[Case Study: Latrodectus] Analyzing and Implementing String Decryption Algorithms

0x0d4y.blog/case-study-analyzing-and-implementing-string-decryption-algorithms-latrodectus/

May 9, 2024

This article has a slightly different objective than the last ones I published, it is not about an analysis of specific malware.

Today's article is about a case study of the **Latrodectus** string decryption algorithm (analyzed in the **previous research**). The objective is to study how to identify a string decryption algorithm when reverse engineering a malware, and how we can implement it in *Python* to statically decrypt them. Specifically, we will cover how to identify whether malware is using a custom decryption algorithm to obfuscate strings.

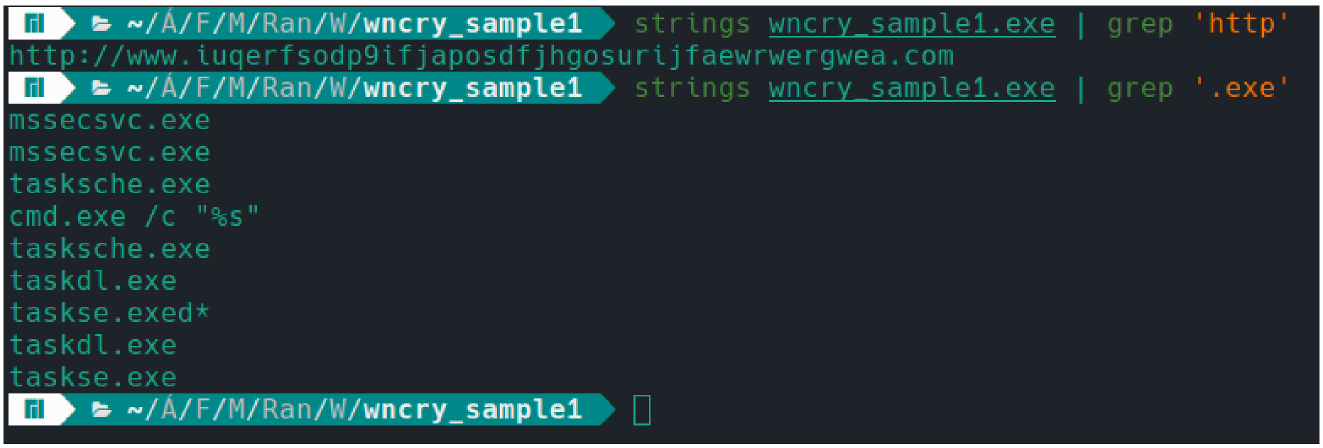
So, let's go!

Why Do Adversaries Encrypt Strings?

It may seem obvious, but it's good to clarify why adversaries encrypt strings to use in their Malware.

The short and thick answer is, to achieve the technical objective of **Obfuscation**! The implementation of encryption to obfuscate strings allows adversaries to hide strings that reveal the objective of the actions that the malware will perform. This technique is registered with MITRE ATT&CK as Obfuscated Files or Information: Encrypted/Encoded File [T1027.013].

Let’s use the example of *WannaCry*, which does not implement the string encryption technique. Below, we can see the simple use of the *strings* command, present in every *Linux* system.



As you can see in the image above, without implementing this obfuscation technique, the malware is more vulnerable to detections by security products, and our analysis is simpler :,)
.

