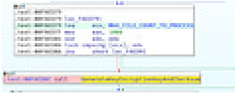
Dissecting LockBit v3 ransomware

blog.calif.io/p/dissecting-lockbit-v3-ransomware

Nhân Huỳnh, Hoang Nguyen, Thai Duong

Share this post



Dissecting LockBit v3 ransomware

blog.calif.io

Introduction

In our last article, we recommended analyzing ransomware binaries as part of an effective ransomware response strategy:

> “Analyzing binaries is hard. Analyzing obfuscated ransomware is even harder.
>
> [...] However, it is worth investing in analyzing and understanding ransomware. Crypto breaking bugs may be rare, but they are not impossible to find. In addition, ransomware authors may not fully understand how to use crypto correctly. The only way to determine if it is possible to recover the data, if any, is the long and detailed ransomware analysis by an expert team.
>
> [...] In addition, a successful analysis can help reassure you that there are no potential bugs in the encryption and decryption process. It also helps the technical team understand and potentially improve the recovery process. This is an investment that should be considered early on in an incident.”

In this article, we show some examples of crucial intelligence you can gain from a meticulous and accurate ransomware analysis. The target of this analysis is a variant of the **LockBit v3** ransomware that we encountered in a recent engagement. This variant is also known as **LockBit Black** due to some code similarity with the **BlackMatter** family. These samples are built from the leaked LockBit v3 builder available on GitHub.

Calif discovered two issues in this version of the ransomware:

- a crypto bug that may allow for the decryption of a portion of the data without the private key, i.e., without paying the ransom.
- a design flaw that may cause data corruption and permanent data loss.

We decided to publish this analysis for the following reasons:

- The crypto bug is already known to the malware author. We have observed newer variants where we can no longer take advantage of this bug.
- We want to share our analysis and research to help other affected organizations prepare and respond to the same ransomware family, especially regarding the data corruption flaw.
- The LockBit v3 family contains interesting anti-analysis techniques and clever use of standard cryptographic algorithms that are not well documented. These technical details would be valuable for malware researchers and threat hunters.

We also publish an open-source decryptor for this variant. You can download the tool from GitHub.

Calif would like to extend a special thank you to Chuong Dong – a malware expert who has previous experience with this ransomware family. During the initial analysis, we requested Chuong's assistance to swiftly comprehend the file encryption scheme. His help proved highly valuable as we managed to quickly reimplement the decryptor.

Note that the screenshots and code snippets within this article assume that the encryptor is loaded at address 0xFA0000 instead of the default ImageBase of 0x400000. The decryptor is loaded at the ImageBase of 0x400000. In addition, the ransomware has many anti-debugging and obfuscation mechanisms. To bypass these protections and reproduce this analysis, please refer to Appendix A: Reverse engineering detail.

Table of contents

Introduction

Encryption and decryption logic

- Encrypted file structure
- Footer structure
- Modified Salsa20
- RSA with no padding
- File encryption
- File decryption

Flaws

- Keystream reuse vulnerability

- Data corruption

Conclusion

Appendix A: Reverse engineering detail

Appendix B: Open-source decryption tool

Appendix C: Binary Information and Indicators of Compromise (IOCs)

Appendix D: IDC script to rename functions

Appendix E: Chunk counts and skip bytes

Encryption and decryption logic

The sample encrypts files using a combination of symmetric and asymmetric cryptography, as follows:

- Generate a 64-byte random key for each targeted file. We will refer to it as the **file_encryption_key**. We identify the encryption algorithm as a variant of **Salsa20**. Normally, Salsa20 uses a 32-byte key, but this variant uses 64-byte. Please refer to the Modified Salsa20section for more details. Unless specified otherwise, all references to Salsa20 in this document refer to this modified version.

- Generate another 64-byte random Salsa20 key to encrypt the file_encryption_key. We will refer to this second key as the **key_encryption_key**. As an optimization to reduce the number of slow RSA encryption operations, the sample reuses this key for 1,000 files before generating a new one. This key reuse leads to a vulnerability described in the Keystream reuse vulnerability section.

- Encrypt the key_encryption_keys using **RSA with no padding**, using a 1024-bit public key embedded within. We describe this algorithm in the RSA with no padding section. Note that since 2015 NIST has recommended against using 1024-bit RSA keys.

Encrypted file structure

The sample processes targeted files the same way during encryption and decryption. It divides each file into chunks of 0x20000 bytes. The sample does not pad the file if the file size or the size of the last chunk is less than 0x20000 bytes.

Consecutive chunks form a group. There are three group types: **before**, **skip**, and **after group**. There is exactly one “before group” at the beginning of the file. The skip group and the after group follow the before group and repeat alternatively throughout the rest of the file.

The sample encrypts chunks of the before group and after groups using Salsa20. It leaves chunks in the skip group unencrypted. It determines the number of chunks in each group based on the file size. Please refer to Appendix E for more details.

An encrypted file ends with a **footer** containing information about the file such as the file’s original name, number of chunks in each group, etc, including the file_encryption_key to decrypt the file data. The sample encrypts this footer, and appends it to the file after the encryption finishes. For a detailed description of the footer structure, refer to the next section.

The overall structure of an encrypted file can be visualized as follows:

Footer structure

We reconstruct the overall structure of the footerin the C snippet below:

```
struct file_encryption_info
{
  char filename[file_encryption_info.filename_size]; // apLib compressed
  uint16_t filename_size;
  LARGE_INTEGER skipped_bytes;
  int before_chunk_count;
  int after_chunk_count;
  uint8_t file_encryption_key[0x40];
};

struct key_encryption_info
{
  uint16_t file_encryption_info_length;  // necessary because filename is dynamically
sized
  int checksum;
  union
 {
    struct
    {
        uint8_t key_encryption_key[0x40];
        uint8_t checksum[0x40];
    } decrypted;
    uint8_t encrypted_key_encryption_key[0x80];  // RSA encrypted
 } key_blob;
};

struct footer
{
  struct file_encryption_info file_encryption_info;  // Salsa20 encrypted
  struct key_encryption_info key_encryption_info;
};
```

The **file_encryption_info** contains a randomly generated key to decrypt the file content. The file_encryption_info is encrypted using Salsa20. The key to decrypt the file_encryption_info is stored in the encrypted_key_encryption_key field of the **key_encryption_info** structure. This field, in turn, is encrypted using the RSA public key embedded in the ransomware.

The decryptor contains an embedded private key, and works as follows:

1. Read the key_encryption_info structure at offset 0x86 bytes from the end of the file.
2. Hash the encrypted_key_encryption_key field and verify it against the checksum field as seen here.
3. Decrypt the encrypted_key_encryption_key using the embedded private RSA key then validate the key_encryption_key with the decrypted.checksum field.
4. Calculate the start of the file_encryption_info structure using the file_encryption_info_length field.

5. Use the key_encryption_key to decrypt the file_encryption_info structure using the modified Salsa20 algorithm. This structure contains the Salsa20 file_encryption_key that can be used to decrypt the chunks in the before group and after group.

Modified Salsa20

The sample encrypts the file_encryption_infostructure and the chunks using Salsa20 at address 0x00FA20AC.

