Part 2: LockBit 2.0 ransomware bugs and database recovery attempts

techcommunity.microsoft.com/t5/security-compliance-and-identity/part-2-lockbit-2-0-ransomware-bugs-and-database-recovery/ba-p/3254421

March 11, 2022

In Part 1 of this series (which you can find here), we provided background about our analysis of the LockBit 2.0 ransomware and described our suspicions that "faulty crypto" was at play. In this post, we will outline the issues that the decryptor poses and how we simply cannot trust it and must remove it from any equation we intend on using to successfully decrypt these database files.

Disclaimer*: The technical information contained in this article is provided for general informational and educational purposes only and is not a substitute for professional advice. Accordingly, before taking any action based upon such information, we encourage you to consult with the appropriate professionals. We do not provide any kind of guarantee of a certain outcome or result based on the information provided. Therefore, the use or reliance of any information contained in this article is solely at your own risk.*

If only it were so easy…

Our earlier Procmon observations identified the encryptor randomly encrypting 65k bytes *after* it was only supposed to encrypt the first 4k. So, while we do successfully decrypt the *intended* encrypted region of the encrypted file, which is the first 0x1000 bytes, we fail to identify and decrypt the *unintended* regions which are splattered throughout the now-decrypted file due to the bug we've outlined in the encryptor.

And as this is a customer-provided file, we don’t have the luxury of a Procmon or TTD trace to quickly identify the corruption. To tackle this problem, we instead crafted an algorithm that will be outlined shortly, that can scan the encrypted file and identify all regions of unintended encryption.

```
Scanning all identifiable encrypted regions:
        +0000000A000000
        +00000014000000
        +0000001E000000
        +00000028000000
        +00000032000000
        +0000003C000000
        +00000046000000
        +00000050000000
        +0000005A000000
        +00000064000000
        +0000006E000000
        +00000078000000
        +00000082000000
        +0000008C000000
        +00000096000000
        +000000A0000000
        +000000AA000000
        +000000B4000000
        +000000BE000000
        +000000C8000000
        +000000D2000000
        +000000DC000000
        +000000E6000000
        +000000F0000000
        +000000FA000000
```

We can see from the output that there are encrypted regions at every 0xA000000 offsets.

Well, not exactly...

