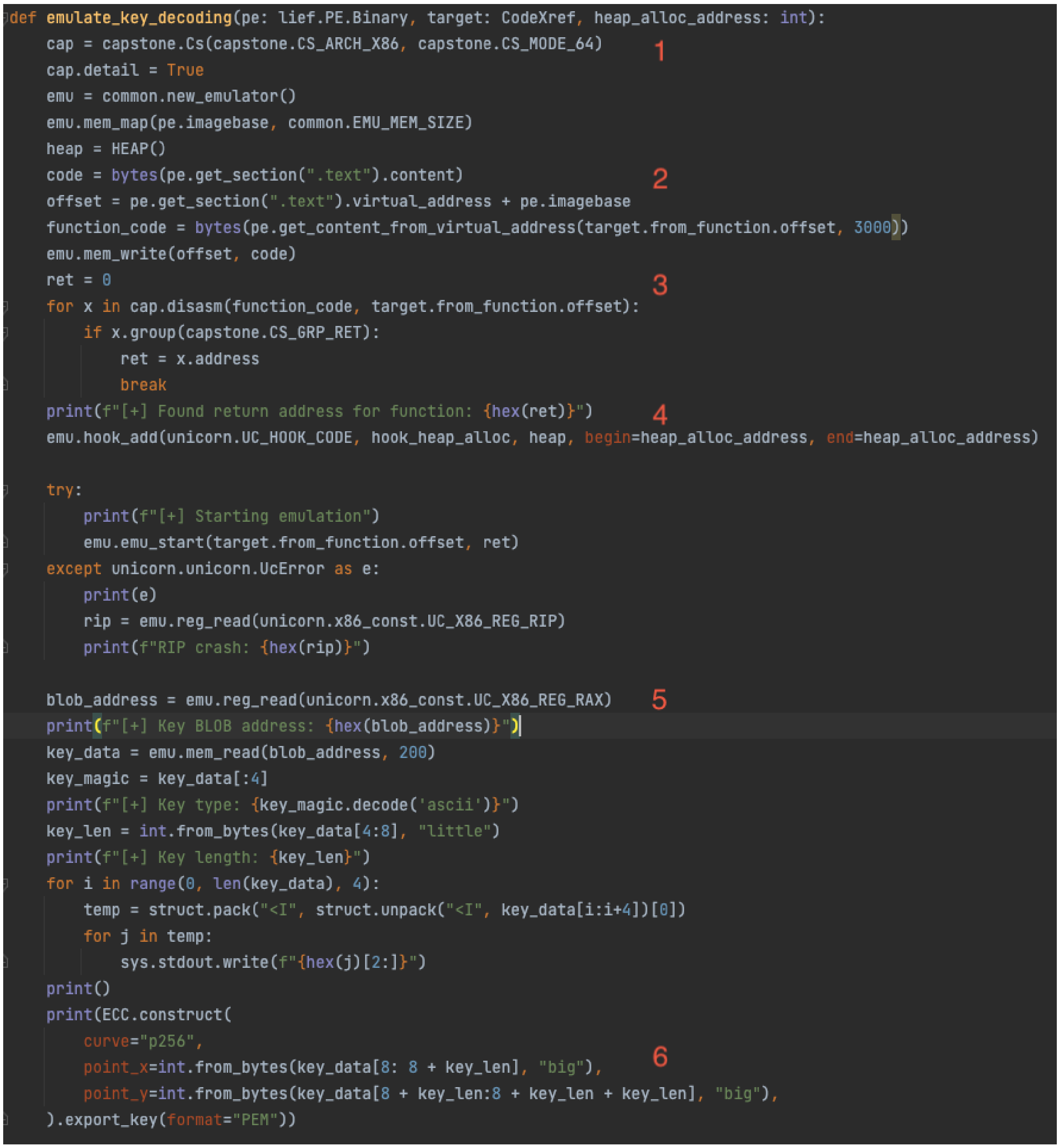


```
def __find_function(file_path: str, rule_file: str) -> SmdaFunction:
    rule_path = resources.path(PACKAGE_NAME, rule_file)
    rule = yara.compile(filepath=str(rule_path))      1
    match = rule.match(file_path)
    if len(match) == 0:                                2
        raise Exception("Unable to find function in file. Check the YARA signature")
    pe = common.parse_pe(file_path)
    return common.function_from_address(
        common.disassemble(file_path),
        pe.imagebase
        + pe.offset_to_virtual_address(match['main'][0]['strings'][0]['offset']),    3
    )
```

Python code to find the start of a function
In the above code snippet:

1. First load the YARA signature.
2. Try to find a match, and if a signature is found in the file.
3. Calculate the function offset based on the offset in the file.

In order to locate the Xrefs to this function, we use the excellent SMDA decompiler. After locating the Xrefs, we can start the emulation process using the CPU emulator, Unicorn.