Salsa20 has a 64-byte state that is used to generate a key stream to encrypt the plaintext one 64-byte block at a time. In the vanilla Salsa20 standard, the initial 64-byte state consists of a 32-byte key, an 8-byte block counter, an 8-byte nonce, and a 16-byte constant that spell "expand 32-byte k" in ASCII.

However, in this variant, the entire initial state is filled with random values. The aforementioned file_encryption_key and key_encryption_key are the initial states of the file encryption and key encryption processes respectively.

This finding shows that LockBit v3 is indeed a successor of BlackMatter, which in turn came from the Darkside ransomware family. Chuong's analysis of Darkside shows that it also fills the Salsa20's initial state, which Chuong called the matrix, with random values.

RSA with no padding

This sample encrypts key_encryption_info.key_encryption_key and key_encryption_info.checksum, using a custom implementation of the RSA algorithm at address 0x00FA17B4.

Recall that an RSA public key consists of two components:

- The modulus N.
- The public exponent e.

To encrypt a message m using RSA with no padding, you compute m^e (mod N). This encryption mode, which is known as textbook RSA, has many potential footguns. For example, it's possible to recover small messages. Therefore, m is usually padded with PKCS v1.5 or OAEP padding schemes.

However, the sample uses no padding. We can't find any obvious issues, because the sample only encrypts messages that have the same size as the modulus. In particular, it uses a 1024-bit key to encrypt key_encryption_info.key_encryption_key and key_encryption_info.checksum, which in total are also 1024 bits long.

File encryption

Before encrypting any files, the sample parses its embedded configuration at address 0x00FC600C. This data are encrypted by the function at 0x00FA6F48 and contain information such as configuration flags, file hashes to avoid, ransom note, and the RSA public key used to encrypt the randomly generated key_encryption_info.key_encryption_key.

After decrypting its configurations, the sample parses its command line arguments and enumerates target paths to encrypt files. The sample operates slightly differently depending on the command line argument. However, the file encryption logic is similar across different execution flows. The sample creates one thread for traversing and queueing files to be encrypted and multiple threads to actually encrypt the files. The threads communicate asynchronously with each other using an IO completion port.

At a high-level, the encryption threads work as follows:

The file traversal and queueing logic starts at 0x00FAF308.

- It drops a ransom note in the current directory.
- For each file in the current target directory, it verifies the filename against the lists of hashes to avoid. If the current filename doesn't belong to any of the lists, it renames the current file and adds a unique extension. In our variant, the extension is **.IzYqBW5pa**.
- It increases various counters, including a counter for the number of files using the current key_encryption_key. This key is randomly generated and reused once every 1,000 files. Once this counter reaches 1,000, the sample resets it back to 0 and generates a new key_encryption_key. This design introduces a bug that allows for the decryption without paying the ransom. This bug is described in detail in the Key stream reuse vulnerability section.
- It fills out and sets up the key_encryption_info structure for the current file. The logic to set the before_chunk_count, skipped_bytes, and after_chunk_count is at address 0x00FAE8AC. These values are determined based on the current file size. Refer to Appendix E for the exact values of each field based on the current file size.

The file encryption thread logic starts at address 0x00FADE78. This function simply determines if it needs to encrypt the current chunk depending on the file_encryption_info structure. It uses a randomly generated key stored at file_encryption_info.file_encryption_key to encrypt each chunk using Salsa20. Finally, when the entire file is processed, it writes the footer structure to the end of the file.

File decryption

The decryptor binary LB3Decryptor.exe is not obfuscated and can be quickly analyzed statically.

Similar to the encryptor, the decryptor parses its command line arguments and enumerates paths to decrypt files. The sample also creates multiple threads for decrypting and one for traversing and queueing files. These threads communicate asynchronously with each other using an IO completion port.

At a high-level, the decryption threads work as follows:

- The file traversal and queueing logic starts at 0x00403CEC. For each file, it decrypts the key_encryption_info (see Footer structure) at address 0x00403960. Then, it obtains the file_encryption_key and the chunk counts before queueing the file.
- The file decryption thread logic starts at 0x004030DC. It decrypts the chunks selected by the grouping algorithm using the file_encryption_key. Finally, when the entire file is processed, it removes the encrypted footer structure at the end of the file.

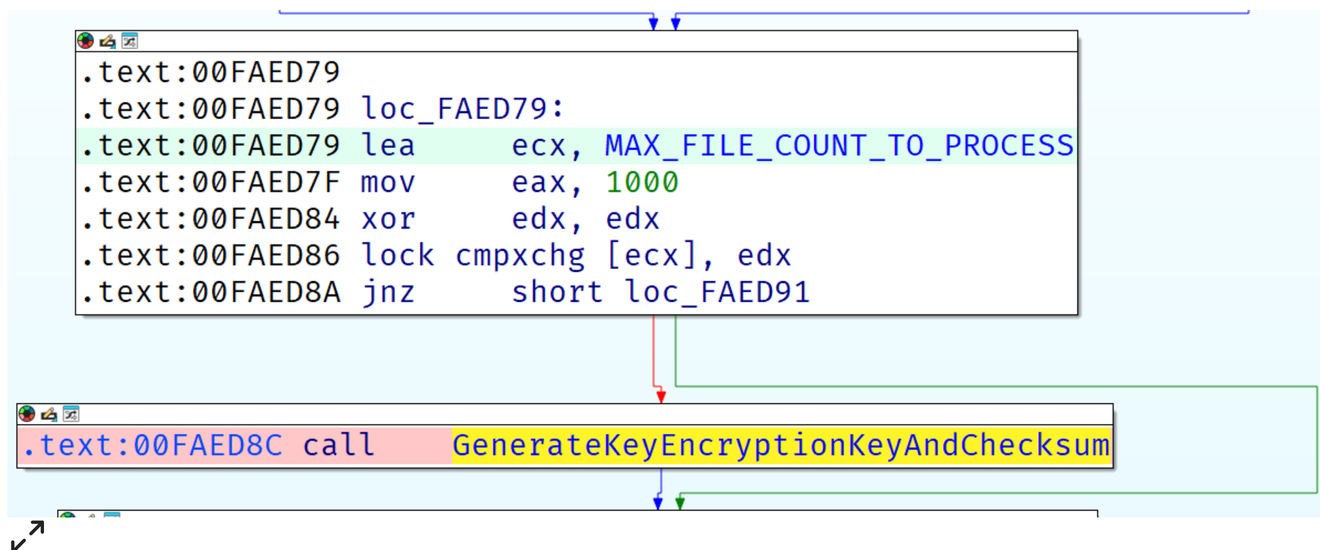
Flaws

Keystream reuse vulnerability

This version of the LockBit v3 ransomware has a keystream reuse vulnerability.

Instead of directly encrypting the file_encryption_info structure with RSA, the sample aims to reduce the number of slow RSA operations by adding another layer of Salsa20 encryption. This is where it makes a mistake that may allow the recovery of a portion of the data.

