Emotet Config Redux

By Oleg Boyarchuk, Stefano Ortolani

Published: 2022-05-25 · Archived: 2026-04-05 18:00:28 UTC

It is no mystery that Emotet's development recently picked up. After its resurrection (some researchers pointing to TrickBot as the threat actor responsible), it bootstrapped two new botnets (Epoch 4 and Epoch 5), and it recently looked at replacing its own modules with native 64-bit implementations. Tracking its network infrastructure is however linked to the ability of decrypting and extracting its configuration file containing, besides the encryption keys that can be in turn used to identify the botnet, a list of compromised hosts that the payload would connect to upon execution. Unfortunately, either willingly or as a side-effect of other changes (or rather, some obfuscation improvements), since mid-May Emotet samples have started to transition to a new method of storing the configuration data within the binary: not anymore as a single blob of data, but as a split collection of fragments, each one obfuscated separately.

In this blogpost we briefly detail this change and show how to keep extracting the list of network indicators despite this new obfuscation technique.

Obfuscation Improvements

To provide actionable threat intelligence, many vendors of dynamic analysis systems often decide to add routines to automatically extract C2 configuration data of known malware families, including Emotet. For a very long time Emotet kept its encrypted C2 config in the beginning of .data section of the PE module. As we noticed in our previous blog post, this did not change when Emotet decided to migrate to pure 64-bit binaries: each and every sample had a function which was responsible for config decryption (see Figure 1).