Now let’s do the same test on a *Latrodectus* sample.

```
~/A/F/M/L/String Decryptor  strings d1e2e287c96c290e161c553d99a115e7d72f83f23c850621169a27cca936f51b.bin | grep 'http'
~/A/F/M/L/String Decryptor  strings d1e2e287c96c290e161c553d99a115e7d72f83f23c850621169a27cca936f51b.bin | grep '.exe'
~/A/F/M/L/String Decryptor  strings d1e2e287c96c290e161c553d99a115e7d72f83f23c850621169a27cca936f51b.bin | tail -n 35
D$0H
T$8H
D$(H
D$ H
T$0H
Mb=Lk
.text$mn
.idata$5
.rdata
.rdata$zzzdbg
.xdata
.edata
.idata$2
.idata$3
.idata$4
.idata$6
.data
.bss
.pdata
UpdaterTag.dll
extra
follower
scub
CreateMutexW
PeekNamedPipe
GetLastError
KERNEL32.dll
MessageBoxA
MessageBeep
USER32.dll
&a(F*Y,@.N0E2    4y6^8J:0<=>
l q"Q$J&C(\*H,Y.
0v2V4A6
8}:R<N>0@-B"D<F H(J&L(NoP-RsT3V>X7Z?\.^+`
bCdJf1hIjDl/n0p^r0t0v
~/A/F/M/L/String Decryptor
```

And as we can see above, it was not possible to detect with the strings command any inconsistencies in *URLs* or *binaries*, as we analyzed *Latrodectus* previously, we know that it communicates with two *C2* servers, so this means one thing, *Latrodectus* implements a encryption to obfuscate strings, and decrypts at runtime!

Now that we understand why adversaries implement this obfuscation technique, let's understand how we can identify the use of a decryption algorithm in malware.