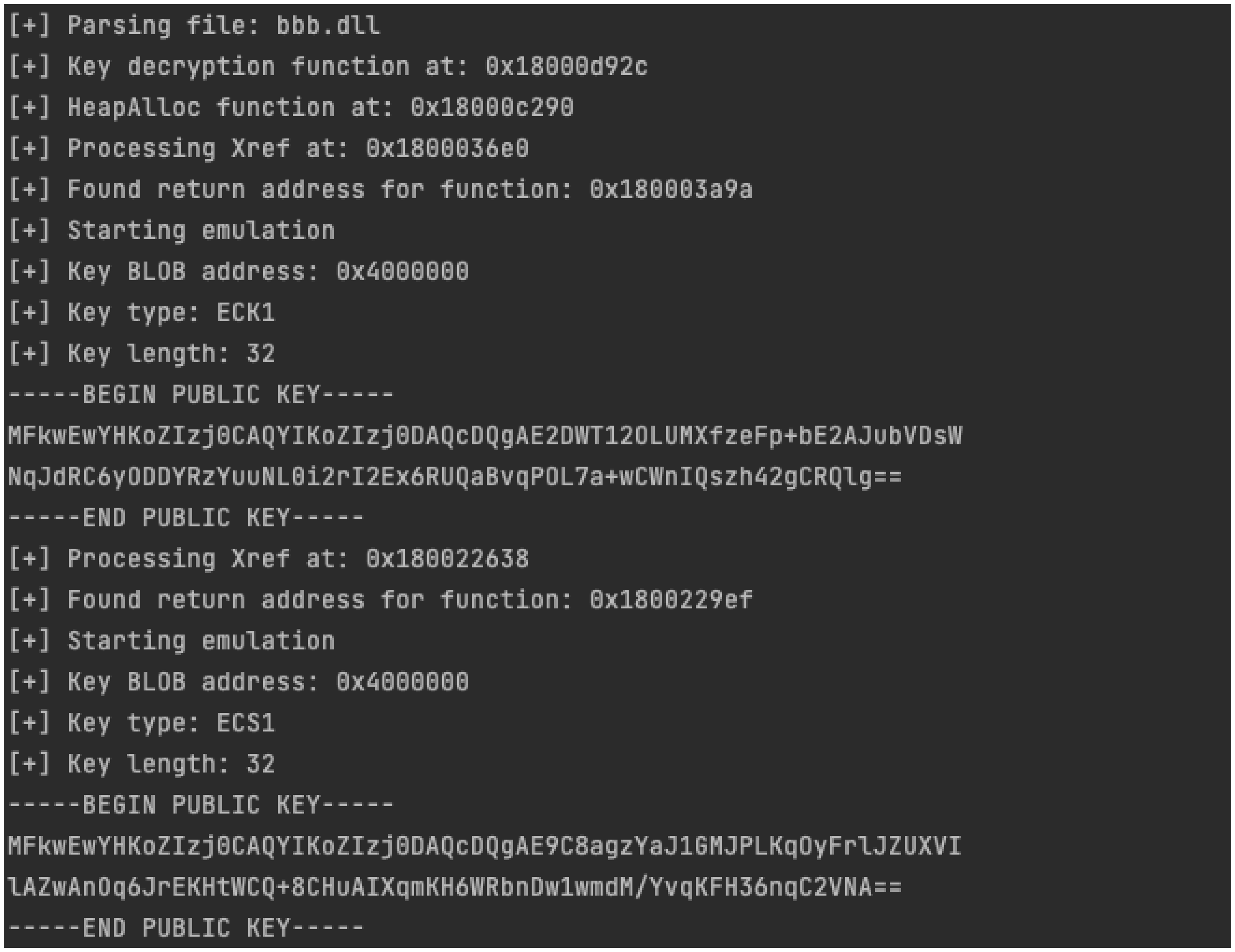


```python
def emulate_key_decoding(pe: lief.PE.Binary, target: CodeXref, heap_alloc_address: int):
    cap = capstone.Cs(capstone.CS_ARCH_X86, capstone.CS_MODE_64)
    cap.detail = True
    emu = common.new_emulator()
    emu.mem_map(pe.imagebase, common.EMU_MEM_SIZE)
    heap = HEAP()
    code = bytes(pe.get_section(".text").content)
    offset = pe.get_section(".text").virtual_address + pe.imagebase
    function_code = bytes(pe.get_content_from_virtual_address(target.from_function.offset, 3000))
    emu.mem_write(offset, code)
    ret = 0
    for x in cap.disasm(function_code, target.from_function.offset):
        if x.group(capstone.CS_GRP_RET):
            ret = x.address
            break
    print(f"[+] Found return address for function: {hex(ret)}")
    emu.hook_add(unicorn.UC_HOOK_CODE, hook_heap_alloc, heap, begin=heap_alloc_address, end=heap_alloc_address)

    try:
        print(f"[+] Starting emulation")
        emu.emu_start(target.from_function.offset, ret)
    except unicorn.unicorn.UcError as e:
        print(e)
        rip = emu.reg_read(unicorn.x86_const.UC_X86_REG_RIP)
        print(f"RIP crash: {hex(rip)}")

    blob_address = emu.reg_read(unicorn.x86_const.UC_X86_REG_RAX)
    print(f"[+] Key BLOB address: {hex(blob_address)}")
    key_data = emu.mem_read(blob_address, 200)
    key_magic = key_data[:4]
    print(f"[+] Key type: {key_magic.decode('ascii')}")
    key_len = int.from_bytes(key_data[4:8], "little")
    print(f"[+] Key length: {key_len}")
    for i in range(0, len(key_data), 4):
        temp = struct.pack("<I", struct.unpack("<I", key_data[i:i+4])[0])
        for j in temp:
            sys.stdout.write(f"{hex(j)[2:]}")
    print()
    print(ECC.construct(
        curve="p256",
        point_x=int.from_bytes(key_data[8: 8 + key_len], "big"),
        point_y=int.from_bytes(key_data[8 + key_len:8 + key_len + key_len], "big"),
    ).export_key(format="PEM"))
```

Python code used to emulate decoding functions

1. Initialize the Unicorn emulator.
2. Load the executable code from the PE file into memory.
3. Disassemble the function to find the return and the end of the execution.
4. The binary will try to use the windows HeapAlloc API to allocate space for the decoded data. Since we don't want to emulate any windows API's, as this would add unnecessary complexity, we hook to code so that we can allocate space ourselves.
5. After the emulation has run the 64-bit “long size” register (RAX), it will contain a pointer to the key data in memory.
6. To present the key in a more readable way, we convert it to the standard PEM format.

By emulating the parts of the binary that we are interested in, we no longer have to statically defeat the obfuscation in order to retrieve the hidden contents. This approach adds a level of complexity to the creation of config extractors. However, since malware authors are adding ever more obfuscation, there is a need for a generic approach to defeating these techniques.