The sample generates a random Salsa20 key_encryption_key to encrypt the file_encryption_info structure once every 1,000 files as seen below:



Therefore, the Salsa20 algorithm would generate the same key stream for 1,000 files from the same key. Within these 1,000 files, if there is a file with a sufficiently long compressed filename, we can recover enough of the keystream to decrypt the file_encryption_info structure of other files with a much shorter compressed filename. This file_encryption_info structure contains the file_encryption_key to decrypt the file content. In other words, if we happen to have a file with a sufficiently long compressed filename, chances are we can recover the content of other files with shorter compressed filenames without the private key from the threat actor, i.e., without paying the ransom.

For example, we created two short text files for our test case:

- a.txt, whose compressed filename is: 61 e0 2e e0 74 e0 78 db 09 02 00 00
- aABCDEFGHIJKLMNOPQRSTUVWXYZ[]^_`abcdefghijklmnopqrstuvwxyz{}~123456789.txt, whose compressed filename is shown below:

```
00000000: 61 e0 41 e0 42 e0 43 e0 44 e0 45 e0 46 e0 47 e0  a.A.B.C.D.E.F.G.
00000010: 48 e0 49 e0 4a e0 4b e0 4c e0 4d e0 4e e0 4f e0  H.I.J.K.L.M.N.O.
00000020: 50 e0 51 e0 52 e0 53 e0 54 e0 55 e0 56 e0 57 e0  P.Q.R.S.T.U.V.W.
00000030: 58 e0 59 e0 5a e0 5b e0 5d e0 5e e0 5f e0 60 e1  X.Y.Z.[.].^._.`.
00000040: 9c 80 62 1c 63 1c 64 1c 65 1c 66 1c 67 1c 68 1c  ..b.c.d.e.f.g.h.
00000050: 69 1c 6a 1c 6b 1c 6c 1c 6d 1c 6e 1c 6f 1c 70 1c  i.j.k.l.m.n.o.p.
00000060: 71 1c 72 1c 73 1c 74 1c 75 1c 76 1c 77 1c 78 1c  q.r.s.t.u.v.w.x.
00000070: 79 1c 7a 1c 7b 1c 7d 1c 7e 1c 31 1c 32 1c 33 1c  y.z.{.}.~.1.2.3.
00000080: 34 1c 35 1c 36 1c 37 1c 38 1c 39 19 2e 51 78 b6  4.5.6.7.8.9..Qx.
00000090: 09 02 00                                          ...
```

The content of the encrypted file with the longer file name is shown here:

```
00000000: 6e c3 86 8b 6d c6 d8 ca d9 e9 52 94 c6 1f 60 13  n...m.....R...`.
00000010: 9d a3 bc d0 f4 1c ce 23 8d a7 bc 13 22 e6 14 68  .......#....."..h
00000020: dd 0c f5 ec 12 b2 d8 91 73 8e 7c d9 e3 29 57 75  ........s.|..)Wu
00000030: 6a 10 7d ba 74 e6 db 81 fe 8a 33 14 30 3b d6 dc  j.}.t.....3.0;..
00000040: 55 c6 e2 08 31 fa 7f 9f 21 0e 1e 76 b1 f7 a5 d1  U...1...!..v....
00000050: 4b 58 a7 89 3e d2 e2 55 0e b7 75 1f d7 04 1e a5  KX..>..U..u.....
00000060: 68 49 e7 9d 99 b7 d8 01 32 68 bc 60 d9 39 ee e5  hI......2h.`.9..
00000070: 2e 75 4d e9 c0 36 8f 1a b0 a4 03 72 ef d0 33 af  .uM..6.....r..3.
00000080: df b9 2c 57 ca ff 2c e6 44 33 9e ef 27 7b 26 ee  ..,W..,.D3..'{&.
00000090: fa 6f 56 bd ec 81 31 63 41 fb 35 b1 7e 0d 85 8a  .oV...1cA.5.~...
000000a0: 07 47 f1 86 95 2c e5 25 c6 37 04 40 81 b5 d6 1b  .G...,.%.7.@....
000000b0: 6d 78 9b 02 da e0 e5 5b e7 80 8a 4d d6 0c 78 ae  mx.....[...M..x.
000000c0: 0e d3 41 e0 ef bd 8e 55 01 0a 44 63 d3 9b 76 51  ..A....U..Dc..vQ
000000d0: f6 ab 3d f0 48 0e d3 20 db 8f 0f d1 6e b5 18 5d  ..=.H.. ....n..]
000000e0: 5f 78 83 7d 07 18 4e a8 91 98 d1 f9 f2 3b 82 5e  _x.}..N......;.^
000000f0: 22 46 bc 60 3e e5 00 00 df e3 05 61 2f 9f a6 15  "F.`>......a/...
00000100: 75 64 f7 ff c5 46 fb c7 42 56 46 8a 67 7a 75 d0  ud...F..BVF.gzu.
00000110: ab df a1 15 18 33 e5 e3 ba 16 c6 7f d4 74 bb ef  .....3.......t..
00000120: 1c 5c f7 46 88 9f 32 b8 45 3b d5 f3 62 f6 1c 29  .\.F..2.E;..b..)
00000130: 2d a3 5e 64 97 4b 77 4f 2a 9a 91 8a 9f cc e3 01  -.^d.KwO*.......
00000140: e7 bd 3e 95 d2 ff 1d b6 d2 e9 56 2a a7 bd fc 75  ..>.......V*...u
00000150: b9 d8 49 3b 26 bb f8 26 62 b9 45 3e ff d1 b2 be  ..I;&..&b.E>....
00000160: 5e 36 a2 50 3c 09 cb ad 4b e5 60 84 d9 1d 50 bc  ^6.P<...K.`...P.
00000170: 45 88 f8 35 62 0e 66 e2 59 47 1d                 E..5b.f.YG.
```

- : This block contains the encrypted file content
- : This block contains the encrypted compressed filename. This is also where the `file_encryption_info` structure begins.
- : This block contains the encrypted `filename_size` field
- : This block contains the encrypted `skipped_bytes` field
- : This block contains the encrypted `before_chunk_count` and `after_chunk_count` fields
- : This block contains the encrypted `file_encryption_key` which is unique for each encrypted file.
- : This block contains the `file_encryption_info_length` field. This is also where the `key_encryption_info` begins
- : This block contains the `checksum` field
- : This block contains the `encrypted_key_encryption_key` field encrypted using the RSA algorithm with the TA's public key for the infected system.

