# Part 1: LockBit 2.0 ransomware bugs and database recovery attempts

techcommunity.microsoft.com/t5/security-compliance-and-identity/part-1-lockbit-2-0-ransomware-bugs-and-database-recovery/ba-p/3254354

March 11, 2022

*Research by: Nino (Detection and Response Team), Team Torstino (Detection and Response Team)*

***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.*

LockBit 2.0 ransomware has been one of the leading ransomware strains over the last six months. Recently, the FBI issued a flash alert outlining the technical aspects and tactics, techniques, and procedures (TTPs) associated with the LockBit 2.0 affiliate-based ransomware-as-a-service.

Suffice it to say, a plethora of detailed research around this ransomware emerged as a result of version "2.0", which surfaced back in the summer of 2021. All these public reports and technical undertakings, however, fail to mention a critical aspect of this ransomware strain that Microsoft Detection and Response Team (DART) researchers have discovered and is something often not discussed when bringing up the topic of ransomware: “buggy code”, and the unpredictable consequences that it can induce.

This post illustrates a much more direct attempt at ransomware recovery targeting MSSQL databases, where we uncovered and further exploited bugs present in the LockBit 2.0 ransomware code, up to the point where we were able to revert the encryption process for these database files and restore them back to a functioning state. This is often an impossible task to carry out, given that it implies breaking decades of practical research into cryptography-- not simply in theory, but in actual implementation.

This two-part blog series will outline all the steps taken and challenges overcome, in order to restore the damaged database files that served as a critical core of this customer's infrastructure.

## Background

We uncovered critical inconsistencies with the logic of this ransomware upon our first interaction with a LockBit 2.0 afflicted customer, who, incidentally, also purchased the software capable of restoring the destruction the ransomware is known to wreak, known as "the decryptor" aspect of ransomware.

The unfortunate customer was soon to find out that the claims the affiliate-based ransomware distributor made, about paying the ransom resolves to obtaining the decryptor capable of restoring the effects of the encryption, were very dubious in their assertions. Upon attempting to use this purchased decryptor to restore critical database files, the customer was met with very disappointing results and was perplexed as to why the restoration of these database files was not going as expected, and what steps to take next.

At some point, DART became engaged with this customer, obtained access to both the encryptor and decryptor aspects of the ransomware, and with suspicions that "faulty crypto" was at play, analysis commenced.

## Our observations on the encryptor and identifying its anomalies

One of the first things we can do to make our lives easier when suspecting faulty encryption/decryption is to first avoid the urge of digging into any literature regarding the densely obtuse aspects of cryptography, or even more menacingly, *modern cryptography*. Instead, use the power that Sysinternals handy-dandy Procmon provides in monitoring file I/O with the hopes of spotting any kind of anomalies or inconsistencies when either the encryptor or decryptor is running.

Through this monitoring we should get a quick (correct) picture of how the encryption/decryption algorithm is implemented, assuming that it is not doing all of this in memory and indeed going through the I/O manager as is generally the case.

For instance, Figure 1 shows the encryptor in action on a test dummy file we created. It's worth noting, when assuming faulty crypto algorithms are at play, to test on a variety of file sizes to see how/if they pan out differently. We often see a common mistake on larger sized files (at least 4GB or greater), especially in 32-bit encryptors, not understanding that the larger the file size gets, the closer we get, and eventually cross, into signed territory. These mistakes can lead to incorrect checks on file sizes, how the internal file pointer is set, and so on, that can introduce unintended corruption by the encryptor. Something to always keep an eye out for.

Figure 1. Test #1 of the encryptor in action

Test #1: high-level observations

- It increases the file size
- It only encrypts the first 0x1000 bytes from the start of the header (in theory, enough to kill off any header metadata)
- Appends some data at the end of the original file size (0x200 bytes)
- Appends a *.lockbit* extension to the original filename

**Spoiler**: The data that it appends to the end of the encrypted file is the required decryption information that the decryptor utilizes as part of its restoration process. Each file is encrypted with a unique 16-byte initialization vector (IV) and AES256 key. Both are stored, encrypted with a modified *cha-cha dance*, at the end of each individual encrypted file. The decryptor in turn knows how to find this “decryption blob”, extract the unique IV and AES256 key, and then leverage them for the decryption. Other data is stored as well in these blobs, such as the original file size and the AES block size.