How to Identify the Implementation of a Decryption Algorithm?

Firstly, there is no single correct answer to this question, secondly, here comes the famous 'it depends'!

Adversaries will always seek to implement custom encryption algorithms to obfuscate strings and some payloads. This is because using known algorithms, whether through Windows APIs or by manually implementing a known algorithm, can reduce the adversary's chances of passing through *detections* and slowing down our rate of analysis of their sample. This is because the algorithm is already known to us, therefore, it is easier to identify constants or the flow of a given algorithm.

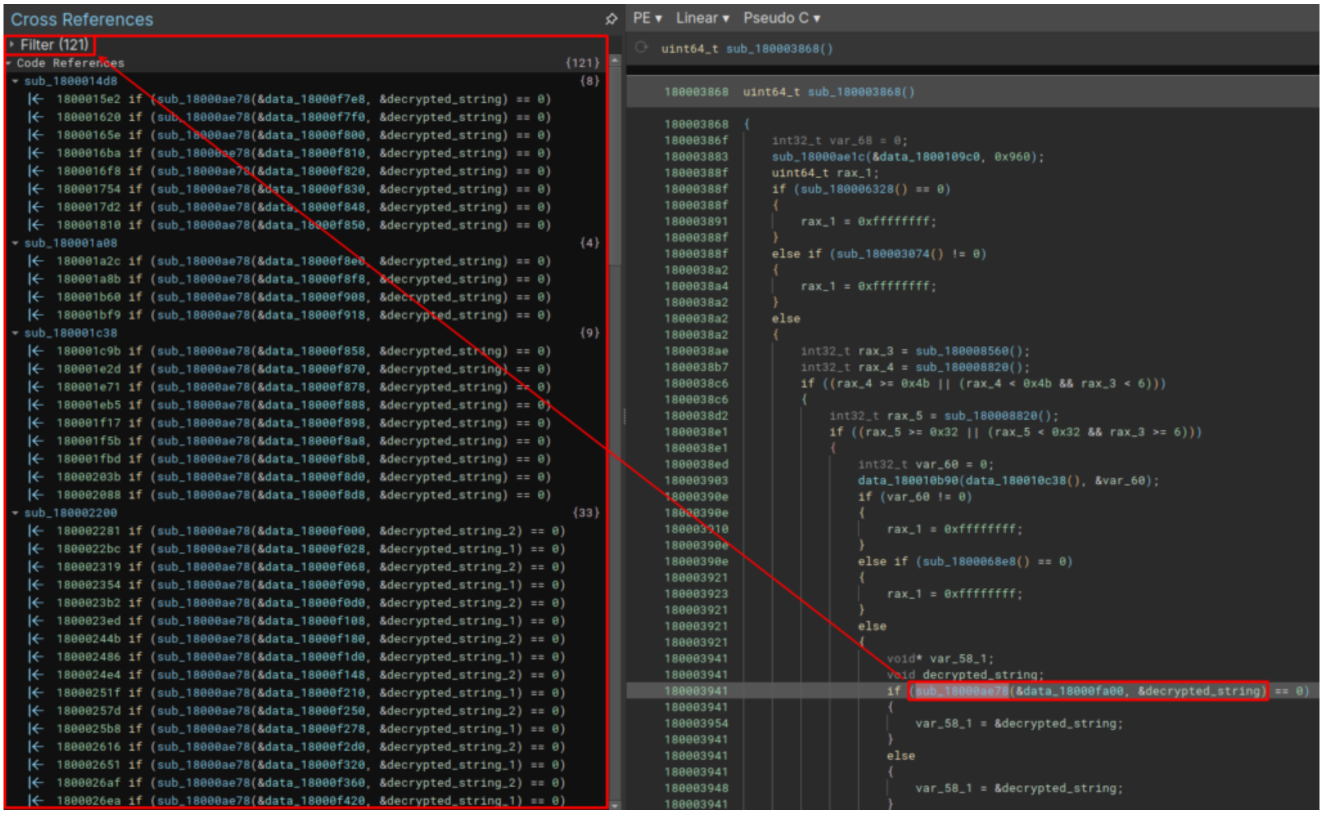
But when the adversary goes down the path of implementing its own algorithm, it runs the risk of implementing something simpler than what already exists. This is because the adversary certainly does not want to disrupt the functioning of the malware. Another risk that the adversary may run is that, once identified by a researcher, *detections* to monitor the presence of a particular malware family will come into existence, in addition to automated extractors using scripts. Therefore, these custom algorithms are short-lived.