Because we know the current filename, we can compute the plaintext compressed filename. XOR-ing the encrypted compressed filename with the plaintext compressed filename gives us the following keystream:

```
00000000: FC 43 FD 30 B6 FC 8D C3  C9 47 F9 F3 64 06 53 88  .C.0.....G..d.S.
00000010: 95 EC BC 0C 58 52 93 71  3F 6E 31 39 AD C9 18 95  ....XR.q?n19....
00000020: 3A F0 2C 5A 26 06 88 61  AA 6A 66 F4 66 DB 81 3C  :.,Z&..a.jf.f..<
00000030: 0D 26 BB E8 6B 1A 24 7F  7C EE 40 96 EE 17 C5 30  .&..k.$.|.@....0
00000040: D7 D8 C5 95 5D CE 86 49  6B AB 13 03 B0 18 76 B9  ....]..Ik.....v.
00000050: 01 55 8D 81 F2 AB B4 1D  5F 74 D2 7C B6 25 9E F9  .U......_t.|.%..
00000060: 5F 69 3F F5 B3 2A FB 06  C5 B8 75 6E 98 CC 4B B3  _i?..*....un..K.
00000070: A6 A5 56 4B B1 E3 51 FA  3A 2F AF F3 15 67 15 F2  ..VK..Q.:/...g..
00000080: CE 73 63 A1 DA 9D 06 7F  79 E7 0C A8 50 5C FD 3C  .sc.....y...P\.<
00000090: 0E 45 F1                                          .E.
```

The content of the encrypted a.txt is shown below with similar color-coded fields:

```
00000000: e0 3e 90 78 c8 c0 6e 8a ae 29 5e 85 e7 02 9d a3  .>.x..n..)^.....
00000010: d3 d0 c2 1c f5 18 c0 45 f9 f3 68 06 53 88 c7 ec  .......E..h.S...
00000020: bc 0c 58 52 90 71 3f 6e 32 39 ad c9 f3 3a 5d f5  ..XR.q?n29...:].
00000030: de e1 33 3e a6 1d dd ef 26 83 0d ca 63 5c 74 ee  ..3>....&...c\t.
00000040: f0 6e 72 28 83 d0 3e d9 00 c8 42 cd 4f 3f 63 c3  .nr(..>...B.O?c.
00000050: 5d 2c 8c 9b 5d c3 61 bd 0a bd 65 c5 e4 54 8e dd  ],..].a...e..T..
00000060: 9d c5 86 43 9b 12 9a 00 e1 f5 56 40 5e 00 00 df  ...C......V@^...
00000070: e3 05 61 2f 9f a6 15 75 64 f7 ff c5 46 fb c7 42  ..a/...ud...F..B
00000080: 56 46 8a 67 7a 75 d0 ab df a1 15 18 33 e5 e3 ba  VF.gzu......3...
00000090: 16 c6 7f d4 74 bb ef 1c 5c f7 46 88 9f 32 b8 45  ....t...\.F..2.E
000000a0: 3b d5 f3 62 f6 1c 29 2d a3 5e 64 97 4b 77 4f 2a  ;..b..)-.^d.KwO*
000000b0: 9a 91 8a 9f cc e3 01 e7 bd 3e 95 d2 ff 1d b6 d2  .........>......
000000c0: e9 56 2a a7 bd fc 75 b9 d8 49 3b 26 bb f8 26 62  .V*...u..I;&..&b
000000d0: b9 45 3e ff d1 b2 be 5e 36 a2 50 3c 09 cb ad 4b  .E>....^6.P<...K
000000e0: e5 60 84 d9 1d 50 bc 45 88 f8 35 62 0e 66 e2 59  .`...P.E..5b.f.Y
000000f0: 47 1d                                            G.
```

XOR-ing the entire file_encryption_info block, starting at offset 0x0e to offset 0x6c with the keystream above would give us the following bytes:

```
00000000: e0 3e 90 78 c8 c0 6e 8a ae 29 5e 85 e7 02 61 e0  .>.x..n..)^...a.
00000010: 2e e0 74 e0 78 db 09 02 00 00 0c 00 00 00 52 00  ..t.x.........R.
00000020: 00 00 00 00 03 00 00 00 03 00 00 00 eb af 67 05  ..............g.
00000030: f2 bb 15 38 2e 7c 77 85 40 77 6b 11 e2 60 79 c8  ...8.|w.@wk..`y.
00000040: 4b 86 19 32 a7 af 42 37 40 5e ac da 8a 0f b4 1b  K..2..B7@^......
00000050: 98 b9 d1 55 db 8a 0a 16 19 be d5 dd 92 ed 8f 88  ...U............
00000060: 10 44 74 e8 2f 0f c5 74 33 89 e0 65 5e 00 00 df  .Dt./..t3..e^...
00000070: e3 05 61 2f 9f a6 15 75 64 f7 ff c5 46 fb c7 42  ..a/...ud...F..B
00000080: 56 46 8a 67 7a 75 d0 ab df a1 15 18 33 e5 e3 ba  VF.gzu......3...
00000090: 16 c6 7f d4 74 bb ef 1c 5c f7 46 88 9f 32 b8 45  ....t...\.F..2.E
000000a0: 3b d5 f3 62 f6 1c 29 2d a3 5e 64 97 4b 77 4f 2a  ;..b..)-.^d.KwO*
000000b0: 9a 91 8a 9f cc e3 01 e7 bd 3e 95 d2 ff 1d b6 d2  .........>......
000000c0: e9 56 2a a7 bd fc 75 b9 d8 49 3b 26 bb f8 26 62  .V*...u..I;&..&b
000000d0: b9 45 3e ff d1 b2 be 5e 36 a2 50 3c 09 cb ad 4b  .E>....^6.P<...K
000000e0: e5 60 84 d9 1d 50 bc 45 88 f8 35 62 0e 66 e2 59  .`...P.E..5b.f.Y
000000f0: 47 1d                                            G.
```

The recovered file_encryption_info fields are:

- Compressed filename: 61 e0 2e e0 74 e0 78 db 09 02 00 00 (compressed a.txt)
- filename_size: 0c 00 (0x0c)
- skipped_bytes: 00 00 52 00 00 00 00 00 (0x520000 in little-endian)
- before_chunk_count: 03 00 00 00 (0x03 in little-endian)
- after_chunk_count: 03 00 00 00 (0x03 in little-endian)
- file_encryption_key:

```
00000000: eb af 67 05 f2 bb 15 38 2e 7c 77 85 40 77 6b 11  ..g....8.|w.@wk.
00000010: e2 60 79 c8 4b 86 19 32 a7 af 42 37 40 5e ac da  .`y.K..2..B7@^..
00000020: 8a 0f b4 1b 98 b9 d1 55 db 8a 0a 16 19 be d5 dd  .......U........
00000030: 92 ed 8f 88 10 44 74 e8 2f 0f c5 74 33 89 e0 65  .....Dt./..t3..e
```

The recovered file_encryption_info structure allows us to decrypt the entire file following the decryption scheme described above.

Data corruption

This version of the LockBit v3 ransomware has a design flaw that can cause permanent data loss. LockBit v3 has a mutex checking mechanism to ensure only one instance of itself is running on the infected system:

```
void __stdcall CheckMutex()
{
  int *v0; // eax
  struct _SECURITY_ATTRIBUTES MutexAttributes; // [esp+0h] [ebp-10h] BYREF
  LPCWSTR lpName; // [esp+Ch] [ebp-4h]

  if ( byte_FC5129 )
  {
    lpName = GenerateMutexName();
    dword_FC51A4 = OpenMutexW(0x100000u, 0, lpName);
    if ( dword_FC51A4 )
    {
      NtClose(dword_FC51A4);
      if ( byte_FC512E )
        sub_FB04B4(0, 0, 0, (unsigned __int8)byte_FC512E);
      ExitProcess(0);
    }
    if ( (unsigned int)VersionStuff() < 0x3C )
      v0 = dword_FA9B60;
    else
      v0 = dword_FA9B00;
    MutexAttributes.nLength = 12;
    MutexAttributes.lpSecurityDescriptor = v0;
    MutexAttributes.bInheritHandle = 0;
    dword_FC51A4 = CreateMutexW(&MutexAttributes, 1, lpName);
    sub_FA68EC(lpName);
  }
}
```

However, this feature can be configured at build time and is disabled in our sample. The flag at byte_FC5129 is part of the sample's encrypted settings configured by the TA and set by the builder. When this feature is disabled, multiple instances of the ransomware can run on the infected system at one time.