Our test #1 from the Procmon output in Figure 1 shows that the encryptor alters the original size of the file it is about to corrupt, so it is only appropriate that it retains this original information somewhere when the decryptor begins to attempt its restoration process. At least this is the theory. In practice, as we’re soon to find out, something quite different has the potential of happening.

Testing the 1GB file was a good start, but let’s try a much larger file and again, observe the behavior of the encryptor through Procmon.

| Process Name | Operation | Path | Result | Detail |
|---|---|---|---|---|
| v2c.exe | CreateFile | C:\iDefense\test.txt | SUCCESS | Desired Access: Read Data/List Directory, Write Data/Add File, Delete, Disposition: Open, Op |
| v2c.exe | QueryStandardInformationFile | C:\iDefense\test.txt | SUCCESS | AllocationSize: 68,719,476,736, EndOfFile: 68,719,476,736, NumberOfLinks: 1, DeletePendin |
| v2c.exe | SetEndOfFileInformationFile | C:\iDefense\test.txt | SUCCESS | EndOfFile: 68,719,477,248 |
| v2c.exe | ReadFile | C:\iDefense\test.txt | SUCCESS | Offset: 0, Length: 4,096, I/O Flags: Non-cached, Priority: Normal |
| v2c.exe | WriteFile | C:\iDefense\test.txt | SUCCESS | Offset: 0, Length: 4,096, I/O Flags: Non-cached, Priority: Normal |
| v2c.exe | WriteFile | C:\iDefense\test.txt | | Offset: 68,719,476,736, Length: 512, I/O Flags: Non-cached, Priority: Normal |
| v2c.exe | WriteFile | C:\iDefense\test.txt | SUCCESS | Offset: 16,384, Length: 65,536, I/O Flags: Non-cached, Paging I/O, Synchronous Paging I/O, F |
| v2c.exe | WriteFile | C:\iDefense\test.txt | SUCCESS | Offset: 81,920, Length: 65,536, I/O Flags: Non-cached, Paging I/O, Synchronous Paging I/O, F |
| v2c.exe | WriteFile | C:\iDefense\test.txt | SUCCESS | Offset: 147,456, Length: 65,536, I/O Flags: Non-cached, Paging I/O, Synchronous Paging I/O, |
| v2c.exe | WriteFile | C:\iDefense\test.txt | SUCCESS | Offset: 212,992, Length: 65,536, I/O Flags: Non-cached, Paging I/O, Synchronous Paging I/O, |
| v2c.exe | WriteFile | C:\iDefense\test.txt | SUCCESS | Offset: 278,528, Length: 65,536, I/O Flags: Non-cached, Paging I/O, Synchronous Paging I/O, |
| v2c.exe | WriteFile | C:\iDefense\test.txt | SUCCESS | Offset: 344,064, Length: 65,536, I/O Flags: Non-cached, Paging I/O, Synchronous Paging I/O, |

Figure 2. Test #2 for encryptor in action

Test #2: high-level observations:

- Starts off like our first test but ends drastically different
- Procmon curiously does not generate a *Result* for the *WriteFile* operation when appending the decryption blob
- It seems to further encrypt, at 65,536-byte intervals, more data

Having some clear differences from our first test run, the second one intrigues us enough to continue digging deeper with the suspicion that something is seriously not right here. It gets even more intriguing when we try to view the call stack for the *WriteFile* operations that follow the instance where Procmon was unable to tell us the *Result* of appending the decryption blob.