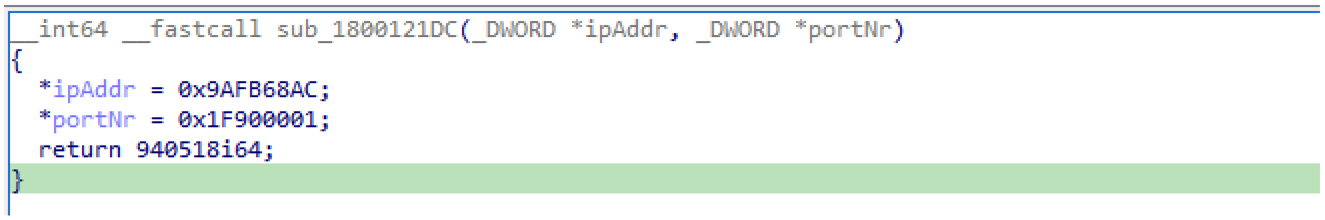
```
[+] Parsing file: bbb.dll
[+] Key decryption function at: 0x18000d92c
[+] HeapAlloc function at: 0x18000c290
[+] Processing Xref at: 0x1800036e0
[+] Found return address for function: 0x180003a9a
[+] Starting emulation
[+] Key BLOB address: 0x4000000
[+] Key type: ECK1
[+] Key length: 32
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE2DWT12OLUMXfzeFp+bE2AJubVDsW
NqJdRC6yODDYRzYuuNL0i2rI2Ex6RUQaBvqPOL7a+wCWnIQszh42gCRQlg==
-----END PUBLIC KEY-----
[+] Processing Xref at: 0x180022638
[+] Found return address for function: 0x1800229ef
[+] Starting emulation
[+] Key BLOB address: 0x4000000
[+] Key type: ECS1
[+] Key length: 32
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE9C8agzYaJ1GMJPLKqOyFrlJZUXVI
lAZwAnOq6JrEKHtWCQ+8CHuAIXqmKH6WRbnDw1wmdM/YvqKFH36nqC2VNA==
-----END PUBLIC KEY-----
```

Example of the extractor used to find key material

C2 server list

An important part of tracking malware families is to get new insights by identifying and discovering which C2 servers they use to operate their network.

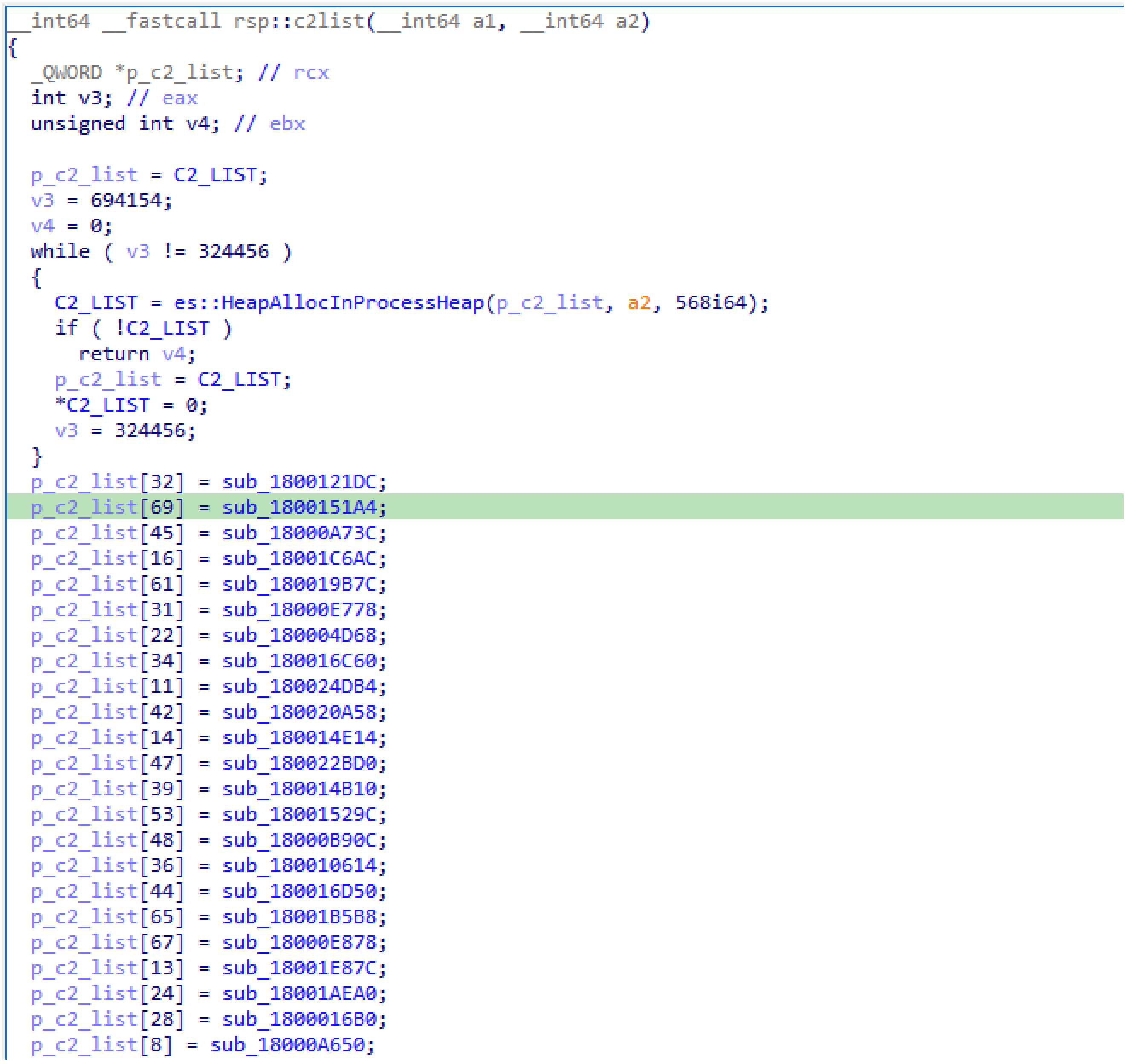
In the 64-bit versions of EMOTET, we see that the IP and port information of the C2 servers are also dynamically calculated at runtime. Every C2 server is represented by a function that calculates and returns a value for the IP address and the port number.



```
__int64 __fastcall sub_1800121DC(_DWORD *ipAddr, _DWORD *portNr)
{
  *ipAddr = 0x9AFB68AC;
  *portNr = 0x1F900001;
  return 940518i64;
}
```