But how can we identify that a malware has a string decryption routine? Let’s go.

As I said earlier, there is no single method of identification. So here are some tips, and then we will analyze how the behavior is observed in *Latrodectus*.

- A single function is called dozens or hundreds of times during code execution (the amount will depend on the malware);
- The function will probably be receiving as an argument an offset that points to a block of data, probably encrypted.
- When looking at the function execution flow graph, you will probably find one or more loops that perform operations on the data. The likely operation you will encounter will be the **XOR** operation, which will have the *encrypted data* and the *XOR key* as protagonists of this operation.
- Another tip I can give is to identify in your favorite *Disassembler* if there are several blocks of data that are called by the same function during code execution.

Below we can see that in Latrodecuts it is possible to identify some of these patterns that I mentioned above.



Function **sub_18000ae78** is called **121 times** during *Latrodectus* execution. Another pattern that we can detect is that this function receives a set of data as an argument, and stores the result of the function's manipulation in a buffer.

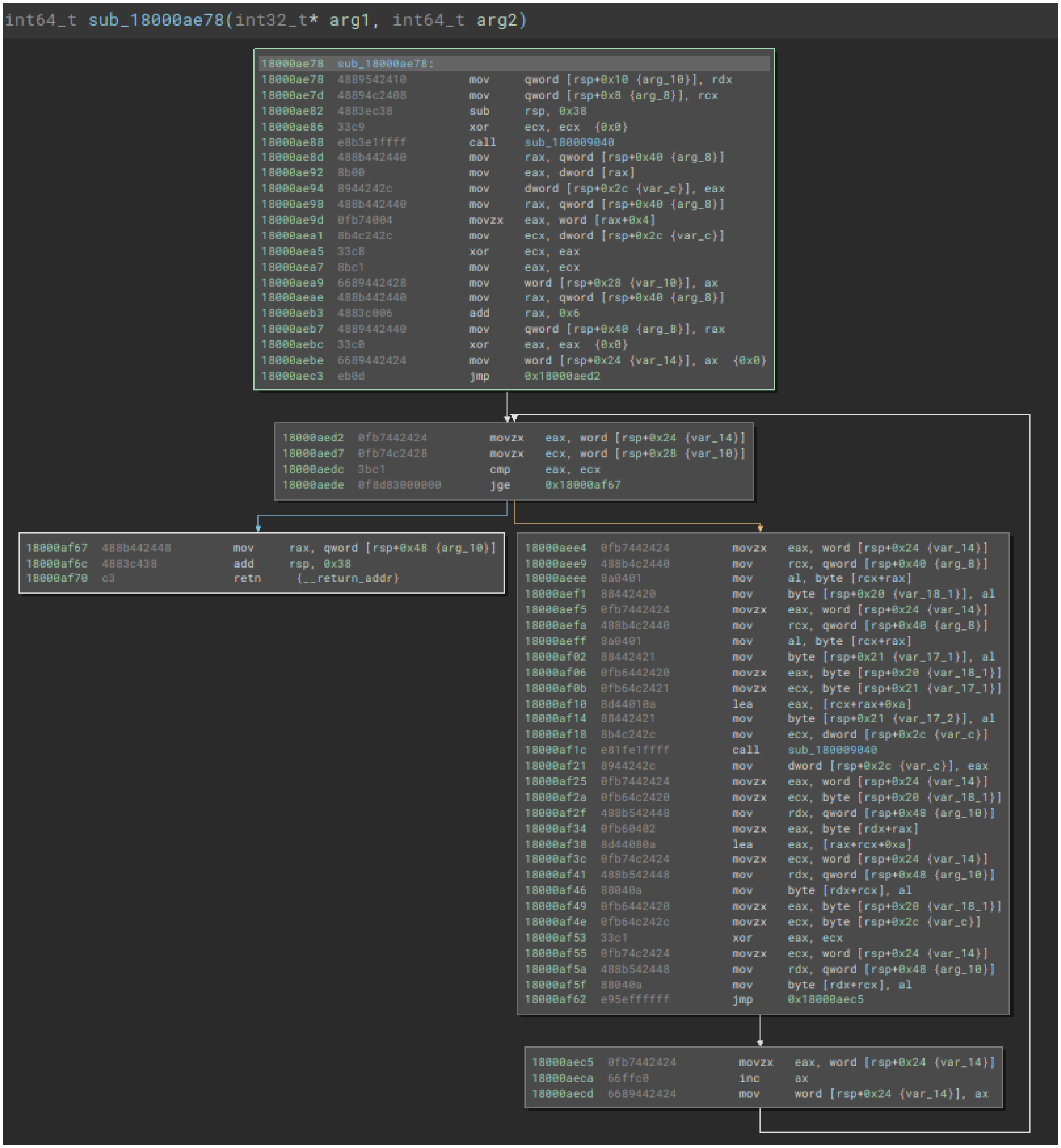
Below we can observe the encrypted data block that is passed as an argument (**data_18000fa00**), in addition to being able to observe the other encrypted data blocks.

```
PE ▾   Linear ▾   Pseudo C ▾

    0x18000f9b0 .data {0x18000f000-0x180011320} Writable data

18000f9b0  7e d0 dd ad 7c d0 7f 80                          ~...|...
18000f9b8  data_18000f9b8:
18000f9b8                          7e d0 dd ad 64 d0 0d 80          ~...d...
18000f9c0  f4 82 ed 84 e1 86 eb 88-e5 8a b8 8c bf 8e a1 90  ................
18000f9d0  f4 92 eb 94 f0 96 97 98                          ........
18000f9d8  data_18000f9d8:
18000f9d8                          7e d0 dd ad 66 d0 5d 80          ~...f.].
18000f9e0  a4 82 f0 84 a7 86 ab 88-a9 8a ae 8c fe 8e af 90  ................
18000f9f0  b4 92 e0 94 95 96 00 00                          ........