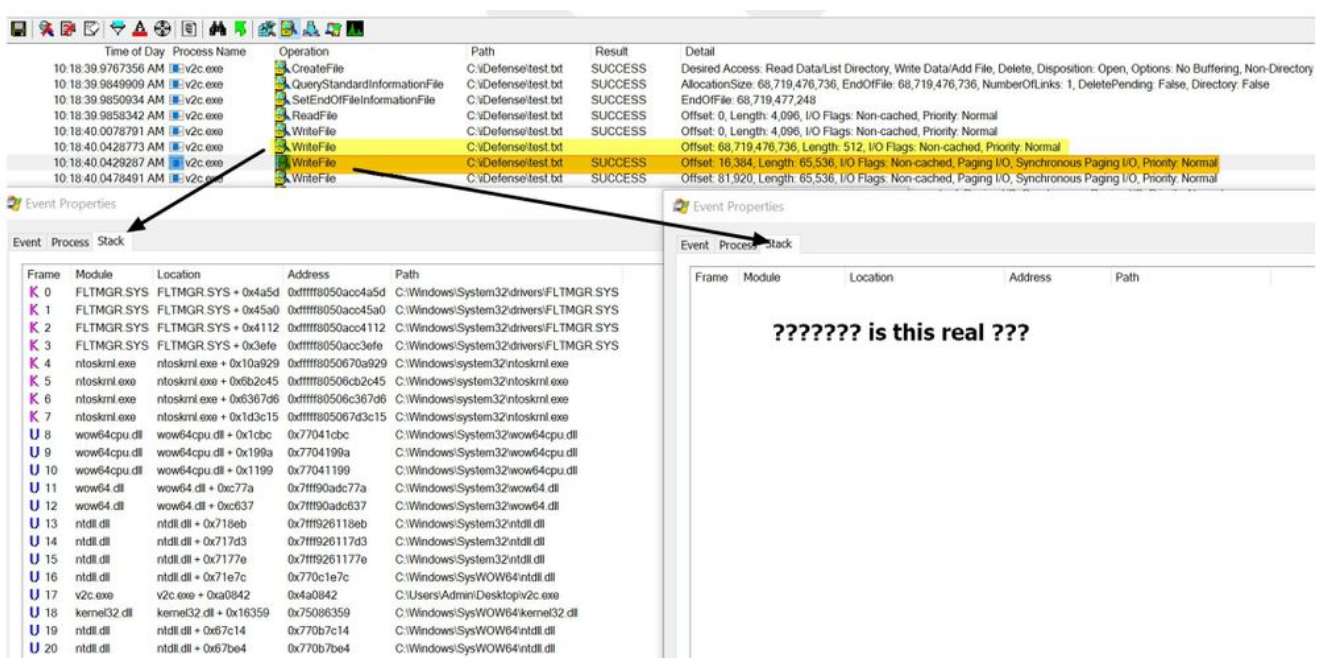

Figure 3. Viewing the call stack for the WriteFile operations

Every *WriteFile* operation following the empty *Result* in the yellow highlighted row looks like the *Event Properties* box on the right: empty. This is very strange indeed and requires a deeper introspection than Procmon can give us. Before departing from the almighty

Procmon, it continues to show its worth by providing us with a valuable vantage point of where to begin looking at: the call stack. We can see that at offset ***+0xA0842*** is where we presumably never return from.

Now feels like the right time to introduce our favorite toolset for any deep troubleshooting into the picture: Time Travel Debugging (TTD)

## What exactly is the issue?

Prior to introducing the TTD framework into the picture, we will first load the encryptor into IDA Pro and go to that offset identified by Procmon to observe the code at that location. Doing so, we can see that we are at the return address of what is a call to *ntdll!NtWriteFile.* Depending on what we can further spot in the disassembly or decompilation, the following plan is to re-run the encryptor again, but this time under the control of *TTTracer* to generate some runtime data that we can work against.

```
.text:004A0825   lea    eax, [ecx+lb_crypt_t.byte_offset]
.text:004A0828   push   eax
.text:004A0829   push   [ecx+lb_crypt_t.encrypted_size]
.text:004A082C   lea    eax, [ecx+lb_crypt_t.io_status]
.text:004A082F   push   [ecx+lb_crypt_t.buffer]
.text:004A0832   push   eax
.text:004A0833   push   ecx
.text:004A0834   push   0
.text:004A0836   push   0
.text:004A0838   push   [esi+lb_encrypt_file_t.file_handle]
.text:004A083B   call   lb::resolve_ntwritefile       ; append tail
.text:004A0840   call   eax
.text:004A0842   test   eax, eax
.text:004A0844   jns    __resolve_nt_remove_io        ; jumptable 004A071F default case
.text:004A084A   or     eax, 0FFFFFFFFh
```

Figure 4. Code responsible for writing the encrypted contents back to disk

Let's also show the cleanup decompilation of this piece of code as well, to observe at a higher level.

```
p_byte_offset = &CompletionValue->byte_offset;
filesz = CompletionValue->encrypted_size;
buf = CompletionValue->buffer;
pIoStatus = &CompletionValue->io_status;
hfile = CompletionKey->file_handle;
WriteFile = lb::resolve_ntwritefile();
if ( WriteFile(hfile, 0, 0, CompletionValue, pIoStatus, buf, filesz, p_byte_offset, 0) < 0
  && !_InterlockedExchangeAdd(&CompletionKey->field_48, 0xFFFFFFFF) )
```

Figure 5. Decompilation of Figure 4

As shown in both Figure 4 and 5, we can spot that something is off here; the **NTSTATUS** return value for the write file is not handled correctly. In fact, it's flat-out wrong. One way that we can demonstrate the consequence of this improper handling of the write file operation is to ask whether the encryptor operates asynchronously. The reasons for introducing this in our inquiry will be explained shortly.

