Cracking Formbook malware: Blind deobfuscation and quick response techniques

tehtris.com/en/blog/cracking-formbook-malware-blind-deobfuscation-and-quick-response-techniques/

November 13, 2024

Threat analysts have to respond quickly to an attack. When dealing with well-obfuscated code, shortcuts sometimes have to be taken to expedite the analysis and detection process. A malware identified by our TEHTRIS Threat Intelligence team (CTI) as Formbook stealer was writing some scrambled files on the victim’s side which will illustrate how analyst could perform a known plaintext attack on an obfuscated file.

First, what is formbook ? FormBook is a widespread information-stealing malware known for targeting Windows systems. It primarily focuses on stealing login credentials, keystrokes, and other sensitive data, often delivered through phishing emails or malicious attachments. FormBook is easy to acquire as malware-as-a-service (MaaS) on underground forums, making it a popular choice for cybercriminals.

When running the FormBook malware (SHA-256: e5aebbb2c6ad445d5a2ee5c33a77d8ecfe74cf206433b723736e2e88b in our sandboxes, one file dropped by the malware (T1588.001) immediately catches our attention:

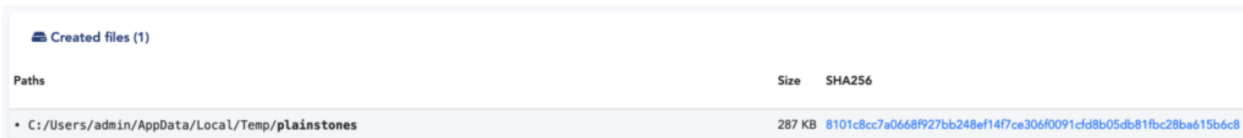

fig.1: Dropped file as seen in the sandbox

This file was heavily obfuscated and completely invisible in the source file. Upon inspecting it, we observed notably low entropy, which was an immediate indicator of something not encrypted. Additionally, a familiar pattern emerged, resembling structures commonly seen in a PE (Portable Executable) file. We suspect that the file is obfuscated using a static key, without the use of any diffusion mechanisms, making it easier to detect certain patterns despite the obfuscation.

The patterns, resembling the MZ header, PE header, and padding, appear to align with typical structures, as shown in the figure below.

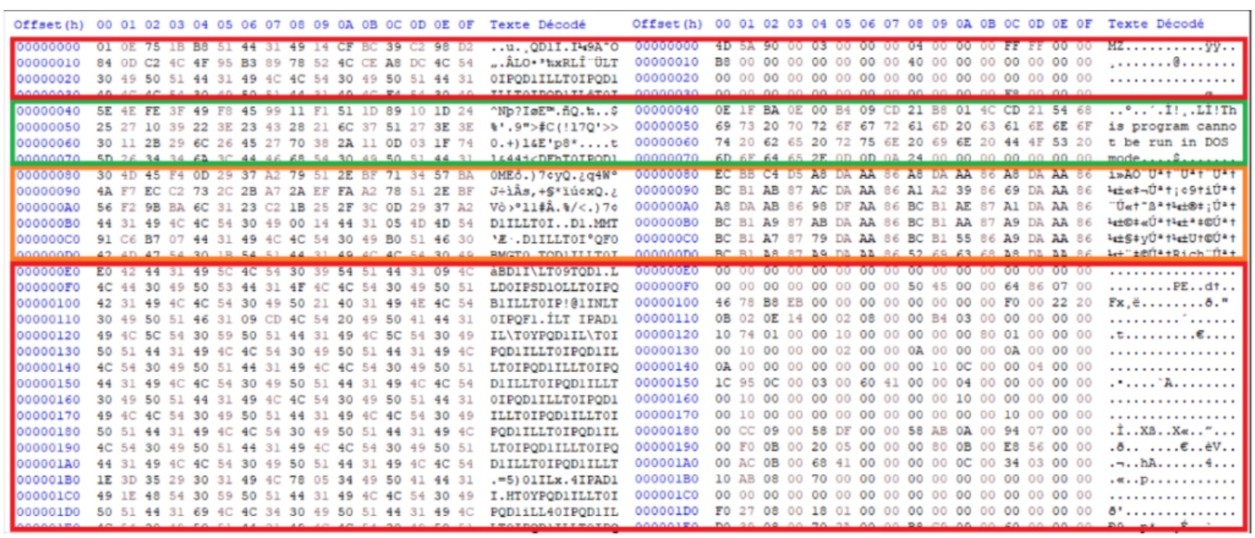

fig.2: Comparison with “kernel32.dll”