The sample needs to process each file with exclusive access to the encryption logic. To do that, it attempts to terminate other processes that prevent exclusive access to the file. The sample uses the restart manager family of APIs (RmStartSession(), RmRegisterResource(), RmGetList()) to get a list of processes with open handles to the file being encrypted. It then terminates all of those processes.

This design can cause permanent data corruption because of the following reasons:

- With multiple instances of the same ransomware running on the same system, one instance can attempt to terminate the other instance that is encrypting the same file. In this case, the randomly generated file_encryption_key from the 1st instance can not be recovered. The file is permanently corrupted. We can detect the corruption by observing files with multiple extra extensions, signaling that the files were encrypted multiple times. Each instance of the ransomware can have multiple encryption threads running parallel, each of which encrypts one file at a time. Since the number of concurrent threads is quite low, the number of files being affected in this case can potentially be low.
- The sample may attempt to terminate another process that is currently writing and modifying the current file. This may cause data corruption depending on how the affected process is designed. We can not easily detect this case. However, the number of files being affected can be very high depending on the services running on the infected system and their utilization. Calif has observed files that are properly decrypted but are corrupted and not recognized by their associated applications.

Conclusion

Analyzing the ransomware could provide critical intelligence when evaluating response strategies to ransomware attacks. In this case, Calif observed flaws in the ransomware design that allowed affected organizations to reconsider the true value of the ransom demand. We hope our analysis helps demystify the inner workings of one ransomware variant. We also hope to encourage more sharing of technical analysis, curated intelligence, and valuable lessons across organizations. Security demands collaboration as no organizations operate in a vacuum. The more secure our peers, the safer we are against cyber criminals.

Appendix A: Reverse engineering detail

Anti-debugging

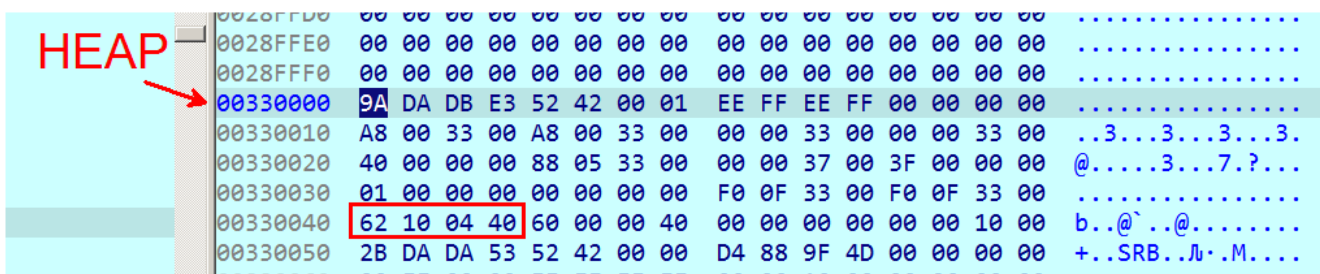
Typically, malware does not want to be analyzed. With a debugger, we can easily control the malware’s execution, dump data, or force the malware to execute a specific code path. Therefore, malware usually contains multiple anti-debugging checks. We found multiple said checks in this sample.

The first check occurs at 0x00FA63C5 (offset 0x57C5 into the file) as seen below:

```
.text:00FA63B6 push    0
.text:00FA63B8 push    0
.text:00FA63BA push    0
.text:00FA63BC push    0
.text:00FA63BE push    0
.text:00FA63C0 push    41002h
.text:00FA63C5 call    eax              ; RtlCreateHeap
.text:00FA63C7 mov     esi, eax
.text:00FA63C9 test    esi, esi
.text:00FA63CB jz      loc_FA654D

.text:00FA63D1 mov     eax, [eax+40h]
.text:00FA63D4 shr     eax, 1Ch
.text:00FA63D7 test    al, 4
.text:00FA63D9 jz      short loc_FA63DD

.text:00FA63DB rol     esi, 1
```

After manually resolving some Windows **Application Programming Interfaces** (**APIs**), the sample calls the RtlCreateHeap() function to create a new heap. The result is a HANDLE to a window HEAP structure provided by the operating system for the current process. This HEAP structure is undocumented by Microsoft. To better understand this structure, refer to other online resources regarding the Windows HEAP. Significant to anti-debugging mechanisms, the HEAP structure contains two flags: **Flag** and **ForceFlag**. These values change depending on whether the current process is running under a debugger.

In the screenshot above, the sample checks the Flag field, which is at offset 0x40 byte into the undocumented HEAP structure. The value of this Flag field is 0x40041062 as shown in the screenshot below:



The sample gets the most significant 4 bits of the flag by rotating the Flag field 28 (0x1c) bits to the right, and tests the result against 0x04. This effectively tests the most significant byte of the Flag field against 0x40000000 (HEAP_VALIDATE_PARAMETER_ENABLED) which is set if the current process is running under a debugger.

If the sample detects a debugger, it modifies the HANDLE to the current process's heap using the rol operation. This causes the process to crash if it ever tries to allocate any memory using the modified heap HANDLE in the future.

A similar check of the heap's ForceFlag field is shown below:

```
.text:00FA6844
.text:00FA6844
.text:00FA6844 ; Attributes: bp-based frame
.text:00FA6844
.text:00FA6844 ; int __stdcall AllocateHeapWithCheck(int)
.text:00FA6844 AllocateHeapWithCheck proc near
.text:00FA6844
.text:00FA6844 arg_0= dword ptr  8
.text:00FA6844
.text:00FA6844 push    ebp
.text:00FA6845 mov     ebp, esp
.text:00FA6847 call    GetPEB
.text:00FA684C mov     eax, [eax+_PEB.ProcessHeap]
.text:00FA684F test    dword ptr [eax+44h], 40000000h
.text:00FA6856 jz      short loc_FA685A
```

```
.text:00FA6858 ror     eax, 1
```

```
.text:00FA685A
.text:00FA685A loc_FA685A:
.text:00FA685A push    [ebp+arg_0]
.text:00FA685D push    8
.text:00FA685F push    eax
.text:00FA6860 call    RtlAllocateHeap
.text:00FA6866 pop     ebp
.text:00FA6867 retn    4
.text:00FA6867 AllocateHeapWithCheck endp
.text:00FA6867
```

In the screenshot above, the sample finds the heap using the current process's **Process Environment Block (PEB)**. Then, it tests the ForceFlag field, which is at offset 0x44, against 0x40000000 to detect a debugger.

These checks are scattered around the sample's logic near any heap operation. The easiest way to bypass these anti-debugging checks is to modify the process heap structures directly and reset both the Flag and ForceFlag fields' most significant byte to 0x00.