But if we do dig a bit into the binary inside IDA, we can confirm the asynchrony of the encryptor, implemented through I/O completion ports. The actual file encryption is done via a callback routine executed as a thread, and very interestingly for the debugging enthusiasts, *hidden* threads.

```
NumberOfProcessors = NtCurrentPeb()->NumberOfProcessors;
g_CPU_COUNT_0 = NumberOfProcessors;
IoCompletion = lb::resolve_ZwCreateIoCompletion();
if ( IoCompletion(&g_IOCP_HANDLE_0, IO_COMPLETION_ALL_ACCESS, 0, NumberOfProcessors) >= 0 )
{
  dword_4FB9EC = lb::mem_alloc((4 * g_CPU_COUNT_0));
  if ( dword_4FB9EC )
  {
    for ( i = 0; i < g_CPU_COUNT_0; ++i )
      *(dword_4FB9EC + 4 * i) = lb::api::create_hidden_thread(lb::crypt::init_cleanup, g_IOCP_HANDLE_0);
    goto LABEL_5;
  }
hthread = CreateThread(0, 0, lpStartAddress, lpParameter, 0, lpThreadId);
if ( hthread != INVALID_HANDLE_VALUE )
{
  NtSetInformationThread = lb::resolve_NtSetInformationThread();
  NtSetInformationThread(hthread, ThreadHideFromDebugger, 0, 0);
}
return hthread;
```

Figure 6. Encryptor multi-threading initialization and using hidden threads that carry out the encryption

What this call to **NtSetInformationThread** does is set the *HideFromDebugger* flag inside the internal, executive thread structure, which guarantees that the debugger will never receive any debug events for this thread, effectively missing the controllable execution of these threads. Something to be aware of when attempting to debug this encryptor in the traditional manner. Since we plan to use TTTracer, these anti-debug shenanigans are moot, and we can ignore them completely.

This is great and all, but what exactly is the issue here with the NTSTATUS value? First, LockBit 2.0 devs mistakenly assume all unsuccessful NTSTATUS values are signed. For instance, the following ones are very relevant to the encryptor given its asynchronous behavior and are clearly not negative numbers.

| | |
|---|---|
| 0x000000C0<br>STATUS_USER_APC | A user-mode APC was delivered before the given Interval expired. |
| 0x00000101<br>STATUS_ALERTED | The delay completed because the thread was alerted. |
| 0x00000102<br>STATUS_TIMEOUT | The given Timeout interval expired. |
| 0x00000103<br>STATUS_PENDING | The operation that was requested is pending completion. |

Figure 7. NTSTATUS values

Second, and more importantly, they entirely neglect the handling of pending I/O operations: **STATUS_PENDING**. And given the asynchronous nature of I/O on Windows, this in theory could be every file I/O operation. Further, given that the encryption is carried out asynchronously as well through I/O completion ports, *ntdll!NtWriteFile* **can and will return STATUS_PENDING**, which the caller must properly account for. How does one account for it? Patience. (See WaitForSingleObject and ZwWaitForSingleObject)

Not doing so will lead to unpredictable and potentially destructive behavior as LockBit 2.0 is mistakenly assuming success after each write operation when the return value is not signed. When multiple threads are at play, which they will be, you now create a situation that can result in all these worker threads writing at unpredictable intervals. Seems like a minor ordeal, but because of this mishandling, the entire stability of the encryptor is now in question. These effects naturally spill over to the decryptor as well.