Figure 12. Example of our first implementation of the algorithm that identifies regions of unintended encryption

```
9FF:FEE0h: 31 38 31 30 33 38 32 2D 30 31 35 32 2D 4D 41 4E  1810382-0152-MAN
9FF:FEF0h: 00 00 21 21 21 21 21 21 21 21 21 21 21 21 21 21  ..!!!!!!!!!!!!!!
9FF:FF00h: 21 21 21 21 21 21 21 21 21 21 21 21 B1 1E 72 1E  !!!!!!!!!!!!±.r.
9FF:FF10h: 33 1E F4 1D B5 1D 76 1D 35 1D F4 1C B3 1C 72 1C  3.ô.µ.v.5.ô.³.r.
9FF:FF20h: 30 1C EE 1B AC 1B 6A 1B 28 1B E6 1A A4 1A 62 1A  0.î.¬.j.(.æ.¤.b.
9FF:FF30h: 23 1A E2 19 A1 19 62 19 23 19 E4 18 A5 18 66 18  #.â.¡.b.#.ä.¥.f.
9FF:FF40h: 27 18 E5 17 A3 17 64 17 25 17 E6 16 A4 16 62 16  '.å.£.d.%.æ.¤.b.
9FF:FF50h: 20 16 DE 15 9C 15 5A 15 18 15 D6 14 94 14 52 14   .Þ.œ.Z...Ö.”.R.
9FF:FF60h: 13 14 D4 13 95 13 56 13 17 13 D8 12 99 12 5A 12  ..Ô.•.V...Ø.™.Z.
9FF:FF70h: 1B 12 DC 11 9D 11 5E 11 1C 11 DA 10 98 10 56 10  ..Ü...^...Ú.˜.V.
9FF:FF80h: 17 10 D8 0F 99 0F 5A 0F 1B 0F DC 0E 9D 0E 5E 0E  ..Ø.™.Z...Ü...^.
9FF:FF90h: 1F 0E E0 0D A1 0D 62 0D 23 0D E4 0C A5 0C 66 0C  ..à.¡.b.#.ä.¥.f.
9FF:FFA0h: 27 0C E8 0B A9 0B 6A 0B 2B 0B E9 0A A7 0A 68 0A  '.è.©.j.+.é.§.h.
9FF:FFB0h: 29 0A EA 09 AB 09 6C 09 2D 09 EE 08 AF 08 70 08  ).ê.«.l.-.î.¯.p.
9FF:FFC0h: 31 08 F2 07 B3 07 71 07 2F 07 ED 06 AB 06 69 06  1.ò.³.q./.í.«.i.
9FF:FFD0h: 27 06 E5 05 A3 05 61 05 1F 05 DD 04 9B 04 59 04  '.å.£.a...Ý.›.Y.
9FF:FFE0h: 17 04 D5 03 93 03 54 03 15 03 D6 02 97 02 58 02  ..Õ.“.T...Ö.—.X.
9FF:FFF0h: 19 02 1C 07 DA 04 9C 02 60 00 35 10 E0 0D 9E 0B  ....Ú.œ.`.5.à.ž.
A00:0000h: 96 51 80 C4 09 2F 7F 9C AC E5 25 82 6F F3 82 2A  –Q€Ä./.œ¬å%‚oó‚*
A00:0010h: 9D 3F 4E F5 E9 F3 20 C9 19 9A 15 A4 41 B9 66 9F  .?Nõéó É.š.¤A¹fŸ
A00:0020h: 05 6B E1 E1 45 C1 3E 5C 3C 75 BD 22 5B D9 FF AC  .kááEÁ>\<u½"[Ùÿ¬
A00:0030h: 67 29 0C 84 D9 CA 1A B6 65 EF F5 DE 32 77 D2 9D  g).„ÙÊ.¶eïõÞ2wÒ.
A00:0040h: C8 C3 FC 98 55 29 1D 88 3A CB C7 DB 8C C3 43 E3  ÈÃü˜U).ˆ:ËÇÛŒÃCã
A00:0050h: E0 B9 B9 CD 7F 48 07 1A 1E BE 68 F1 98 E2 3D 9C  à¹¹Í.H...¾hñ˜â=œ
A00:0060h: C0 6C 58 4D 50 7C C0 8A D8 67 AE B8 A3 80 4A 5A  ÀlXMP|ÀŠØg®¸£€JZ
A00:0070h: C0 A1 F8 32 E5 CA AD CD 53 D4 5D D7 AA 54 51 B2  À¡ø2åÊ-ÍSÔ]×ªTQ²
A00:0080h: 4C 6B 30 0A BF EF 55 3E 11 31 93 C0 D8 27 2C 34  Lk0.¿ïU>.1“ÀØ',4
A00:0090h: 7F 41 16 7C 7C 12 D0 93 2D 97 E9 B3 05 35 D5 68  .A.||.Ð“-—é³.5Õh
A00:00A0h: A7 2D BB 6E 53 7D 1A EF E0 83 17 4A 95 98 18 98  §-»nS}.ïàƒ.J•˜.˜
A00:00B0h: 26 56 D0 91 D8 62 FA BF B2 E4 8D 21 55 6E 91 B4  &VÐ‘Øbú¿²ä.!Un‘´
A00:00C0h: A4 69 F9 6C 24 5D 5B C6 BA D6 08 B5 DD 93 41 9C  ¤iùl$][ÆºÖ.µÝ“Aœ
A00:00D0h: 38 75 BA 5A 16 B2 D4 58 98 57 FC 3A 12 E0 AF 34  8uºZ.²ÔX˜Wü:.à¯4
A00:00E0h: D3 B1 1C C4 D6 DC 59 B3 C4 CA 10 BA 1A B9 7D F7  Ó±.ÄÖÜY³ÄÊ.º.¹}÷
A00:00F0h: A6 36 05 C4 F0 AB 9F 0B 9E AE C4 F6 F7 5D E8 43  ¦6.Äð«Ÿ.ž®Äö÷]èC
A00:0100h: AA BA E6 B6 15 B0 07 8A 41 4D 3C F5 C6 CC F1 F3  ªºæ¶.°.ŠAM<õÆÌñó
A00:0110h: 20 08 2B 9E B6 CD 59 AB 8F 16 F0 5F E8 C4 C9 82   .+ž¶ÍY«..ð_èÄÉ‚
```

Valid data (unencrypted)

Corrupted data (encrypted)

Figure 13. Valid data and corrupted data

In case it’s not clear by now, patience, the willingness to remain calm and *wait*, seems to be a virtue that is prioritized in blocking I/O. Due to the LockBit 2.0 developers not giving this virtue its due diligence, it gets worse for us in the regard that the decryptor itself suffers from *the exact same bugs* as the encryptor. It fails to handle **STATUS_PENDING** states; it falsely assumes all NTSTATUS errors/non-successes values are signed. To put it much more succinctly, *we cannot trust the decryptor.*

```
__decrypt_contents:                        ; CODE XREF: lb_cb_decrypt+263↑j
  lea     ecx, [esp+3E0h+var_258_AES_context]
  call    mbedtls_aes_setkey_dec
  mov     eax, [edi+lb_crypt_t.buffer]
  lea     ecx, [esp+3E0h+var_258_AES_context]
  mov     edx, [edi+lb_crypt_t.encrypted_size]
  push    eax
  push    eax
  lea     eax, [edi+lb_crypt_t.iv]
  push    eax
  call    mbedtls_aes_crypt_cbc
  add     esp, 0Ch
  lea     eax, [esp+3E0h+var_258_AES_context]
  push    118h
__write_decrypted_content_back_to_file:
  mov     eax, [esp+3ECh+var_3C4]       ; lb_crypt_t
  add     esp, 0Ch
  mov     [eax+lb_crypt_t.cmd_code], 2  ; set cmdcode to resize next
  mov     ecx, [esp+3E0h+var_3C4]       ; lb_crypt_t
  mov     edi, [esp+3E0h+var_3D0]       ; lb_encrypt_file_t
  push    0
  mov     [esp+3E4h+CompletionKey], edi
  lea     eax, [ecx+lb_crypt_t.byte_offset]
  push    eax
  push    [ecx+lb_crypt_t.encrypted_size]
  lea     eax, [ecx+lb_crypt_t.io_status.anonymous_0.Pointer]
  push    [ecx+lb_crypt_t.buffer]
  push    eax
  push    ecx
  push    0
  push    0
  push    [edi+lb_encrypt_file_t.file_handle]
  call    lb_resolve_ntwritefile
  call    eax
  test    eax, eax
  jns     __close_current_io_processing ; ERROR: no check for STATUS_PENDING
```