This sample also contains the following additional anti-debugging features:

- Checking beyond the bound of the allocated heap memory against magic constants like 0xABABABAB. These magic constants come from a Windows feature that adds additional guardrails to heap memory to quickly detect memory corruption bugs. This feature is only enabled if the current process is running under a debugger. This check can also be bypassed by modifying the Flagand ForceFlagfields of the heap.
- Calling NtProtectVirtualMemory() and RtlEncryptMemory() to encrypt the DbgUiRemoteBreakin() function. This causes the current process to crash if there is any attempt to attach a debugger afterwards. This does not have any effect if we start executing the sample using the debugger.
- Calling NtSetInformationThread() with ThreadHideFromDebugger (0x11) for the current thread. This call only happens a few times at the beginning of the execution flow. A quick way to bypass this is patching the function to simply return NT_SUCCESS (0x00).

Obfuscation

Manual API resolution

To avoid leaking capabilities and being tracked using the import hash, this sample manually resolves Windows APIs using the PEB.

The PEB contains all the properties of its associated process, including a list of loaded DLLs. The sample can walk this list of DLLs and their export tables to manually find addresses of the necessary Windows APIs. To further avoid leaking strings, the sample manually resolves Windows APIs using a hashing algorithm shown below:

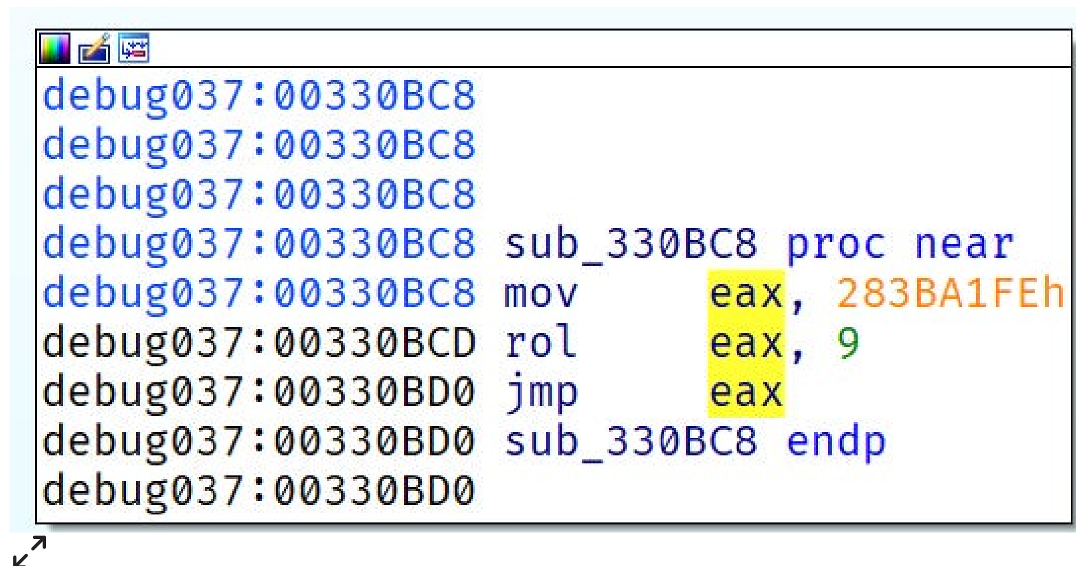
```
int __stdcall sub_FA11C4(WCHAR *pwcsString, int nHash)
{
  unsigned __int16 c; // ax

  *(&c + 1) = 0;
  do
  {
    c = *pwcsString++;
    if ( (unsigned int)c >= 'A' && (unsigned int)c <= 'Z' )
      c |= 0x20u;
    nHash = *(_DWORD *)&c + __ROR4__(nHash, 13);
  }
  while ( *(_DWORD *)&c );
  return nHash;
}
```

The sample applies the hashing algorithm above on the DLLs and their export names to find a match instead of comparing strings normally. However, in addition to the “Addition-Rotate Right 13” operation, the sample also XORs the result with the 0x10035FFF constant. This results in a set of API hashes that are different from other malware families using a similar technique.

Trampoline code

The sample doesn’t use the resolved APIs directly. Instead, for each API, it allocates a small memory chunk and builds a small piece of trampoline code which calculates and jumps to the target API.

For example, instead of executing a standard indirect call to NtOpenProcess() as an import, the sample calls to a pointer at address 0x00fc5474, which points to a function at address 0x00330bc8 on the heap. This function is shown below:



After the rol operation, eax becomes 0x7743FC50, which is the address of ntdll!NtOpenProcess(). This makes static analysis significantly more tedious. We would have a hard time tracking all the calls to the trampoline code.

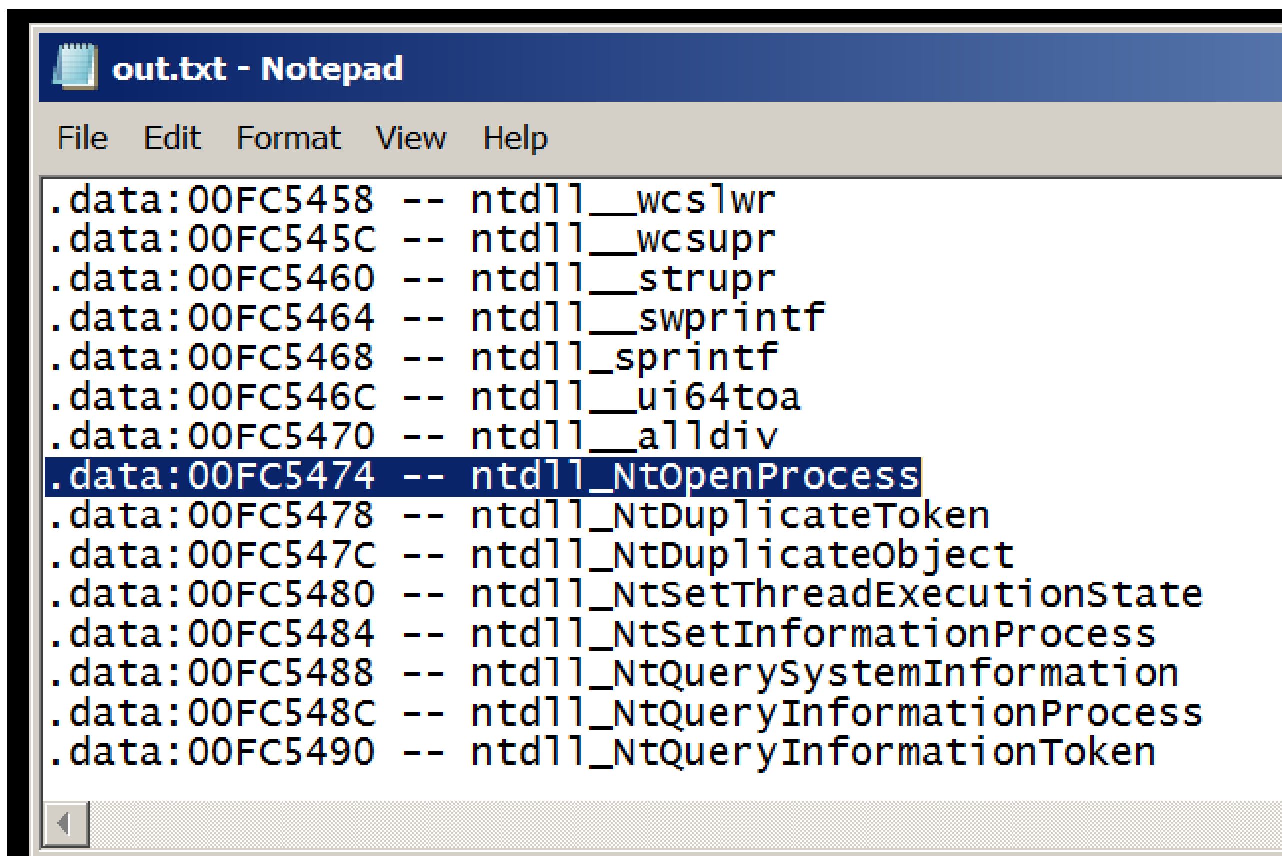
Because we can bypass the anti-debugging checks, we can use a debugger to help us automate the renaming of the trampoline calls. The logic to resolve APIs and setup the trampoline code is at 0x00FA5dA0. Using the IDA Free debugger, we can set a breakpoint at 0x00FA5DDB, which is the instruction right after the call to manually resolve Windows APIs. Then, we can edit the breakpoint to execute the following one-liner:

```
fprintf(fopen("out.txt", "a+"),
        "%s\n",
        sprintf("%a -- %s",
                GetRegValue("edi"),
                get_name(GetRegValue("eax"))
        )
)
```