Examples of encoded IP/port combination

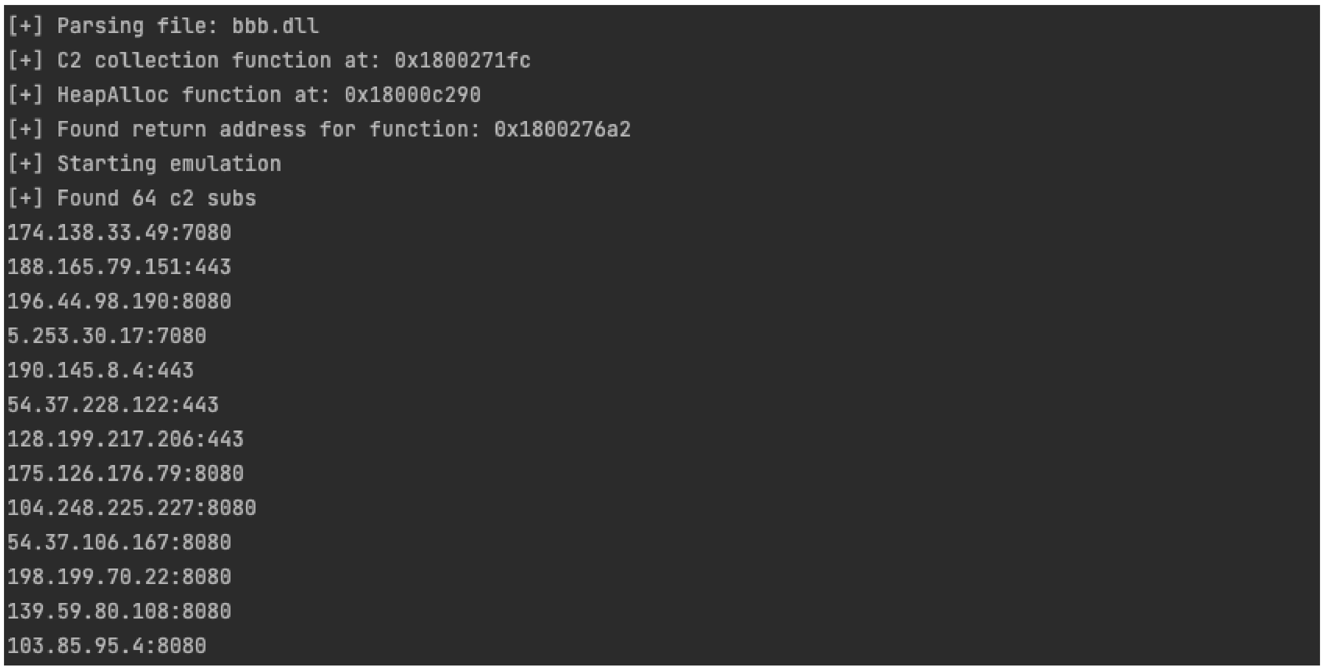
These functions don’t have a direct cross reference available for searching. However, a procedure references all the C2 functions and creates the **p_c2_list** array of pointers.



```
__int64 __fastcall rsp::c2list(__int64 a1, __int64 a2)
{
  _QWORD *p_c2_list; // rcx
  int v3; // eax
  unsigned int v4; // ebx

  p_c2_list = C2_LIST;
  v3 = 694154;
  v4 = 0;
  while ( v3 != 324456 )
  {
    C2_LIST = es::HeapAllocInProcessHeap(p_c2_list, a2, 568i64);
    if ( !C2_LIST )
      return v4;
    p_c2_list = C2_LIST;
    *C2_LIST = 0;
    v3 = 324456;
  }
  p_c2_list[32] = sub_1800121DC;
  p_c2_list[69] = sub_1800151A4;
  p_c2_list[45] = sub_18000A73C;
  p_c2_list[16] = sub_18001C6AC;
  p_c2_list[61] = sub_180019B7C;
  p_c2_list[31] = sub_18000E778;
  p_c2_list[22] = sub_180004D68;
  p_c2_list[34] = sub_180016C60;
  p_c2_list[11] = sub_180024DB4;
  p_c2_list[42] = sub_180020A58;
  p_c2_list[14] = sub_180014E14;
  p_c2_list[47] = sub_180022BD0;
  p_c2_list[39] = sub_180014B10;
  p_c2_list[53] = sub_18001529C;
  p_c2_list[48] = sub_18000B90C;
  p_c2_list[36] = sub_180010614;
  p_c2_list[44] = sub_180016D50;
  p_c2_list[65] = sub_18001B5B8;
  p_c2_list[67] = sub_18000E878;
  p_c2_list[13] = sub_18001E87C;
  p_c2_list[24] = sub_18001AEA0;
  p_c2_list[28] = sub_1800016B0;
  p_c2_list[8] = sub_18000A650;
```

C2 server list

After that, we can emulate every C2-server function individually to retrieve the IP and port combination as seen below.