IO_STATUS_BLOCK

```
NtWriteFile(
IN HANDLE           FileHandle,
IN HANDLE           Event OPTIONAL,
IN PIO_APC_ROUTINE  ApcRoutine OPTIONAL,
IN PVOID ApcContext OPTIONAL,
OUTPIO_STATUS_BLOCK IoStatusBlock,
IN PVOID             Buffer,
IN ULONG             Length,
IN PLARGE_INTEGER ByteOffset OPTIONAL,
IN PULONG Key        OPTIONAL);
);
```

*The operating system implements support routines that write IO_STATUS_BLOCK values to caller-supplied output buffers. For example, see ZwOpenFile or NtOpenFile. These routines return status codes that might not match the status codes in the IO_STATUS_BLOCK* *structures.* ***If one of these routines returns STATUS_PENDING, the caller should wait for the I/O operation to complete, and then check the status code in the IO_STATUS_BLOCK structure*** *to determine the final status of the operation.*

*If the routine returns a status code other than STATUS_PENDING, the caller should rely on this status code instead of the status code in the IO_STATUS_BLOCK structure.*

## About the broken decryptor (and decrypting files that it couldn't)

Having now identified at least one critical flaw that can result in faulty crypto, let's shift our attention to the decryption process itself, because our primary goal is to confirm, and then hopefully implement, a capacity to do what the purchased decryptor was supposed to do.

From the customer, we were given several MSSQL encrypted database files which had the potential of being correctly decrypted. The reason that we can make such a claim is that the required decryption information (recall our earlier Procmon adventures) was still intact somewhere in the file. Not where it's *supposed* to be, but it's there, nonetheless. This misplacement, a direct result of the improper handling of the write file operation outlined above, is what causes the decryptor to miss retrieving this blob of data. This mishandling can even unwittingly truncate or expand the original file size. Simply having the decryption blob information present in the encrypted binary does not really mean anything at this stage of what we're trying to accomplish.

One of the first things that we tried to get the decryptor up and running accurately, was to remove all the data that follows the decryption blob in the encrypted database file, giving it the appearance of being "correctly" appended, as it was originally intended to be. We then ran the decryptor against it (under TTTracer) to see what would happen. We failed to decrypt the file with this approach but with the resulting TTD trace, we have a window to peek into and identify the flaws in our wishful approach.

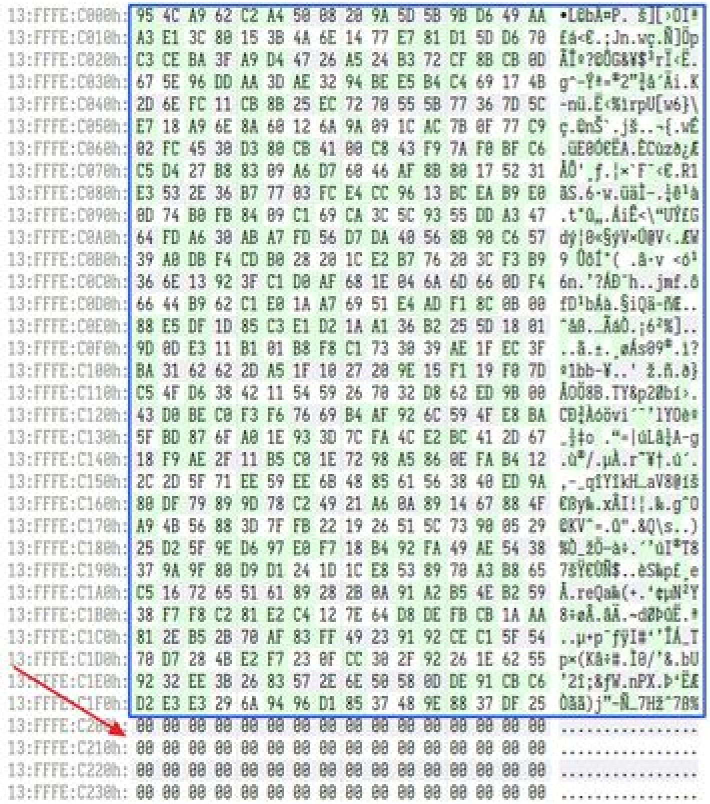

Figure 8. The decryption blob was found, but it’s not at the end/tail of the file as it’s supposed to be

Going through the generated trace file, we were able to identify that the decryptor does indeed find the decryption blob correctly now and furthermore, is able to successfully decrypt it to acquire the necessary IV and AES key for decryption. However, the file still does not get decrypted. Digging deeper, we identified the issue being in how it tries to compare two LARGE_INTEGERs, that of the incoming, encrypted file size and the AES block size stored in the decryption blob data that it assumed it appended correctly.