This small snippet tells the IDA Free debugger to log the following items to the file "out.txt" in the current working directory:

- The current value of the edi register. This is the address of the trampoline code. In our example, this would be 0x00FA5dA0.
- The name of the value in the eax register. The eax register holds the address of the resolved API. In our example, eax would be 0x7743FC50. Within the current process context, this is the address of ntdll!NtOpenProcess().

Once the breakpoint is ready, we can let the sample execute through all the API resolution logic. At the end, we should see the out.txt file that looks similar to this:



After the sample finishes resolving all the APIs, we can dump the current process including all of its allocated memory for further analysis. Then, we can write a small IDA script to parse out.txt and rename all the trampoline calls to the appropriate APIs. This will help speed up our

static analysis significantly. An example of such a script is available in Appendix D.

Appendix B: Open-source decryption tool

The leaked LockBit v3 builder generated the encryptor and decryptor for Windows. Although the decryptor can run on Linux using Wine, Calif decided to re-implement the decryption logic in C for the following reasons:

- We want to run the decryptor natively on VMWARE ESXi.
- We want to confirm our understanding of the encryption scheme.
- We want to avoid executing the malware author's decryptor which may contain other data corruption bugs.
- Other affected organizations may also find our decryptor useful.

This section describes how we build a decryption tool for Linux. The tool is open-source and can be downloaded from GitHub.

Extracting the decryption function

Calif identified the two crypto functions to be Salsa20 (0x00FA20AC) and RSA with no padding (0x00FA17B4). Initially, instead of fully reverse-engineering these functions, we take the code directly from the binary and run it as shellcode inside a C wrapper. Calif's decisions were based on the following reasons:

- It would take us too long to fully analyze and confirm the algorithms.
- The sample uses a custom implementation of the two algorithms. Therefore, re-implementation or using a standard library may introduce discrepancies and bugs.

We extract the following items directly from the ransomware into shellcode that we can call using our wrapper:

- The Salsa20 encryption function and related functions
- The Raw RSA function and related functions
- The checksum calculation algorithm
- The APLib compression function and related functions

When preparing these functions, we also fix any absolute address references so we can call them correctly in our wrapper without causing a crash.

```
typedef uint32_t (__attribute__((stdcall)) *FUNC_CHECKSUM_tasklet)(void *, uint32_t, uint32_t);
typedef void     (__attribute__((stdcall)) *FUNC_RSA_decrypt)(void *, void *, void *);
typedef void     (__attribute__((stdcall)) *FUNC_SALSA20_decrypt)(uint32_t, void *, void *);
typedef void     (__attribute__((stdcall)) *FUNC_APLib_decompress)(void *, void *);

FUNC_CHECKSUM_tasklet   CHECKSUM_tasklet_func = NULL;
FUNC_RSA_decrypt        RSA_decrypt_func      = NULL;
FUNC_SALSA20_decrypt    SALSA20_decrypt_func  = NULL;
FUNC_APLib_decompress   APLib_decompress_func = NULL;
```

Implementation

The sample is compiled for a 32-bit Windows environment. To get the shellcode to run correctly, we also need to compile our code for this environment. By default, GCC would default to the **cdecl** calling convention, but Windows uses **stdcall**. We fix that by adding the attribute (__attribute__((stdcall)).

The data section of an executable is marked as non-executable, therefore we can not execute the shellcode directly from there. Instead, we allocate new memory pages with executable permission and copy the shellcode over.

```
0x80f21c0->0x80f3a08 at 0x000a91c0: .data ALLOC LOAD DATA HAS_CONTENTS
       (unsigned char (*)[1016]) 0x80f21e0 <rsa_decrypt>
       (unsigned char (*)[1073]) 0x80f25e0 <salsa_crypt>
       (unsigned char (*)[328]) 0x80f2a20 <apdecompress_bin>
```

Appendix C: Binary Information and Indicators of Compromise (IOCs)

The following IOCs come from our specific build of this variant of LockBit v3. Here are the components that may be different across different builds:

- The unique ID for this build: IzYqBW5pa.
- File hashes other than the hash of the icon and desktop background.

Binary information

Filename: LB3.exe

- File type: Windows Portable Executable (PE) x86
- File size: 156,160
- SHA256 hash: f34dd8449b9b03fedde335f8be51bdc7f96cda29a2dde176c3db667ba0713c6f

Filename: LB3Decryptor.exe

- File type: Windows Portable Executable (PE) x86
- File size: 33,280
- SHA256 hash: 8f0a2d5b47441fbcf1882aa41cae22fd0db057ccc38abad87ccc28813df3a83c

Indicators of Compromise

Host-based indicators (HBIs)

Volatile:

When configured, the sample creates the following mutex: Global\a91a66d6abc26041b701bf8da3de4d0f where a91a66d6abc26041b701bf8da3de4d0f is calculated from the embedded RSA private key

Files

- Filename: C:\ProgramData\IzYqBW5pa.ico where IzYqBW5pa is the unique ID for this specific variant.
- File type: ICO
- File size: 15,086
- SHA256 hash: 95e059ef72686460884b9aea5c292c22917f75d56fe737d43be440f82034f438

Filename: C:\ProgramData\IzYqBW5pa.bmp.

- File type: BMP
- File size: 86,708
- SHA256 hash: ef66e202c7a1f2a9bc27ae2f5abe3fd6e9e6f1bdd9d178ab510d1c02a1db9e4f

Filename: IzYqBW5pa.README.txt.