Figure 14. Part of the decryptor code that illustrates the trust issues that we have with it

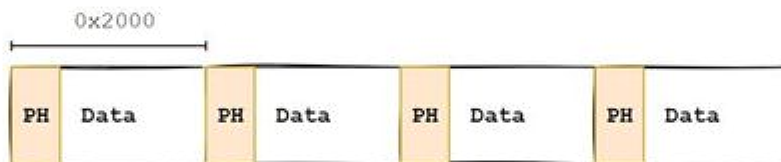
Because of suffering from the identical misconceptions as the encryptor, when decrypting the database file that ended up having the appearance of being correctly decrypted, it in actuality *further corrupted* the file trying to decrypt regions that were never encrypted to begin with!

These random regions further complicate the situation for us and now force us to deal with them. Or do they? Fixing the encrypted unintended regions that were a result of the encryptor is a logical step; fixing the newly encrypted regions from software that is solely responsible for decrypting is not. So, to make our lives *easier* we took the logical high road and decided to make our own decryptor.

Encryption overspill and rebuilding database files

Before outlining our decryptor and the details of the algorithm alluded to earlier, we must point out yet another subtlety that must be addressed. Due to the unpredictable behavior the encryptor is capable of we are facing further issues, outside of the encryption procedure itself, of potentially irrecoverable corruption. The best way to see this is to have some semblance of the underlying structures involved for a *.ndf* file, which is the format of the database files that we had to work with. The understanding of this structure, at least the essential parts relevant to us, serves as the basis for our recovery algorithm.

For our purposes, it suffices to understand that for every 0x2000 bytes, we have what are called *pages*. Each page begins with a *header* that is 0x60 bytes in size. Pages can also be classified as empty; 0x2000 bytes full of 0's.