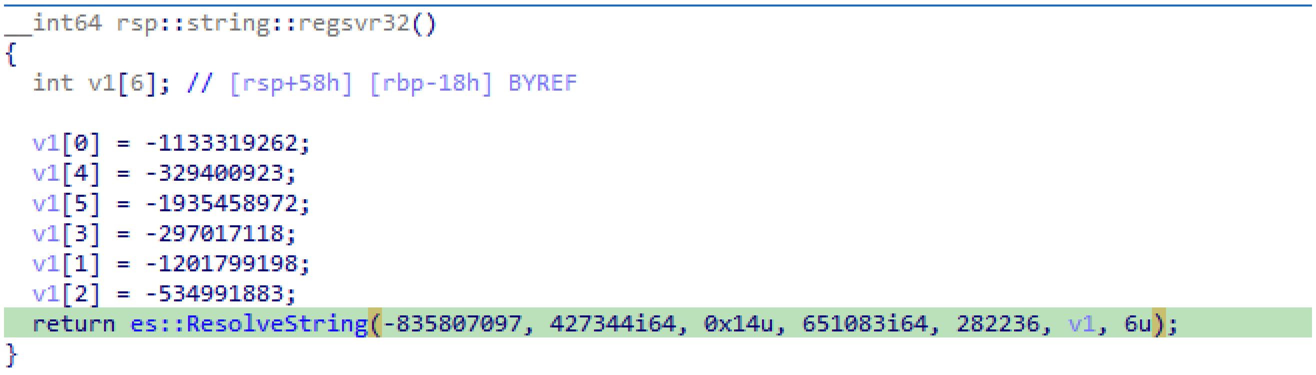


```
[+] Parsing file: bbb.dll
[+] C2 collection function at: 0x1800271fc
[+] HeapAlloc function at: 0x18000c290
[+] Found return address for function: 0x1800276a2
[+] Starting emulation
[+] Found 64 c2 subs
174.138.33.49:7080
188.165.79.151:443
196.44.98.190:8080
5.253.30.17:7080
190.145.8.4:443
54.37.228.122:443
128.199.217.206:443
175.126.176.79:8080
104.248.225.227:8080
54.37.106.167:8080
198.199.70.22:8080
139.59.80.108:8080
103.85.95.4:8080
```

Example of the extractor used to find C2 server list

Strings

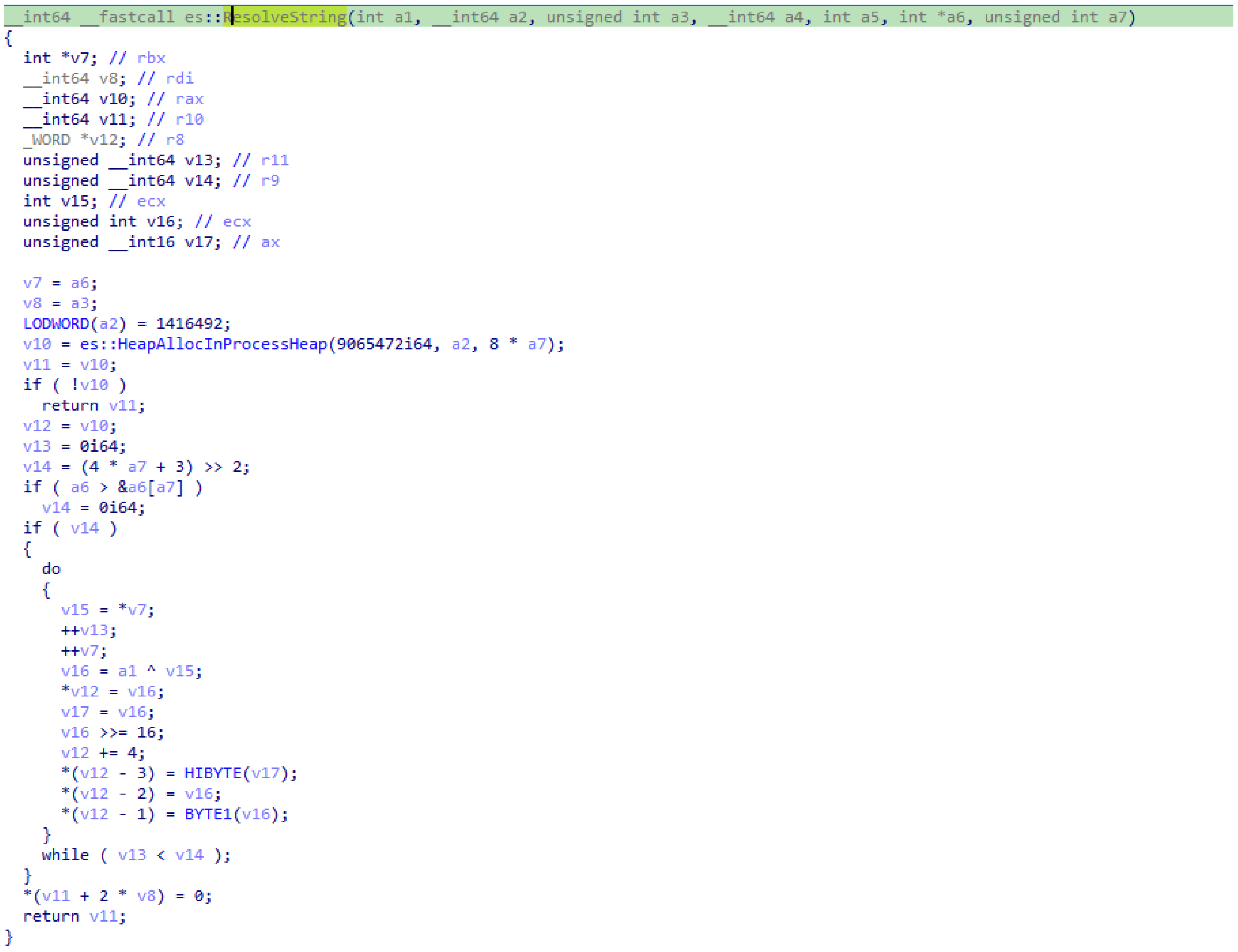
The same method is applied to the use of strings in memory. Every string has its own function. In the following example, the function would return a pointer to the string **%s\regsvr32.exe "%s"**.



```
__int64 rsp::string::regsvr32()
{
  int v1[6]; // [rsp+58h] [rbp-18h] BYREF

  v1[0] = -1133319262;
  v1[4] = -329400923;
  v1[5] = -1935458972;
  v1[3] = -297017118;
  v1[1] = -1201799198;
  v1[2] = -534991883;
  return es::ResolveString(-835807097, 427344i64, 0x14u, 651083i64, 282236, v1, 6u);
}
```