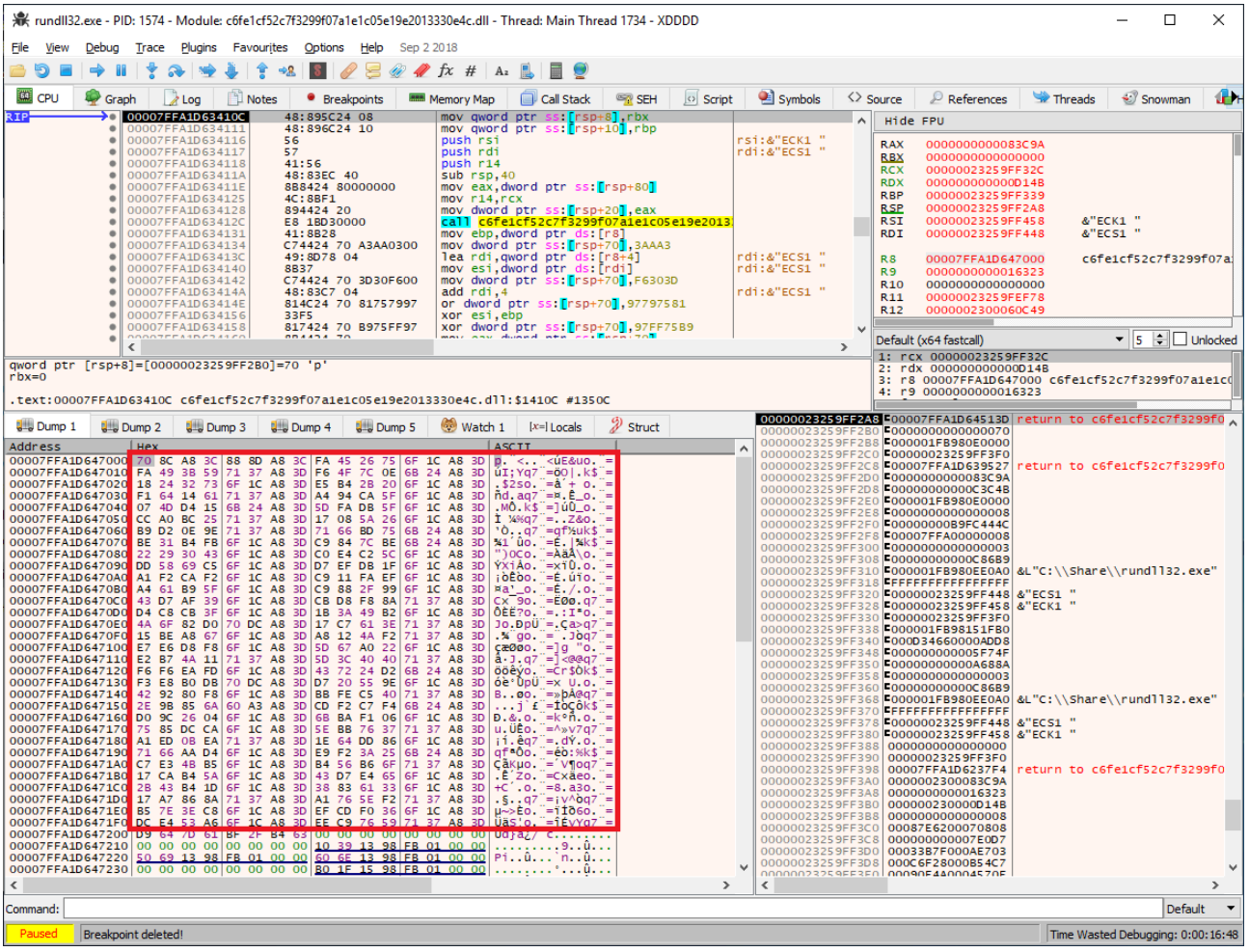

Figure 1: Encrypted C2 config passed as a parameter to the decryption routine (c6fe1cf52c7f3299f07a1e1c05e19e2013330e4c).

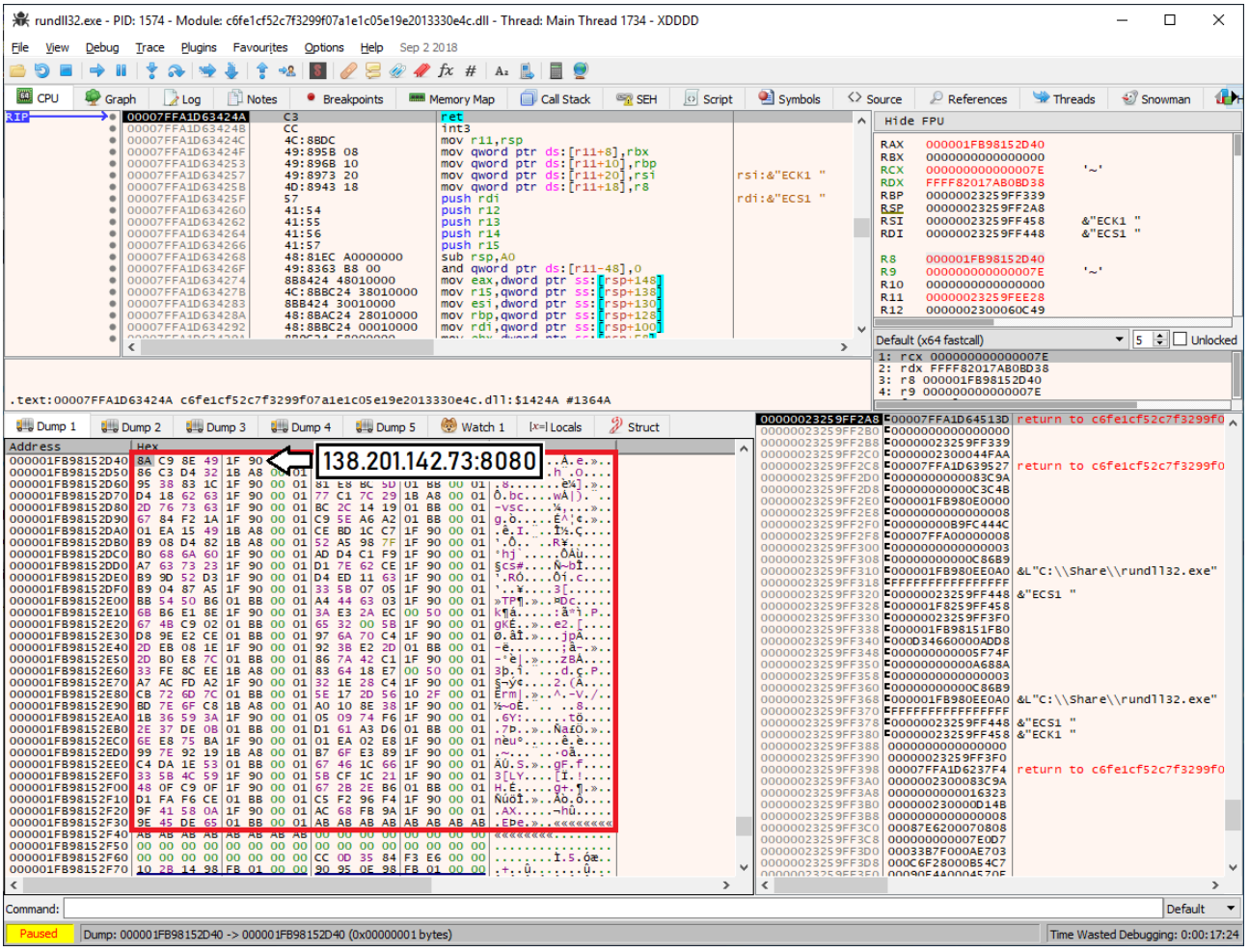

Figure 2: Decrypted C2 config is an array of IP:port pairs (c6fe1cf52c7f3299f07a1e1c05e19e2013330e4c).