The header contains valuable metadata that we can leverage to identify areas of corruption. Upon careful examination and side-by-side comparison of all the *.ndf* files that we had to work with, we were able to uncover three relevant properties in the header that would serve as the cornerstone of our recovery algorithm.

```
Page Type   Page Index   Database Identifer

       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
0000h: 01 0F 00 00 08 02 00 00 00 00 00 00 00 00 00 00  ................
0010h: 00 00 00 00 00 00 01 00 63 00 00 00 63 1B D6 08  ........c...c.Ö.
0020h: 00 00 00 00 04 00 00 00 A0 8B 07 00 B0 96 07 00  ........ ‹..°–..
0030h: 02 00 00 00 00 00 00 00 00 00 00 00 7E D5 0B C4  ............~Õ.Ä
0040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

2000h: 01 0B 00 00 00 02 00 00 00 00 00 00 00 00 00 00  ................
2010h: 00 00 00 00 00 00 01 00 63 00 00 00 02 00 FC 1F  ........c.....ü.
2020h: 01 00 00 00 04 00 00 00 EC 8B 07 00 00 65 2B 00  ........ì‹...e+.
2030h: 03 00 00 00 00 00 00 00 00 00 00 00 8F 35 B8 04  .............5¸.
2040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
2050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

4000h: 01 08 00 00 00 02 00 00 00 00 00 00 00 00 5A 00  ..............Z.
4010h: 00 00 00 00 00 00 02 00 63 00 00 00 06 00 F6 1F  ........c.....ö.
4020h: 02 00 00 00 04 00 00 00 E2 8B 07 00 D0 7A 09 00  ........â‹..Ðz..
4030h: 23 00 00 00 00 00 00 00 00 00 00 00 0F 54 04 82  #............T.‚
4040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
4050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

8000h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
8010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
8020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
8030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
8040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
8050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Figure 15. Illustration of all three properties as shown in a hex editor

The *PageType* field identifies the type of that individual page, which from our understanding can either be a 1 (an occupied page) or a 0 (unoccupied/empty page). The *PageIndex* property identifies the current page and its location within the database file.

So, "Page 0" would be at "index 0"; "Page 1" would be at "index 1", and so on. It is a way to get to individual pages inside the *.ndf* file. And speaking of the database file itself, what follows the *PageIndex* is yet another unique value that serves to identify the entire *.ndf* file as a whole. In the above case this is indicated by the value "4", but other database files had "3" as a value here instead. What we care about, which is being able to identify the integrity of each individual page that we come across, is that this is a value that we know must be constant throughout each page for each database file we are processing.

From having a sufficient understanding of the page header, we can construct an algorithm to verify the integrity of each individual page, which in turn allows us to also identify any potential corruption to any of the pages. We can iterate from the start of the file at 0x2000

(page sized) increments and inspect the validity of each header. Wherever we don't have a valid header, we at least know that something is going on at that location which we can investigate further as needed.

For example, if we wanted to verify that a specific page is valid, we ascertain that the first byte is either a 1 or a 0, and if it is a 1, we go to the 0x20 offset from the start of the header, pull out the 4-byte value there, and calculate whether the *PageIndex* value matches the offset to the start of the page header. We also further validate that the *database identifier* is consistent throughout.

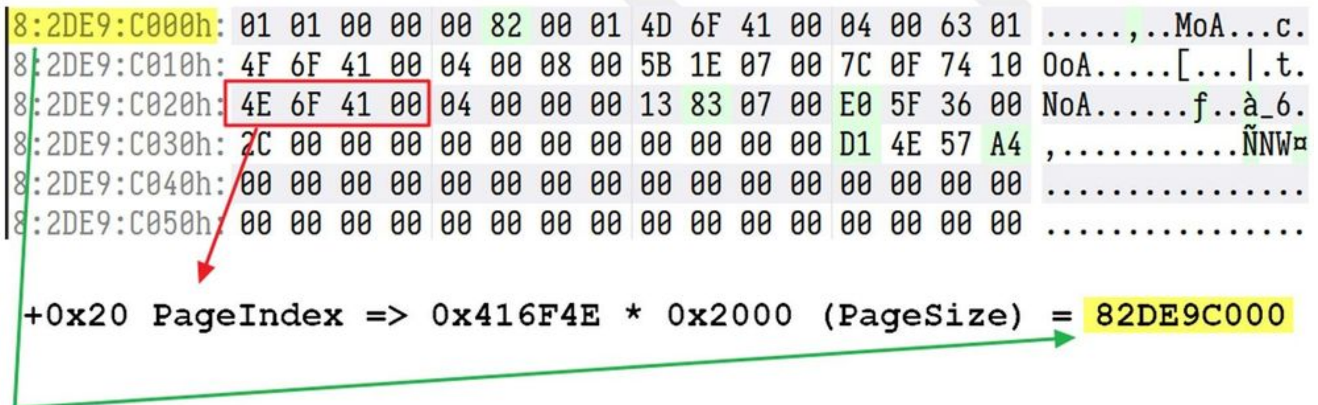


Figure 16. Calculating the PageIndex value

If none of the above conditions are satisfied, then we are looking at corrupted data and we can begin to programmatically identify all these areas. In our case, these were almost exclusively the unintended encrypted regions we outlined earlier.

If the above conditions are indeed satisfied, we know that we have a valid page at its correct location, so we can note that as well.

Where the conditions are *half* satisfied is where it gets interesting i.e., we pass the *PageType* and *Database Identifier* check but the *PageIndex* value doesn't match the offset to the start of the page header. We classify these as a **misaligned header** because the *PageIndex* value is pointing to the location of where this header is *supposed* to be:

```
>> valid header @ +F50EC000
>> valid header @ +F50EE000
>> valid header @ +F50F0000
>> valid header @ +F50F2000
>> valid header @ +F50F4000
>> valid header @ +F50F6000
!! invalid header identifed @ +F50F8000
!! misaligned header @        +F50F9000
   Target should be @         +F50FA000
```

```
F50F:9000h: 01 01 00 00 00 82 00 01 9A 52 29 00 03 00 4B 00  .....‚..šR)...K.
F50F:9010h: 9B 52 29 00 03 00 42 00 FA 1D 07 00 6E 00 0E 1F  ›R)...B.ú...n...
F50F:9020h: 7D A8 07 00 04 00 00 00 BD 8B 07 00 E8 A6 0F 00  }¨......½‹..è¦..
F50F:9030h: 2A 00 00 00 00 00 00 00 00 00 00 00 E8 9F E8 60  *...........èŸè`
F50F:9040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

+0x20 PageIndex => 0x07A87D * 0x2000 (PageSize) = 0xF50FA000

Figure 17. Identifying a misaligned header (off by 0x1000 bytes)

Also, whenever we hit a zero page (*PageType* == 0), we can safely ignore and continue.

Moving closer to the ultimate goal: successfully restoring all encrypted database files

In the sections described above, we discussed the commonality that all of these database files share: their file format. We outlined the characteristics of an algorithm that can validate the integrity of these database files and categorized four types of classifications by leveraging our understanding of how a *.ndf* database file is supposed to be structured. This, in theory, should be able to deal with all intended and *unintended* corruption the encryptor and decryptor are known to impose.

Now it's time to put this theory and understanding into practice and build upon this algorithm to achieve our ultimate goal: the successful restoration of all encrypted database files.

Recovering encrypted and corrupted database files

With the stage now set given that all known underlying issues have been exposed, we approached the problem in the following manner.