Encoded string

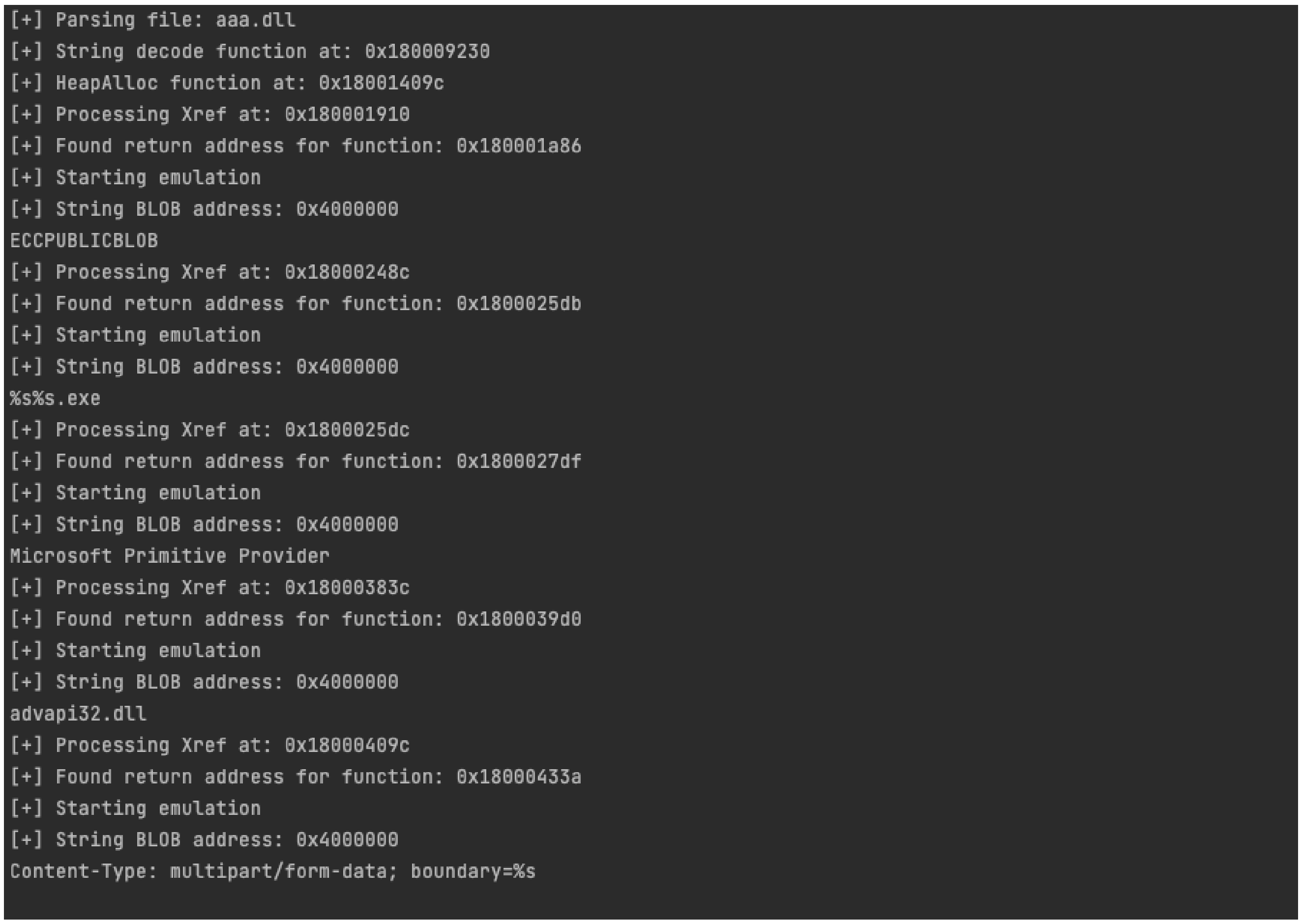
All of the EMOTET strings share a common function to decode or resolve the string at runtime. In the sample that we are analyzing here, the string resolver function is referenced 29 times.



```
__int64 __fastcall es::ResolveString(int a1, __int64 a2, unsigned int a3, __int64 a4, int a5, int *a6, unsigned int a7)
{
  int *v7; // rbx
  __int64 v8; // rdi
  __int64 v10; // rax
  __int64 v11; // r10
  _WORD *v12; // r8
  unsigned __int64 v13; // r11
  unsigned __int64 v14; // r9
  int v15; // ecx
  unsigned int v16; // ecx
  unsigned __int16 v17; // ax

  v7 = a6;
  v8 = a3;
  LODWORD(a2) = 1416492;
  v10 = es::HeapAllocInProcessHeap(9065472i64, a2, 8 * a7);
  v11 = v10;
  if ( !v10 )
    return v11;
  v12 = v10;
  v13 = 0i64;
  v14 = (4 * a7 + 3) >> 2;
  if ( a6 > &a6[a7] )
    v14 = 0i64;
  if ( v14 )
  {
    do
    {
      v15 = *v7;
      ++v13;
      ++v7;
      v16 = a1 ^ v15;
      *v12 = v16;
      v17 = v16;
      v16 >>= 16;
      v12 += 4;
      *(v12 - 3) = HIBYTE(v17);
      *(v12 - 2) = v16;
      *(v12 - 1) = BYTE1(v16);
    }
    while ( v13 < v14 );
  }
  *(v11 + 2 * v8) = 0;
  return v11;
}
```