- File type: TXT
- File size: 6,197
- SHA256 hash: af23f7d2cf9a263802a25246e2d45eaf4a4f8370e1b6115e79b9e1e13bf20bfe

Registry:

- Path: HKEY_CLASSES_ROOT\.IzYqBW5pa\DefaultIcon
- Value: C:\ProgramData\IzYqBW5pa.ico

Network-based indicators (NBIs):

- When configured, the sample communicates with the configured C2 server using HTTP Protocol POST method. This specific variant is not configured with a C2 server.
- When communicating with the C2 server, the sample uses the following User-Agent string: Chrome/91.0.4472.77.
- Communication with the C2 server is encrypted using the AES algorithm. This specific variant is not configured to communicate with the C2 server. Therefore, it also does not contain the AES key.

Appendix D: IDC script to rename functions

```
#include <idc.idc>

static process(line) {
    // example line: .data:00FC5410 -- ntdll_RtlCreateHeap
    auto idx = strstr(line, " -- ");

    // saddr: .data:00FC5410
    auto saddr = substr(line, 0, idx);

    // name: ntdll_RtlCreateHeap
    auto name = substr(line, idx + 1, -1);

    // old saddr: .data:00FC5410
    // new saddr: 00FC5410 as a string
    // addr     : 0x00FC5410
    auto _idx = strstr(saddr, ":");
    saddr = substr(saddr, _idx + 1, -1);
    auto addr = xtol(saddr);

    // old name: ntdll_RtlCreateHeap
    // new name: RtlCreateHeap
    _idx = strstr(name, "_");
    name = substr(name, _idx + 1, -1);

    auto len = strlen(name);
    // NULL terminate the last byte
    name[len-1] = '\0';

    Message("Addr: 0x%x, name: %s\n", addr, name);
    set_name(addr, name, SN_NOCHECK|SN_FORCE);
}


static load_file() {
    auto fd = fopen("out.txt", "r");
    auto line = readstr(fd);
    while (value_is_string(line)) {
        process(line);
        line = readstr(fd);
    }
    fclose(fd);
    return 0;
}


static main() {
    load_file();
}
```

Appendix E: Chunk counts and skip bytes

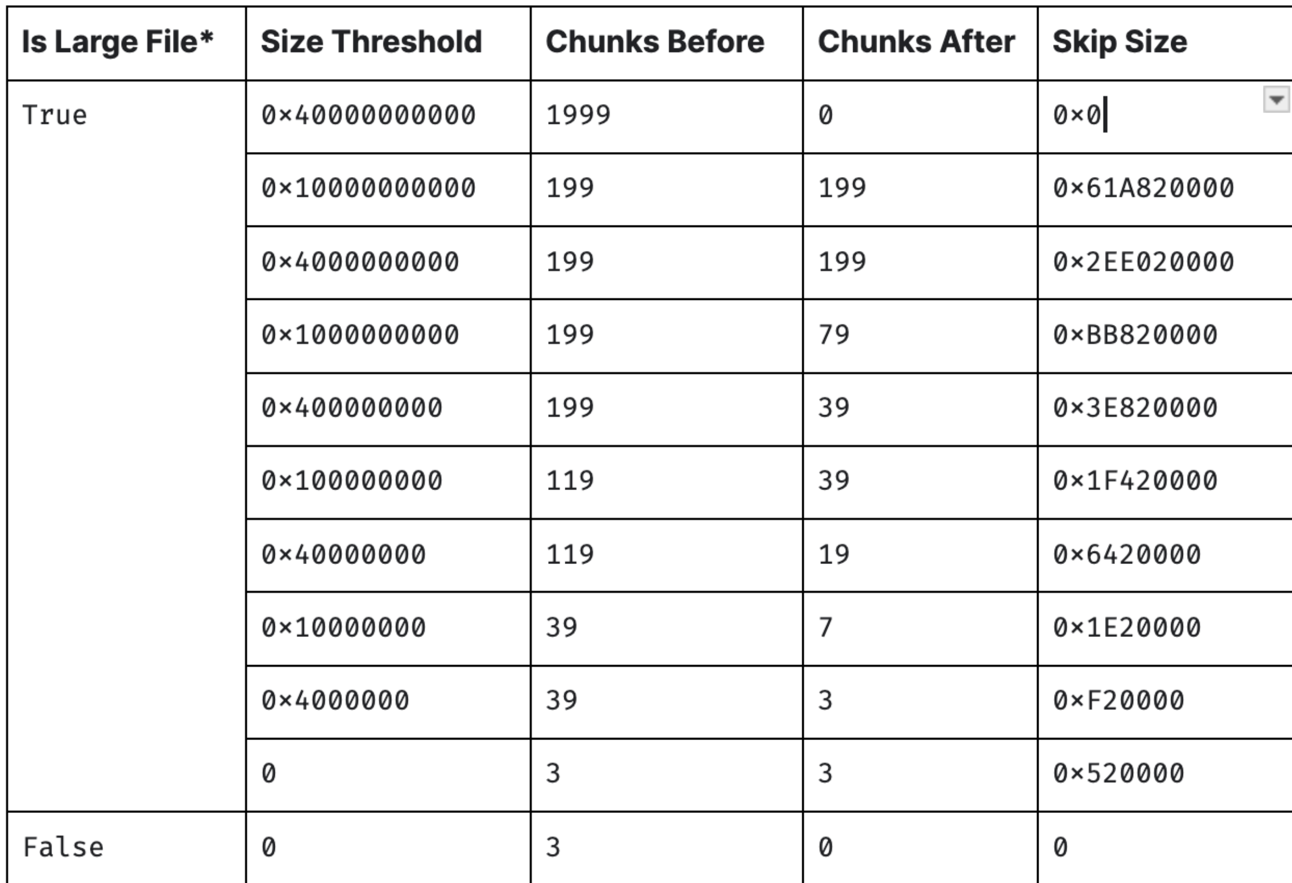
Grouping algorithm

Each chunk of the file belongs to one of the three groups (***before group***, ***skip group***, ***after group***). But for the sake of simplicity, let's only consider the state of each chunk: encrypted (*before group* or *after group*), or unencrypted (*skip group*).

To determine if a chunk needs encrypting or decrypting, we can use the following algorithm:

```
chunk_state = ''
# first 'before_chunk_count' chunks belong to before group and are encrypted
crypt_chunk_count = before_chunk_count
skip_chunk_count = (skipped_bytes / 0x20000) -1
skip_count = skip_chunk_count

for chunk in chunks:
        if (crypt_chunk_count):
                chunk_state = "en(de)crypt"
                crypt_chunk_count = crypt_chunk_count - 1
        else:
                chunk_state = "skip" # belongs to a skip group
                skip_count = skip_count - 1
                if (skip_count == 0):
                        crypt_chunk_count = after_chunk_count
```

The decrypted file_encryption_info contains the value for before_chunk_count, after_chunk_count and skipped_bytes. To see how and where they are generated, refer to the File encryption section**.** The sample determines the chunk count based on the file size as described in the following table:



Is Large File*	Size Threshold	Chunks Before	Chunks After	Skip Size
True	0×40000000000	1999	0	0×0
	0×10000000000	199	199	0×61A820000
	0×4000000000	199	199	0×2EE020000
	0×1000000000	199	79	0×BB820000
	0×400000000	199	39	0×3E820000
	0×100000000	119	39	0×1F420000
	0×40000000	119	19	0×6420000
	0×10000000	39	7	0×1E20000
	0×4000000	39	3	0×F20000
	0	3	3	0×520000
False	0	3	0	0

Comments