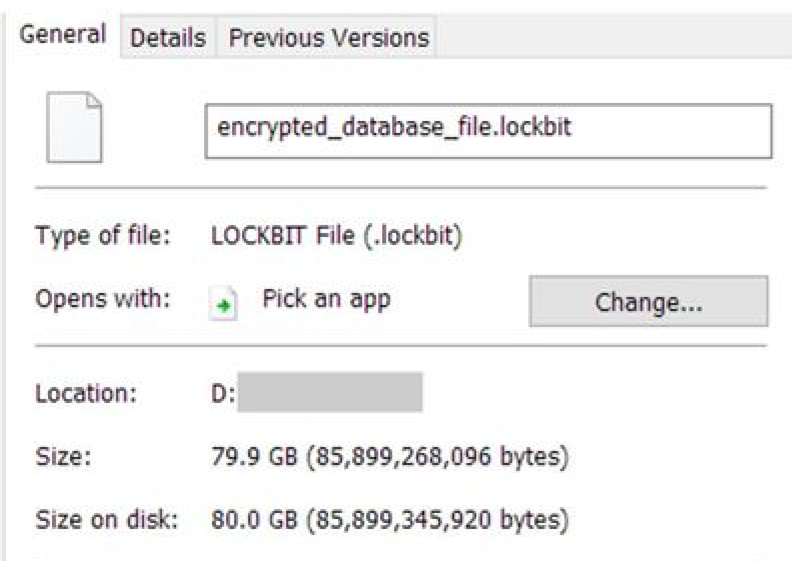


Figure 9. File size and the encrypted database file we’re working against

```
// disassembly responsible for initiating this sequence, by storing the incoming file
size
.text:00428721 mov esi, dword ptr [eax+lb_encrypt_file_t.og_filesz] ; fetch the
LowerPart of the file size
.text:00428724 mov eax, [eax+lb_encrypt_file_t.og_filesz.anonymous_0.HighPart] ;
fetch the HighPart of the file size
.text:00428727 mov [esp+1Ch], eax ; store the HighPart of the file size
.text:0042872B lea eax, [esp+3E8h+var_268]
.text:00428732 push eax
.text:00428733 mov [esp+18h], esi ; save the LowerPart of the file size

// in the TTD trace, looking at the incoming file size being stored as a
LARGE_INTEGER
00428724 8b4024 mov eax,dword ptr [eax+24h]
ds:002b:1c9e0024=00000013
0:014> dd @eax
1c9e0000 00000000 00000000 00000000 00000000
1c9e0010 00000000 00000000 00000000 00000000
1c9e0020 fffec200 00000013 00000000 00000001

// size of the incoming file
0:014> dt ntdll!_LARGE_INTEGER 1c9e0020 QuadPart
0x00000013`fffec200
+0x000 QuadPart : 0n85899264512

// code that does the check after the offset has been calculated from the decryption
blob
.text:004288E6 mov eax, [esi+lb_encrypt_file_t.byte_offset.anonymous_0.HighPart]
.text:004288E9 add edx, ecx
.text:004288EB adc edi, eax
.text:004288ED cmp [esp+1Ch], edx ; now check the LowerPart
.text:004288F1 jnz __size_check_fail_cleanup
.text:004288F7 cmp [esp+18h], edi ; now check the HigherPart
.text:004288FB jnz __size_check_fail_cleanup
__success_go_for_decryption_of_encrypted_content

// go to the location where the check and “bug” is at
0:014> dx @$calls(0x4288ED).First().TimeStart.SeekTo()
Time Travel Position: 1CC3E8:F20 [Unindexed] Index
0:014> u . l4
decryptor+0x288ed:
004288ed cmpdword ptr [esp+1Ch],edx ; compare against LowerPart
004288f1 jne __size_check_fail_cleanup ; they have to match, otherwise decryption is
skipped
004288f7 cmp dword ptr [esp+18h],edi ; compare against the HighPart
004288fb jne __size_check_fail_cleanup ; they have to match, otherwise decryption is
skipped
0:014> r edx
edx=00000200 ; AES block size calculated out of the data inside the decryption blob
0:014> dd @esp+1c l1
1a73fb9c fffec200 ; LowPart of incoming file size, failing when being compared to the
size of the decryption blob

 0:014> r edi
edi=00000014 ; very revealing, this tells us where the decryption blob should
```

```
actually be (what the HighPart should be)
0:014> dd @esp+18 l1
1a73fba4 00000013 ; HighPart, we see our cutting off all the data after the
decryption blob breaks the logic here
```

Based on the TTD trace, simply cutting off all the data that follows the decryption blob won't work either, but we can spot what the issue is and even where the decryption blob is originally supposed to be: minimum at offset **0x1400000000** in the file. The high part of the large integer for the incoming file is at offset **0x1300000000**, but it fails when compared to the original size that was calculated out of the decryption blob: **0x1400000000**. But even before that, the comparison of **0xfffec200** and **0x200** also fails, since it's expecting to have correctly calculated the AES block size, which it did not.