We can now attempt to guess the key. By XORing the first two bytes of the obfuscated file with the known plaintext (signature of the “MZ” header), we can retrieve the first two bytes of the key. If the input buffer is zero, the key will appear in plain text. By searching for this pattern in large areas of zero bytes within the input, we can determine both the key content and its length (orange).

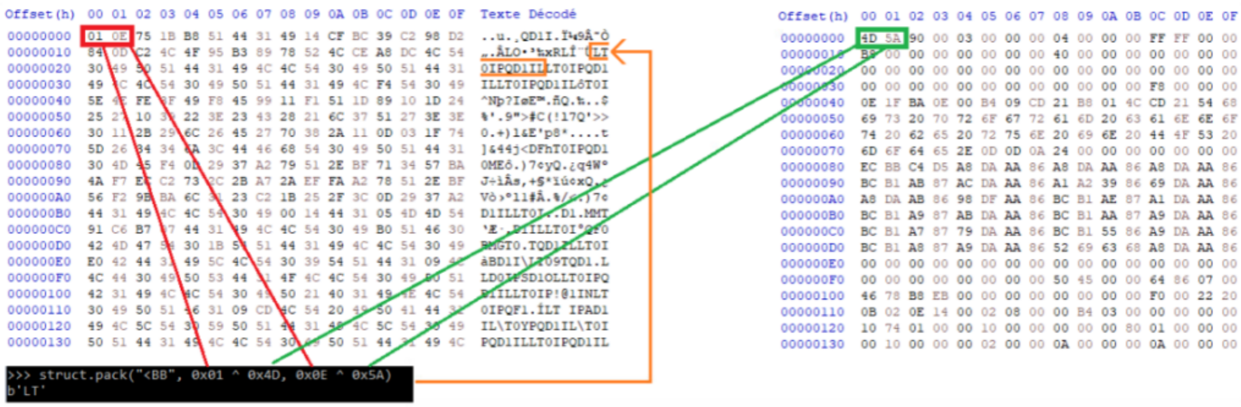

fig.3: Key guessing

A simple Python script can be used to deobfuscate this type of obfuscated file by applying the retrieved key to reverse the XOR encryption.

```
import sys
import pathlib
import array


def xor_decrypt(encrypted_file: pathlib.Path,
                                decrypted_file: pathlib.Path,
                                key: bytes) -> None:

        key = array.array("B", key)
        with encrypted_file.open("rb") as enc_fd, decrypted_file.open("wb") as
dec_fd:
                while chunk := enc_fd.read(len(key)):
                        chunk = array.array("B", chunk)
                        for i in  range(len(chunk)):
                                chunk[i] ^= key[i]
                        dec_fd.write(chunk.tobytes())


if __name__ == "__main__":
    infile = pathlib.Path(sys.argv[1])
    outfile = pathlib.Path(sys.argv[1] + ".exe")
    xor_decrypt(infile, outfile, b"LT0IPQD1IL")
    print(outfile)
```

The deobfuscated dropped file was generated using Hasherezade. As a result, this file is already detectable by our sandbox.