Unfortunately, that is not the case anymore: in a recent wave of Emotet samples, the configuration data is not embedded in the binary in one single blob; rather, each new sample now features an accumulator function (see Figure 3) which returns a pointer to an array of function pointers, each returning a single C2 IP address and port (see Figure 4).

```
__int64 get_C2_config()
{
  _QWORD *v0; // rcx
  int v1; // eax
  unsigned int v2; // ebx

  v0 = (_QWORD *)qword_7FFA1B6BD060;
  v1 = 974161;
  v2 = 0;
  while ( v1 != 916372 )
  {
    qword_7FFA1B6BD060 = alloc_memory(0x250u);
    if ( !qword_7FFA1B6BD060 )
      return v2;
    v0 = (_QWORD *)qword_7FFA1B6BD060;
    *(_DWORD *)(qword_7FFA1B6BD060 + 584) = 0;
    v1 = 916372;
  }
  v0[29] = sub_7FFA1B6AEAA4;
  v0[37] = sub_7FFA1B6BA048;
  v0[50] = sub_7FFA1B699D98;
  v0[44] = sub_7FFA1B6B6F30;
  v0[14] = sub_7FFA1B69D6D4;
  v0[48] = sub_7FFA1B6973E8;
  v0[41] = sub_7FFA1B6926DC;
  v0[13] = sub_7FFA1B6B0598;
  v0[57] = sub_7FFA1B6ADEFC;
  v0[27] = sub_7FFA1B691294;
  v0[31] = sub_7FFA1B699598;
  v0[64] = sub_7FFA1B6A62BC;
  v0[53] = sub_7FFA1B6B7FF0;
  v0[40] = sub_7FFA1B6B6550;
  v0[7] = sub_7FFA1B694D64;
  v0[60] = sub_7FFA1B694A1C;
  v0[66] = sub_7FFA1B695B88;
  v0[18] = sub_7FFA1B69E7E4;
  v0[63] = sub_7FFA1B6923DC;
  v0[65] = sub_7FFA1B6B8250;
  v0[39] = sub_7FFA1B6ADFF0;
  v0[59] = sub_7FFA1B699320;
  v0[21] = sub_7FFA1B693DCC;
  v0[42] = sub_7FFA1B6A7550;
```

Figure 3: C2 accumulator function from the new wave (b409ca9851fecca61e6cb0aaaa56fdaafc7242f5).

This means that it is now impossible to statically retrieve a blob of data from a sample, decrypt it, and extract a list of IP addresses, as the C2 configuration is now spread across the code. Another challenge is that IP address and port cannot be extracted from a static view of the disassembled code because of some newly added obfuscation, as shown in Figure 4: the code now performs a series of mathematical operations on a set of hardcoded values before leading to what are the actual values representing IP address and port.

```
sub_7FFA1B6AEAA4 proc near               ; DATA XREF: get_C2_config:loc_7FFA1B6B10CB↓o
                                         ; .pdata:00007FFA1B6BEAE0↓o

var_18          = dword ptr -18h
var_10          = dword ptr -10h
var_C           = dword ptr -0Ch
arg_0           = dword ptr  8
arg_8           = dword ptr  10h
arg_10          = dword ptr  18h
arg_18          = dword ptr  20h

                sub     rsp, 18h
                mov     [rsp+18h+var_10], 3FB4Eh
                xor     eax, eax
                mov     r8, rcx
                mov     [rsp+18h+var_C], eax
                mov     [rsp+18h+arg_0], 6EACB3h
                mov     r9, rdx
                shr     [rsp+18h+arg_0], 6
                xor     [rsp+18h+arg_0], 0FE354h
                mov     eax, [rsp+18h+arg_0]
                mov     [rsp+18h+arg_0], eax
                mov     [rsp+18h+arg_10], 267DBC3Ah
                mov     [rsp+18h+var_18], 30CA5D18h
                mov     [rsp+18h+arg_8], 451FA4EEh
                mov     [rsp+18h+arg_18], 2F5A5D19h
                mov     [rsp+18h+arg_0], 0E88EB0h
                xor     [rsp+18h+arg_0], 6B888E2h
                or      [rsp+18h+arg_0], 0DA36A92Bh
                add     [rsp+18h+arg_0], 3444h
                xor     [rsp+18h+arg_0], 0DE78BA59h
                mov     eax, [rsp+18h+arg_0]
                mov     [rsp+18h+arg_0], eax
                mov     ecx, [rsp+18h+arg_8]
                mov     eax, [rsp+18h+arg_10]
                xor     ecx, eax
                mov     eax, 0A0A0A0A1h
                mov     [r8], ecx
                mov     [rsp+18h+arg_0], 702B01h
                xor     [rsp+18h+arg_0], 0EEC02504h
                mov     ecx, [rsp+18h+arg_0]
                mul     ecx
```