String decoding algorithm

This allows us to follow the same approach as noted earlier in order to decode all of the EMOTET strings. We pinpoint the string decoding function using YARA, find the cross-references, and emulate the resulting functions.



```
[+] Parsing file: aaa.dll
[+] String decode function at: 0x180009230
[+] HeapAlloc function at: 0x18001409c
[+] Processing Xref at: 0x180001910
[+] Found return address for function: 0x180001a86
[+] Starting emulation
[+] String BLOB address: 0x4000000
ECCPUBLICBLOB
[+] Processing Xref at: 0x18000248c
[+] Found return address for function: 0x1800025db
[+] Starting emulation
[+] String BLOB address: 0x4000000
%s%s.exe
[+] Processing Xref at: 0x1800025dc
[+] Found return address for function: 0x1800027df
[+] Starting emulation
[+] String BLOB address: 0x4000000
Microsoft Primitive Provider
[+] Processing Xref at: 0x18000383c
[+] Found return address for function: 0x1800039d0
[+] Starting emulation
[+] String BLOB address: 0x4000000
advapi32.dll
[+] Processing Xref at: 0x18000409c
[+] Found return address for function: 0x18000433a
[+] Starting emulation
[+] String BLOB address: 0x4000000
Content-Type: multipart/form-data; boundary=%s
```

Example of the extractor used to find strings

Configuration extractor

Automating the payload extraction from EMOTET is a crucial aspect of threat hunting as it gives visibility of the campaign and the malware deployed by the threat actors, enabling practitioners to discover new unknown samples in a timely manner.

```
% emotet-config-extractor --help
usage: Emotet Configuration Extractor [-h] (-f FILE | -d DIRECTORY) [-k] [-c] [-s] [-
a]

options:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  Emotet sample path
  -d DIRECTORY, --directory DIRECTORY
                        Emotet samples folder
  -k                    Extract Encryption keys
  -c                    Extract C2 information
  -s                    Extract strings
  -a                    Extract strings (ascii)Read more
```

Our extractor takes either a directory of samples with **-d** option or **-f** for a single sample and then can output parts of the configuration of note, specifically:

- **-k**: extract the encryption keys
- **-c**: extract the C2 information
- **-s**: extract the wide-character strings
- **-a**: extract the ASCII character stings

EMOTET uses a different routine for decoding wide and ASCII strings. That is why the extractor provides flags to extract them separately.

The C2 information displays a list of IP addresses found in the sample. It is worth noting that EMOTET downloads submodules to perform specific tasks. These submodules can contain their own list of C2 servers. The extractor is also able to process these submodules.

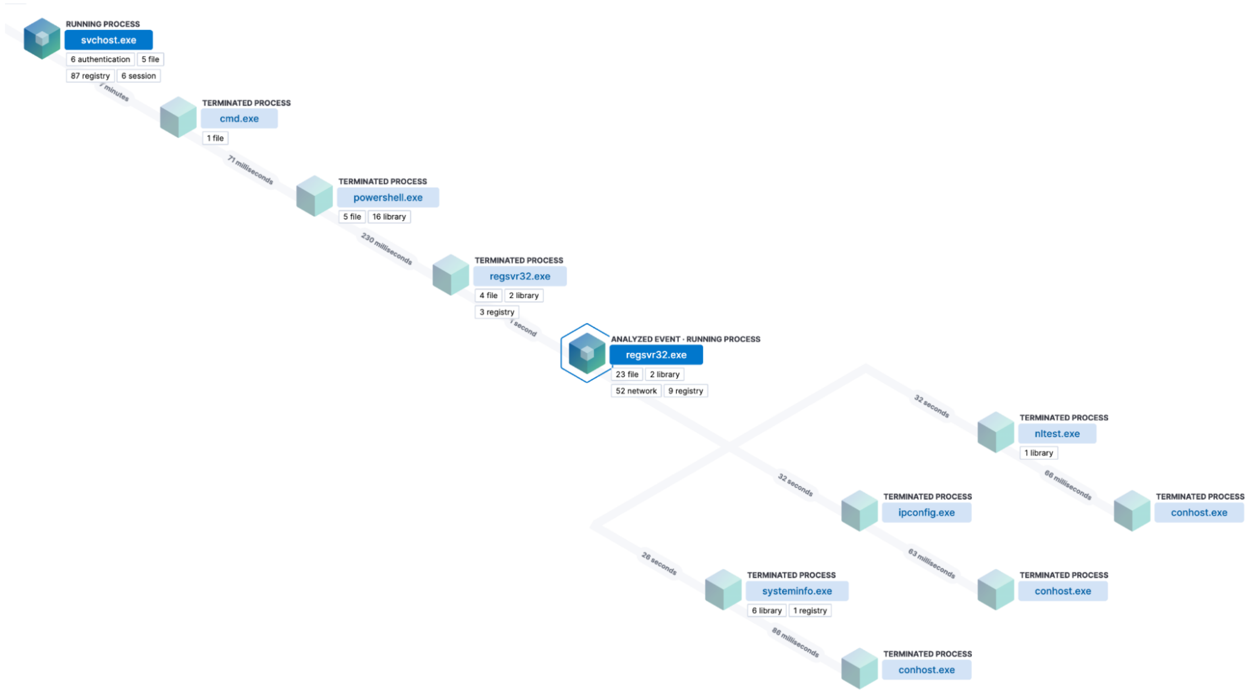
The submodules that we observed do not contain encryption keys. While processing submodules you can omit the **-k** flag.

```
[...]
[+] Key type: ECK1
[+] Key length: 32
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE2DWT12OLUMXfzeFp+bE2AJubVDsW
NqJdRC6yODDYRzYuuNL0i2rI2Ex6RUQaBvqPOL7a+wCWnIQszh42gCRQlg==
-----END PUBLIC KEY-----
[...]
[+] Key type: ECS1
[+] Key length: 32
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE9C8agzYaJ1GMJPLKqOyFrlJZUXVI
lAZwAnOq6JrEKHtWCQ+8CHuAIXqmKH6WRbnDw1wmdM/YvqKFH36nqC2VNA==
-----END PUBLIC KEY-----
[...]
[+] Found 64 c2 subs
174.138.33.49:7080
188.165.79.151:443
196.44.98.190:8080
[...]
[+] Starting emulation
[+] String BLOB address: 0x4000000
KeyDataBlob
[...]
[+] String BLOB address: 0x4000000
bcrypt.dll
[...]
[+] String BLOB address: 0x4000000
RNGRead more
```