Realizing this, we decided to "push" the decryption blob up to its proper offset, and then again cut off all the data that followed it, to recreate the encrypted file once more into what should be its originally intended structure. Once done, we re-run it through the decryptor and excitedly await the results.

Figure 10. Correctly aligning the decryption blob before we re-run the decryptor against it

Upon running the decryptor this time around, we successfully decrypted the file!

```
decryptor_pp+0x288ed:
004288ed cmp     dword ptr [esp+0Ch],edx ss:002b:0271fb9c=00000200
0:007> r edx
edx=00000200// edx, as expected is 0x200
0:007> dd @esp+c l1
0271fb9c  00000200       // aes block size has correctly been calculated this time
0:007> t                 // step into, to validate the jne
decryptor_pp+0x288f1:
004288f1 jne     decryptor_pp+0x28c0a (00428c0a)         [br=0]
0:007> r zf
zf=1
0:007> t                 // step into to compare the next check for the HighPart
decryptor_pp+0x288f7:
004288f7 397c2414        cmp     dword ptr [esp+14h],edi ss:002b:0271fba4=00000014
0:007> dd @esp+14 l1
0271fba4  00000014       // we see that they're the same, and the decryptor works as
expected
0:007> r edi
edi=00000014
0:007> t
0:007> r zf
zf=1
```

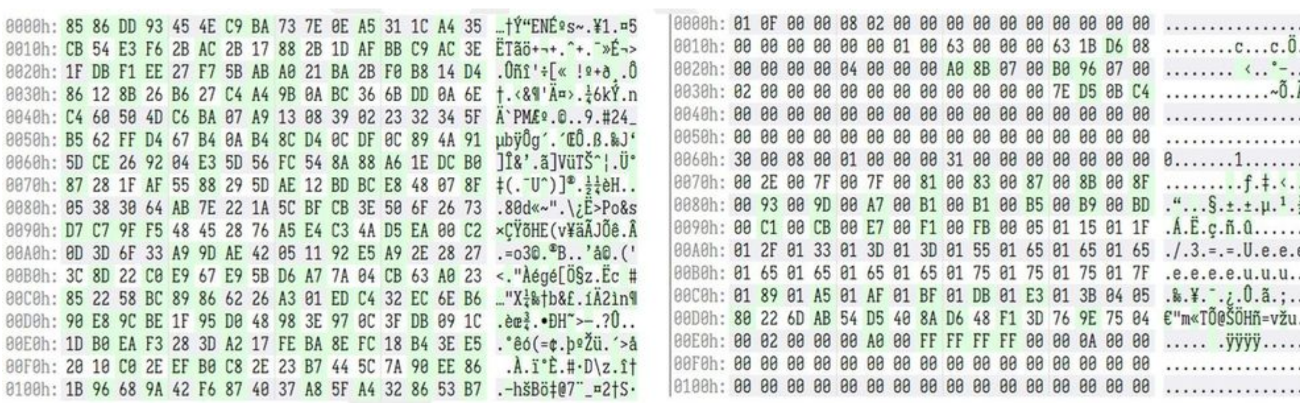

Figure 11. (L) Encrypted file; (R) Successfully decrypted file

While this has the deceptive appearance of some kind of success, we must remain ever cognizant of the fatal bug that's inside the encryptor. The critical flaw by these ransomware developers in misunderstanding how NTSTATUS values work, and the consequences they can have for naïve thread synchronization. Given that we don't want to be unwitting victims of naivety ourselves, we quickly realized that the immensity of the problem was just now slowly starting to reveal itself.

## Coming up in Part 2

In the second part of this series, we will shift our focus to outlining the issues that the decryptor poses, uncover the file structure of the database files that we're dealing with, throw in a little bit of crypto magic into play, and take the necessary steps to achieve our ultimate goal: the successful restoration of all encrypted database files.