Figure 4: Obfuscated C2 function from the new wave which returns 212.24.98.99:8080 (b409ca9851fecca61e6cb0aaaa56fdaafc7242f5).

How to Defeat Obfuscation

A straightforward approach to deal with this kind of obfuscations is to use code decompilers (for example Hex-Rays). An often-underestimated advantage of decompilers is the ability to also reduce code complexity as a by-product of lifting the binary code to higher level languages; Figure 5 shows an example of a decompiled and deobfuscated code fragment. While ideal for manual analysis, decompilers are expensive and are not guaranteed to work in the general case (deobfuscation tends to be hit-or-miss).

```
__int64 __fastcall sub_7FFA1B6AEAA4(_DWORD *a1, _DWORD *a2)
{
  *a1 = 0x636218D4;                             // 0xD4.0x18.0x62.0x63 = IP 212.24.98.99
  *a2 = 0x1F900001;                             // 0x1F90 = port 8080
  return 0xE59E6i64;
}
```

Figure 5: C2 function from the new wave deobfuscated by Hex-Rays (b409ca9851fecca61e6cb0aaaa56fdaafc7242f5).

Running the code in a code emulator such as QEMU or Qiling (both free for commercial use) is often a more reliable way to extract the required data because they can emulate both the CPU and the underlying OS environment. In this scenario, the starting point needs to be the inner DLL extracted as shown in our blog post. Once that is done, static analysis can be used to identify the accumulator function, and thereby obtain the full list of functions used to decode the C2 data. The last step is computing the physical offset within the module and feeding it to the emulator as a starting instruction. Figure 6 contains a quick implementation to decode the C2 data from the function shown in Figure 5 (i.e., sub_7FFA1B6AEAA4); in this case the physical offset was 0x1DEA4.

```
In [1]: import qiling
   ...: import pefile
   ...: import struct
   ...:
   ...: with open('b409ca9851fecca61e6cb0aaaa56fdaafc7242f5_dumped.dll', 'rb') as f:
   ...:     file_data = f.read()
   ...: ql = qiling.Qiling(code=file_data[0x1DEA4:], archtype='x8664', ostype='windows', verbose=qiling.const.QL_VERBOSE.DISABLED)
   ...:
   ...: ql.stack_push(0)
   ...: ql.reg.rcx = ql.reg.rsp
   ...: ip_addr = ql.reg.rsp
   ...:
   ...: ql.stack_push(0)
   ...: ql.reg.rdx = ql.reg.rsp
   ...: port_addr = ql.reg.rsp
   ...:
   ...: for i in range(1,5):
   ...:     ql.stack_push(0)
   ...:
   ...: def detect_ret(ql, addr, size):
   ...:     if size == 1 and ql.mem.read(addr, 1) == b'\xC3':
   ...:         ql.emu_stop()
   ...:
   ...: ql.hook_code(detect_ret)
   ...: ql.run()
   ...:
   ...: ip, = struct.unpack_from("<L", ql.mem.read(ip_addr, 4))
   ...: port, = struct.unpack_from("<L", ql.mem.read(port_addr, 4))
   ...: port >>= 16
   ...:
   ...: print("{}.{}.{}.{}:{}".format(ip & 0xFF, (ip >> 8) & 0xFF, (ip >> 16) & 0xFF, (ip >> 24) & 0xFF, port))
212.24.98.99:8080

In [2]:
```

Figure 6: Small program to decode a single network indicator given a physical offset.

Conclusions

While a reader might start to conclude (as we predicted) that Emotet has finally resumed playing cat and mouse games with security researchers, we still believe these improvements to be part of a more comprehensive refactoring, and that payloads will likely keep evolving following internal roadmaps yet to become clear to the public. Unfortunately, all these changes are the only signal that security defenders can use to improve detectors and keep users safe from this ever-evolving threat.