18000f9f8  data_18000f9f8:
18000f9f8                          20 00 00 00 00 00 00 00           .......
18000fa00  data_18000fa00:
18000fa00  7e d0 dd ad 6e d0 0d 80-f4 82 ed 84 eb 86 f2 88  ~...n...........
18000fa10  e7 8a ec 8c 8d 8e 00 00                          ........
18000fa18  data_18000fa18:
18000fa18                          7e d0 dd ad 6e d0 45 80          ~...n.E.
18000fa20  f6 82 f7 84 e3 86 e5 88-eb 8a fa 8c 8d 8e 00 00  ................
18000fa30  data_18000fa30:
18000fa30  2c 00 00 00 00 00 00 00                          ,.......
18000fa38  data_18000fa38:
18000fa38                          7e d0 dd ad 78 d0 5a 80          ~...x.Z.
18000fa40  e5 82 83 84 00 00 00 00                          ........
18000fa48  data_18000fa48:
18000fa48                          7e d0 dd ad 74 d0 5a 80          ~...t.Z.
18000fa50  f2 82 a6 84 f6 86 87 88                          ........
18000fa58  data_18000fa58:
18000fa58                          7e d0 dd ad 73 d0 19 e9          ~...s...
18000fa60  ed e7 f0 ab e7 f6 a9 ec-e8 fe 8b 00 00 00 00 00  ................
18000fa70  data_18000fa70:
18000fa70  7e d0 dd ad 6a d0 5a 80-f2 82 df 84 a0 86 e3 88  ~...j.Z.........
18000fa80  a7 8a ef 8c e1 8e e3 90-91 92 00 00 00 00 00 00  ................
18000fa90  data_18000fa90:
18000fa90  7e d0 dd ad 70 d0 5a 80-e5 82 ad 84 e1 86 e6 88  ~...p.Z.........
18000faa0  fd 8a 8b 8c 00 00 00 00                          ........
18000faa8  data_18000faa8:
18000faa8                          7e d0 dd ad 72 d0 5a 80          ~...r.Z.
18000fab0  f2 82 df 84 a0 86 f4 88-89 8a 00 00 00 00 00 00  ................
18000fac0  data_18000fac0:
18000fac0  7e d0 dd ad 58 d0 16 80-ef 82 ea 84 f1 86 a7 88  ~...X...........
18000fad0  a4 8a f1 8c f7 8e f5 90-eb 92 ae 94 b7 96 b2 98  ................
18000fae0  ea 9a c7 9c b8 9e ec a0-83 a2 a3 a4 00 00 00 00  ................
18000faf0  data_18000faf0:
18000faf0  7e d0 dd ad 78 d0 19 f2-ee ec f7 84 00 00 00 00  ~...x...........
18000fb00  data_18000fb00:
18000fb00  7e d0 dd ad 76 d0 50 e6-e8 ee e6 f7 aa 86 00 00  ~...v.P.........
18000fb10  data_18000fb10:
```