1. Identify and decrypt (fully) any encrypted database files with our homemade decryptor
2. Process the output of step 1 and account for any misaligned page headers accordingly
3. Process the output of step 2 and “clean up” the final remnants of leftover data from the misaligned headers

Step 1. Identify and decrypt (fully) any encrypted database files with our homemade decryptor

We come back to our homemade decryptor now. The details of how our homemade decryptor works under the hood are not as relevant as understanding how we're going to leverage it. More important is being able to identify all the encrypted regions throughout the file and not letting the modified LockBit 2.0 decryptor loose on it to further destroy it.

But the primary structure of our decryptor is to identify, decrypt, and extract the necessary initialization vector and AES key for the encrypted file, and then utilize this information to carry out the AES decryption through the mbedtls library, which is exactly the same 3rd party library that the Lockbit 2.0 developers are using.

```
::mbedtls_aes_init( &ctx_ );
::mbedtls_aes_setkey_dec( &ctx_, key_.data(), 128 );
bool decrypt_1k(
    __inout u8* data )
{
    auto r = ::mbedtls_aes_crypt_cbc(
        &ctx_,
        MBEDTLS_AES_DECRYPT,
        0x1000,
        iv_.data(),
        data,
        data
    );
```

Figure 18. AES decryption through the mbedtls library

The approach we took to finding the encrypted regions was outlined earlier and revolves around what a valid, misaligned, or null page is expected to look like. We further build on this with our decryptor by adding Shannon entropy checks on buffers of **0x1000** bytes in size. Any buffer that has a very high level (>= *7.8*), we will decrypt and then further validate the decrypted data based on what a page header constitutes.

Taking advantage of the fact that a *.ndf* file is so "well" structured i.e., every 0x2000 bytes will always follow a guaranteed format, we can run this algorithm to identify every encrypted region within the file and successfully decrypt it. Further validation after the decryption is also required because *.ndf* file formats unsurprisingly house compressed data which can flag on our entropy scan. We need to ignore all these cases and leave them as is.