To enable the community to further defend themselves against existing and new variants of EMOTET, we are making the payload extractor open source under the Apache 2 License. Access the payload extractor documentation and binary download.

The future of EMOTET

The EMOTET developers are implementing new techniques to hide their configurations from security researchers. These techniques will slow down initial analysis, however, EMOTET will eventually have to execute to achieve its purpose, and that means that we can collect information that we can use to uncover more about the campaign and infrastructure. Using code emulators, we can still find and extract the information from the binary without having to deal with any obfuscation techniques. EMOTET is a great example where multiple obfuscation techniques make static analysis harder. But of course, we expect more malware authors to follow the same example. That is why we expect to see more emulation-based configuration extract in the future.



EMOTET running and gathering system information

Detection

YARA

Elastic Security has created YARA rules to identify this activity. The YARA rules shown here are not meant to be used to solely detect EMOTET binaries, they are created to support the configuration extractor. The YARA rules for detecting EMOTET can be found in the protections-artifacts repository.

EMOTET key decryption function

```
rule resolve_keys
{
meta:
     author = "Elastic Security"
     description = "EMOTET - find the key decoding algorithm in the PE"
     creation_date = "2022-08-02"
     last_modified = "2022-08-11"
     os = "Windows"
     family = "EMOTET"
     threat_name = "Windows.Trojan.EMOTET"
     reference_sample =
"debad0131060d5dd9c4642bd6aed186c4a57b46b0f4c69f1af16b1ff9c0a77b1"
   strings:
       $chunk_1 = {
        45 33 C9
        4C 8B D0
        48 85 C0
        74 ??
        48 8D ?? ??
        4C 8B ??
        48 8B ??
        48 2B ??
        48 83 ?? ??
        48 C1 ?? ??
        48 3B ??
        49 0F 47 ??
        48 85 ??
        74 ??
        48 2B D8
        42 8B 04 03
     }
   condition:
       any of them
}
```
Read more

EMOTET C2 aggregation

```
rule c2_list
{
     author = "Elastic Security"
     description = "EMOTET - find the C2 collection in the PE"
     creation_date = "2022-08-02"
     last_modified = "2022-08-11"
     os = "Windows"
     family = "EMOTET"
     threat_name = "Windows.Trojan.EMOTET"
     reference_sample =
"debad0131060d5dd9c4642bd6aed186c4a57b46b0f4c69f1af16b1ff9c0a77b1"
  strings:
     $chunk_1 = {
        48 8D 05 ?? ?? ?? ??
        48 89 81 ?? ?? ?? ??
        48 8D 05 ?? ?? ?? ??
        48 89 81 ?? ?? ?? ??
        48 8D 05 ?? ?? ?? ??
        48 89 81 ?? ?? ?? ??
        48 8D 05 ?? ?? ?? ??
        48 89 81 ?? ?? ?? ??
        48 8D 05 ?? ?? ?? ??
        48 89 81 ?? ?? ?? ??
        48 8D 05 ?? ?? ?? ??
        48 89 81 ?? ?? ?? ??
        48 8D 05 ?? ?? ?? ??
        48 89 81 ?? ?? ?? ??
     }
  condition:
     any of them
}Read more
```

EMOTET string decoder

```
rule string_decode
{
   meta:
     author = "Elastic Security"
     description = "EMOTET - find the string decoding algorithm in the PE"
     creation_date = "2022-08-02"
     last_modified = "2022-08-11"
     os = "Windows"
     family = "EMOTET"
     threat_name = "Windows.Trojan.EMOTET"
     reference_sample =
"debad0131060d5dd9c4642bd6aed186c4a57b46b0f4c69f1af16b1ff9c0a77b1"
  strings:
     $chunk_1 = {
        8B 0B
        49 FF C3
        48 8D 5B ??
        33 CD
        0F B6 C1
        66 41 89 00
        0F B7 C1
        C1 E9 10
        66 C1 E8 08
        4D 8D 40 ??
        66 41 89 40 ??
        0F B6 C1
        66 C1 E9 08
        66 41 89 40 ??
        66 41 89 48 ??
        4D 3B D9
        72 ??
     }
     $chunk_2 = {
        8B 0B
        49 FF C3
        48 8D 5B ??
        33 CD
        0F B6 C1
        66 41 89 00
        0F B7 C1
        C1 E9 ??
        66 C1 E8 ??
        4D 8D 40 ??
        66 41 89 40 ??
        0F B6 C1
        66 C1 E9 ??
        66 41 89 40 ??
        66 41 89 48 ??
        4D 3B D9
        72 ??
     }
  condition:
     any of them
}
```
Read more