[8 / 10] Yara rule detection triggered while analyzing binaries

- **Hit**: Binary triggered the Yara rule : Executable has trailing data
- **Hit**: Binary triggered the Yara rule : Inconsistant ASLR in PE header. Bad compilation
- **Hit**: Binary triggered the Yara rule : Hasherezade PE to shellcode (Peshc)

fig.4: Sandbox detection

To detect this particular type of obfuscated payload, we will apply basic mathematical checks to identify if the known plaintext pattern can be found. First, we will attempt to guess the first two bytes of the key and estimate a multiple of the key length, which should be determined quickly due to the abundance of null bytes in the PE header. Finally, we will compare two offsets that should contain null bytes in the original file (0x3FD and 0x3A0), aligned with the multiple of the guessed key length, with the first two bytes of the clear text key to confirm the key's validity.

The following YARA rule implements these calculations:

```
rule formbook_obfuscated_payload {
    meta:
        author = "PEZIER Pierre-Henri. Copyright TEHTRIS 2024"
    condition:
        filesize > 100KB and
        int8(0) != 0x4D and                                                      // Regular
PE is encrypted with null key. Blacklist it!
        for any i in (2..0x30) : (                                               // Look at
the PE header only
            int8(0) ^ 0x4D == int8(i) and int8(1) ^ 0x5A == int8(i + 1)   // Guess it
matches the next 2 bytes
            and (                                                                // i is a
modulo of the key length
                    (                                                            // Zero
byte known plaintext attack on 2 bytes
                        int8(0x3FD - 0x3FD % i) == int8(0) ^ 0x4D
                        and int8(0x3FD - 0x3FD % i + 1) == int8(1) ^ 0x5A
                    )
                    and (                                                        // Test
another offset
                        int8(0x3A0 - 0x3A0 % i) == int8(0) ^ 0x4D
                        and int8(0x3A0 - 0x3A0 % i + 1) == int8(1) ^ 0x5A
                    )
            )
        )
}
```

The analysis employs various techniques to expedite the process, allowing us to be more responsive to our customers. This quick “live my life” offers a brief preview of our work, which is often overlooked but crucial for maintaining security.

IOCs (indicators of compromise)

The following SHA-256 hashes correspond to similar samples:

- 08ceff2aa4f92b26b10f1accbc1d41cb2729e3beb2f558ce0fe060f77243a86c
- 0e9622e96afd3166996349ccd288105bb5c2c9c287c979ab2f0baa3b0b461929
- 10882b3477b6a32049e6f67e67885927ddcc28750884e0b02df5f228bc10f905
- 44162eee61f7d49a55fe0f815d0bc996cd728d96307b5bc6277fe430941ad068
- 47155987c94e0b921887ed3aa2278fb857781238c518fbea52224728b88b0436
- 492cf9ce5a8baf6b424bf890106ba96ae824bc8ba93c4dd2da25cbf37a685c90
- 59c25af850a539e0863d6018774ac029419d1581ca8a034b1c4ce9239bd8084b
- 5e74f08923fec3a5daf99b9a6c0763b21a98226f90c537235408a4258389ca01
- 6cc54bd57057a1fc07c2726c351a42f47caef4ae05a2693fbf6b9f693c6761c6
- 7616904db54d77cb25cc58f279bfdf6ef5cbabe19573cbd781238be01daaa1c4
- 998328ecd3a13fd3287f88e37119064b3a4094d2e935786a5327d47e4ed4466b
- b3c5c896606eb408bd97f255b916cda8cf8aa4291c3f68c5108c9ff0f5b7c0b7
- b9906d121c2b4a44b38c657e3f051be5dd55fca2d8f3e51150cafad9afd77d03
- c16321285091a58a2a0e63e4d445a71d6b9a60f27a6741c0a590a4bc5290d368
- d9c7fa0a03560078e3eb0a61d42cafd68ee7c90da0ca06ca83bc05bab596f7c6
- e5aebbb2c6ad445d5a2ee5c33a77d8ecfe74cf206433b723736e2e88b9b5d78f
- e63b97535e194d90756cc01a322550d4fa41a76117799a798ea0a78c6dd940bd
- e88d0ce2a1ef65103221e16ad5a3d31c74799fa7b75dc6ccea1c2a7c8ba5a857
- f3f0ac7ba7b93d8571adfa54987fb7374451f863b44946202bc623a528fc5b5f