Below is the successful output for step one. Also note, for one file the distance between encrypted regions is at **0xA000000** intervals, whereas the other is at **0x17570000** intervals. Again, the effects of the unpredictable nature of malformed asynchronous I/O, which do not pose a threat anymore.

```
kd > .\sql_parser.exe
[*] Encrypted: A000000
[*] Encrypted: 14000000
[*] Encrypted: 1E000000
[*] Encrypted: 28000000
[*] Encrypted: 32000000
[*] Encrypted: 3C000000
[*] Encrypted: 46000000
[*] Encrypted: 50000000
[*] Encrypted: 5A000000
[*] Encrypted: 64000000
[*] Encrypted: 6E000000
[*] Encrypted: 78000000
[*] Encrypted: 82000000
[*] Encrypted: 8C000000
[*] Encrypted: 96000000
[*] Encrypted: A0000000
[*] Encrypted: AA000000
[*] Encrypted: B4000000
[*] Encrypted: BE000000
[*] Encrypted: C8000000
[*] Encrypted: D2000000
[*] Encrypted: DC000000
[*] Encrypted: E6000000
[*] Encrypted: F0000000
```

```
[*] Encrypted: 17570000
[*] Encrypted: 2EAE0000
[*] Encrypted: 46050000
[*] Encrypted: 5D5C0000
[*] Encrypted: 74B30000
[*] Encrypted: 8C0A0000
[*] Encrypted: A3610000
[*] Encrypted: 1BB74E000
[*] Encrypted: 1D2CBE000
[*] Encrypted: 1EA22E000
[*] Encrypted: 20179E000
[*] Encrypted: 218D0E000
[*] Encrypted: 23027E000
[*] Encrypted: 2477EE000
[*] Encrypted: 25ED5E000
[*] Encrypted: 2762CE000
[*] Encrypted: 28D83E000
[*] Encrypted: 2A4DAE000
[*] Encrypted: 3BCEEC000
[*] Encrypted: 3D445C000
[*] Encrypted: 3EB9CC000
[*] Encrypted: 402F3C000
```

```
            0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
A00:0000h: 96 51 80 C4 09 2F 7F 9C AC E5 25 82 6F F3 82 2A  -Q€Ä./.œ¬å%,oó,*
A00:0010h: 9D 3F 4E F5 E9 F3 20 C9 19 9A 15 A4 41 B9 66 9F  .?Nõéó É.š.¤A¹fŸ
A00:0020h: 05 6B E1 E1 45 C1 3E 5C 3C 75 BD 22 5B D9 FF AC  .kááEÁ>\<u½"[Ùÿ¬
A00:0030h: 67 29 0C 84 D9 CA 1A B6 65 EF F5 DE 32 77 D2 9D  g).„ÙÊ.¶eïõÞ2wÒ.
A00:0040h: C8 C3 FC 98 55 29 1D 88 3A CB C7 DB 8C C3 43 E3  ÈÃü˜U).^:ËÇÛŒÃCã
A00:0050h: E0 B9 B9 CD 7F 48 07 1A 1E BE 68 F1 98 E2 3D 9C  à¹¹Í.H...¾hñ˜â=œ
A00:0060h: C0 6C 58 4D 50 7C C0 8A D8 67 AE B8 A3 80 4A 5A  ÀlXMP|ÀŠØg®¸£€JZ
A00:0070h: C0 A1 F8 32 E5 CA AD CD 53 D4 5D D7 AA 54 51 B2  À¡ø2åÊ-ÍSÔ]×ªTQ²
A00:0080h: 4C 6B 30 0A BF EF 55 3E 11 31 93 C0 D8 27 2C 34  Lk0.¿ïU>.1“ÀØ',4
A00:0090h: 7F 41 16 7C 7C 12 D0 93 2D 97 E9 B3 05 35 D5 68  .A.||.Ð“-—é³.5Õh
A00:00A0h: A7 2D BB 6E 53 7D 1A EF E0 83 17 4A 95 98 18 98  §-»nS}.ïàƒ.J•˜.˜
A00:00B0h: 26 56 D0 91 D8 62 FA BF B2 E4 8D 21 55 6E 91 B4  &VÐ‘Øbú¿²ä.!Un‘´
A00:00C0h: A4 69 F9 6C 24 5D 5B C6 BA D6 08 B5 DD 93 41 9C  ¤iùl$][ÆºÖ.µÝ“Aœ
A00:00D0h: 38 75 BA 5A 16 B2 D4 58 98 57 FC 3A 12 E0 AF 34  8uºZ.²ÔX˜Wü:.à¯4
A00:00E0h: D3 B1 1C C4 D6 DC 59 B3 C4 CA 10 BA 1A B9 7D F7  Ó±.ÄÖÜY³ÄÊ.º.¹}÷
A00:00F0h: A6 36 05 C4 F0 AB 9F 0B 9E AE C4 F6 F7 5D E8 43  ¦6.Äð«Ÿ.ž®Äö÷]èC
A00:0100h: AA BA E6 B6 15 B0 07 8A 41 4D 3C F5 C6 CC F1 F3  ªºæ¶.°.ŠAM<õÆÌñó
A00:0110h: 20 08 2B 9E B6 CD 59 AB 8F 16 F0 5F E8 C4 C9 82   .+ž¶ÍY«..ð_èÄÉ‚
A00:0120h: B1 E2 95 EE 56 77 CA BB 45 52 8E E6 26 51 3F D8  ±â•îVwÊ»ERŽæ&Q?Ø
A00:0130h: C8 A5 E2 C8 96 15 D8 92 E2 55 C2 2E 33 04 90 0A  È¥âÈ–.Ø’âUÂ.3...
A00:0140h: 13 4F DC EF 02 0E E7 D3 CC C6 8C 11 0A 4C A2 B7  .OÜï..çÓÌÆŒ..L¢·
A00:0150h: 6B 96 AE A6 2A B8 FF F0 A1 7B 2D F9 3F DC 4B 0B  k–®¦*¸ÿð¡{-ù?ÜK.
A00:0160h: 42 8F E9 F3 09 27 8F B7 F0 EC C9 BD 67 70 B9 2D  B.éó.'.·ðìÉ½gp¹-
A00:0170h: BD 48 EF 9A 92 73 D1 07 CB 65 2B 3B 30 EC 84 A8  ½Hïš’sÑ.Ëe+;0ì„¨
A00:0180h: 22 DB 07 AE 9D 9B F1 A8 AC A0 AE D8 EC B2 BC 3C  "Û.®.›ñ¨¬ ®Øì²¼<
```

```
            0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
A00:0000h: 01 01 00 00 00 82 00 01 BF 4E 00 00 04 00 FF 01  .....‚..¿N....ÿ.
A00:0010h: 01 50 00 00 04 00 0B 00 74 AB 06 00 81 02 69 1D  .P......t«....i.
A00:0020h: 00 50 00 00 04 00 00 00 F6 86 07 00 50 48 05 00  .P......ö†..PH..
A00:0030h: 2C 00 00 00 00 00 00 00 00 00 00 00 3F 6D 06 61  ,...........?m.a
A00:0040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
A00:0050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
A00:0060h: 30 00 FF 01 00 00 00 00 96 00 00 00 00 00 00 03  0.ÿ.....–.......
A00:0070h: 7B CD B2 F3 05 00 00 00 01 00 00 10 63 2D 5E C7  {Í²ó........c-^Ç
A00:0080h: 6B 05 00 00 00 00 00 00 00 01 00 00 88 B1 16 AF  k...........ˆ±.¯
A00:0090h: E3 B5 02 00 00 00 00 00 00 00 01 00 00 88 B1 16  ãµ...........ˆ±.
A00:00A0h: AF E3 B5 02 00 00 00 00 00 00 00 01 00 00 88 B1  ¯ãµ...........ˆ±
A00:00B0h: 16 AF E3 B5 02 00 00 00 00 00 00 00 01 00 00 88  .¯ãµ...........ˆ
A00:00C0h: B1 16 AF E3 B5 02 00 00 00 00 00 00 00 00 00 00  ±.¯ãµ...........
A00:00D0h: 00 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00  ................
A00:00E0h: 00 00 00 00 00 00 01 00 00 00 00 00 00 00 00 00  ................
A00:00F0h: 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 00  ................
A00:0100h: 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00  ................
A00:0110h: 00 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00  ................
A00:0120h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
A00:0130h: 00 01 01 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
A00:0140h: 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00  ................
A00:0150h: 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00  ................
A00:0160h: 00 00 00 00 00 02 00 00 00 00 00 00 00 00 00 00  ................
A00:0170h: 00 00 00 00 00 00 00 00 00 00 46 2E FF FF 00 00  ..........F.ÿÿ..
A00:0180h: 00 00 00 00 46 2E FF FF 00 00 00 00 46 2E FF FF  ....F.ÿÿ....F.ÿÿ
```

Step 2. Process the output of Step 1 and account for any misaligned page headers accordingly

Now that we can successfully decrypt the files, we need to account for any headers that are misaligned. We saw how to do this earlier by comparing the *PageIndex* field inside the page header. This is the index value that identifies where this particular page needs to be inside the *.ndf* file. Refer to the misaligned header on Figure 17.

Similar to how we found all the encrypted regions, we will proceed in the same manner (excluding the Shannon entropy check this time) of validating each expected page header, and in any instances where there is a case of misalignment, we will create a new file where we correctly insert it at its expected location. We will, of course, copy over all the already valid and existing data into the new file as well. This new file will then be fully decrypted and more importantly, correctly aligned.

```
>> Misaligned header @ 2CD5B59000 | Target: 2CD5B86000
>> Misaligned header @ 2CD5B5B000 | Target: 2CD5B88000
>> Misaligned header @ 2CD5B63000 | Target: 2CD5B90000
>> Misaligned header @ 2CD5B65000 | Target: 2CD5B92000
>> Misaligned header @ 2CD5B67000 | Target: 2CD5B94000
>> Misaligned header @ 2CD5B69000 | Target: 2CD5B96000
>> Misaligned header @ 2CD5B6B000 | Target: 2CD5B98000
>> Misaligned header @ 2CD5B6D000 | Target: 2CD5B9A000
>> Misaligned header @ 2CD5B6F000 | Target: 2CD5B9C000
>> Misaligned header @ 2CD5B8B000 | Target: 2CD5BB8000
>> Misaligned header @ 2CD5B8D000 | Target: 2CD5BBA000
```

```
>> Misaligned header @ 2CD5B8F000 | Target: 2CD5BBC000
>> Misaligned header @ 2CD5DED000 | Target: 2CD5E1A000
>> Misaligned header @ 2CD5DEF000 | Target: 2CD5E1C000
>> Misaligned header @ 2CD5F4D000 | Target: 2CD5F7A000
>> Misaligned header @ 2CD5F4F000 | Target: 2CD5F7C000
>> Misaligned header @ 2CD604D000 | Target: 2CD607A000
>> Misaligned header @ 2CD604F000 | Target: 2CD607C000
>> Misaligned header @ 2CD64FF000 | Target: 2CD652C000
>> Misaligned header @ 2CD6D0B000 | Target: 2CD6D38000
>> Misaligned header @ 2CD6D0D000 | Target: 2CD6D3A000
>> Misaligned header @ 2CD6D0F000 | Target: 2CD6D3C000
>> Misaligned header @ 2CD9919000 | Target: 2CD9946000
>> Misaligned header @ 2CD991B000 | Target: 2CD9948000
>> Misaligned header @ 2CD991D000 | Target: 2CD994A000
>> Misaligned header @ 2CD991F000 | Target: 2CD994C000
>> Misaligned header @ 2CD9BB7000 | Target: 2CD9BE4000
>> Misaligned header @ 2CD9BB9000 | Target: 2CD9BE6000
>> Misaligned header @ 2CD9BBB000 | Target: 2CD9BE8000
>> Misaligned header @ 2CD9BBD000 | Target: 2CD9BEA000
>> Misaligned header @ 2CD9BBF000 | Target: 2CD9BEC000
>> Misaligned header @ 2CD9BC7000 | Target: 2CD9BF4000
>> Misaligned header @ 2CD9BC9000 | Target: 2CD9BF6000
>> Misaligned header @ 2CD9BCB000 | Target: 2CD9BF8000
>> Misaligned header @ 2CD9BCD000 | Target: 2CD9BFA000
>> Misaligned header @ 2CD9BCF000 | Target: 2CD9BFC000
>> Misaligned header @ 2CDAC05000 | Target: 2CDAC32000
>> Misaligned header @ 2CDAC07000 | Target: 2CDAC34000
```

Figure 19. Validating misaligned headers

Before alignment, random data at target location

```
F50F:A000h: 00 00 0A 00 00 00 04 00 59 00 60 00 6D 00 77 00  ........Y.`.m.w.
F50F:A010h: 35 34 2D 4D 49 4C 45 50 46 31 37 30 30 35 34 31  54-MILEPF1700541
F50F:A020h: 37 2D 30 31 50 46 31 37 30 30 35 34 31 37 30 00  7-01PF170054170.
F50F:A030h: 4B 00 00 00 00 00 F9 A6 00 00 00 00 00 00 02 00  K.....ù¦........
F50F:A040h: 00 00 00 00 00 00 01 00 00 A8 5F FB 6F 95 54 98  .........¨_ûo•T˜
F50F:A050h: 00 00 00 00 00 00 00 01 00 00 D4 AF FD B7 4A 2A  ..........Ô¯ý·J*
F50F:A060h: 4C 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00  L...............
F50F:A070h: 00 00 00 00 00 00 00 00 00 0A 00 00 00 04 00 59  ...............Y
F50F:A080h: 00 60 00 6D 00 77 00 35 34 2D 4D 49 4C 45 50 46  .`.m.w.54-MILEPF
F50F:A090h: 31 37 30 30 35 34 31 37 2D 30 32 50 46 31 37 30  17005417-02PF170
F50F:A0A0h: 30 35 34 31 37 30 00 4B 00 00 00 00 00 BE A6 00  054170.K.....¾¦.
F50F:A0B0h: 00 00 00 00 00 01 00 00 00 00 00 00 00 01 00 00  ................
```

After full decryption and alignment set

```
F50F:A000h: 01 01 00 00 00 82 00 01 9A 52 29 00 03 00 4B 00  .....‚..šR)...K.
F50F:A010h: 9B 52 29 00 03 00 42 00 FA 1D 07 00 6E 00 0E 1F  ›R)...B.ú...n...
F50F:A020h: 7D A8 07 00 04 00 00 00 BD 8B 07 00 E8 A6 0F 00  }¨......½‹..è¦..
F50F:A030h: 2A 00 00 00 00 00 00 00 00 00 00 00 E8 9F E8 60  *...........èŸè`
F50F:A040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
F50F:A050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
F50F:A060h: 30 00 4B 00 00 00 00 00 B5 A6 00 00 00 00 00 00  0.K.....µ¦......
F50F:A070h: 01 00 00 00 00 00 00 00 01 00 00 00 31 D6 E2 75  ............1Öâu
F50F:A080h: BC 56 00 00 00 00 00 00 00 01 00 00 80 18 6B F1  ¼V..........€.kñ
F50F:A090h: 3A 5E 2B 00 00 00 00 00 00 00 01 00 00 00 00 00  :^+.............
F50F:A0A0h: 00 00 00 00 00 00 00 00 00 00 00 0A 00 00 00 04  ................
F50F:A0B0h: 00 59 00 60 00 6D 00 77 00 35 34 2D 4D 49 4C 45  .Y.`.m.w.54-MILE

07A87D * pagesz => F50FA000
```

Figure 20. Before and after header alignment

Step 3. Process the output of step 2 and "clean up" the final remnants of leftover data from the misaligned headers

This is great and all and certainly brings us very close to fully realizing our ultimate goal, however, the data still present at the misalignment location is just that: *still present*. We need to do something about this leftover data.

One approach is to place "dummy" headers at these locations, in the hopes they satisfy the loading of the database file. But playing the dummy roles ourselves, we opted to just null these locations. Again, we follow the same pattern of validating the headers, but this time we know there cannot be any more misalignment, so for any leftover data that we encounter we simply null that entire page, making it in effect a null/empty page. Naturally, this loses the data there but is a willing compromise to make since these entries at this state should be far and few in between compared to the enormity of the entire database file.

Combining all three steps outlined above led to the full restoration of the MSSQL database files to the extent that was possible, even reverting their functionality back to normal in the majority of cases. Throughout our analysis, we also had an internal MSSQL subject matter expert (SME) continuously verify our undertakings and found around 7 million inconsistencies with the initial, corrupted files, give or take, down to just single thousand digits after the entirety of our restoration process was completed. Conjuring up SQL queries became possible once again, and although at DART we prefer our KQL, we still carry a fondness for our SQL predecessors.

Conclusion

LockBit 2.0 is one of the leading ransomware strains currently active and has been over the last six months. DART became engaged with a particular customer where we were exposed to our first instance of a Lockbit 2.0 afflicted customer, curiously interested in the plausibility of recovering their corrupted database files. Through the combined efforts of this customer and DART, we were able to successfully satisfy the customer's curiosity and in doing so, outlined the implications "buggy code" can have, and given the right set of circumstances,

can paradoxically become a catalyst to make recovery of destroyed, critical database files a reality, even though it was the original culprit responsible for corrupting them in the first place.

It is typical in incident response engagements for incident responders to identify the full functionality of any collected samples, extract all relevant forensic evidence that can further facilitate the ongoing investigation, all while having proper detections in place. However, we simply cannot overlook our ultimate goal as cybersecurity consultants: that of satisfying the needs of our customers, who as any organization victim to a devastating cyber attack, is seeking the right guidance and support. If those needs are within our means, we have a responsibility to act on them.