If we look at the execution flow in graphical mode, we will also detect another pattern, a loop that manipulates data through **XOR** operations.

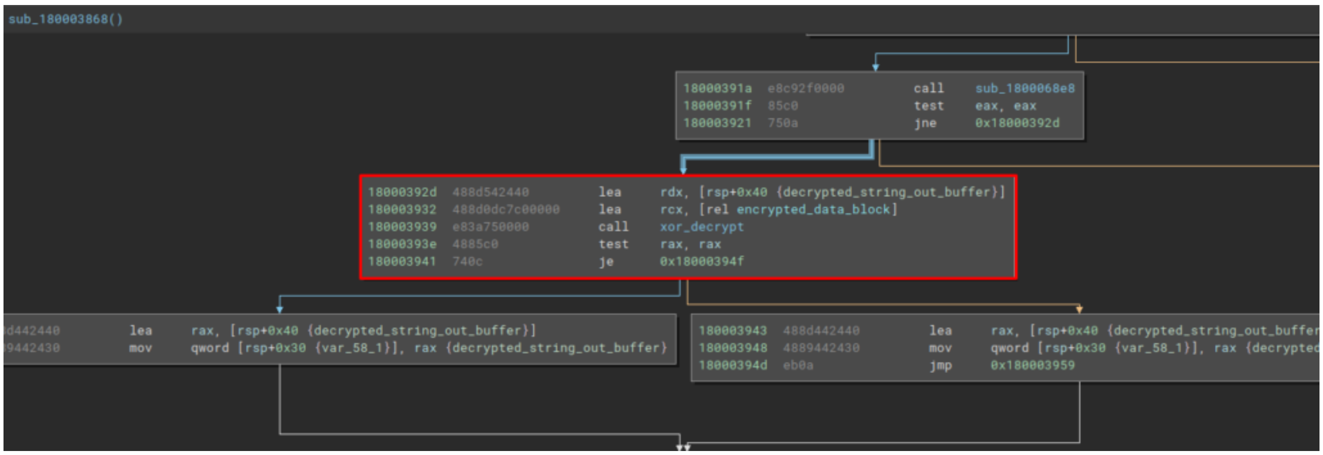


Now that we have been able to identify the string decryption function, let's analyze how it works statically and dynamically.

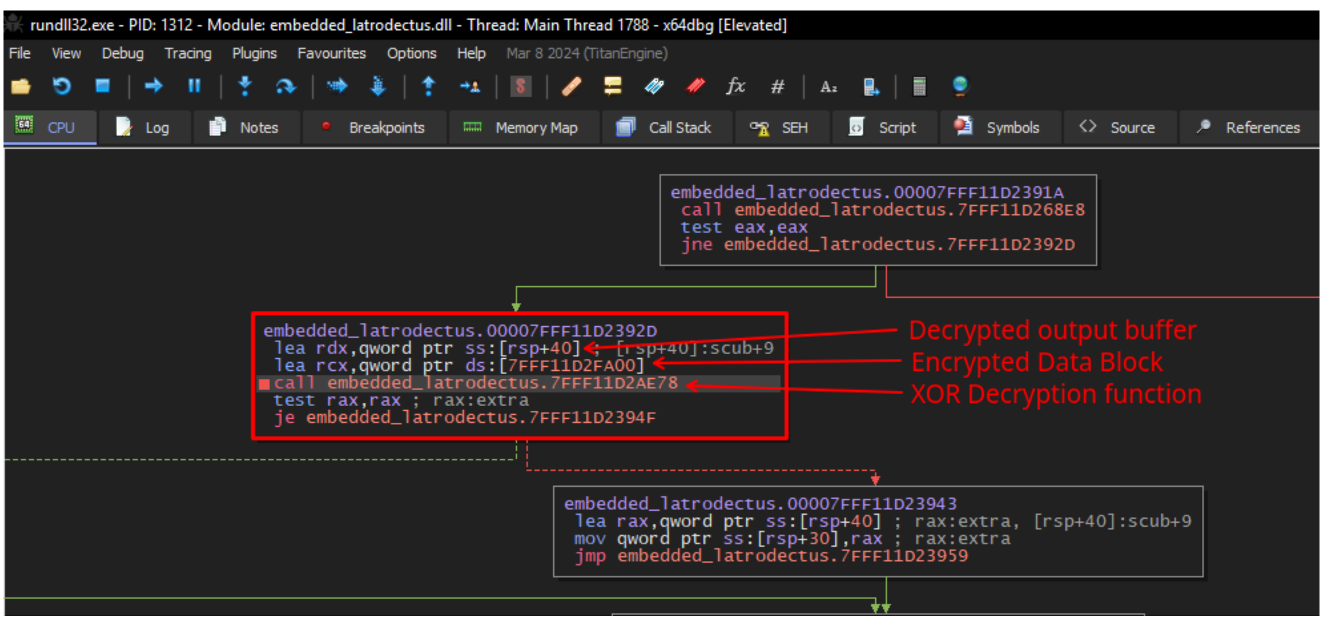
Analyzing Latrodectus Decryption Algorithm

Here, we begin our hands-on adventure. First, when we are going to do our dynamic analysis as a complement to the static one, we need to locate the exact decryption function in the debugger, so as not to get lost. In the debugger we do not have the Decompiler crutch, so it is important that during dynamic analysis using a debugger, you have the disassembler/decompiler open.

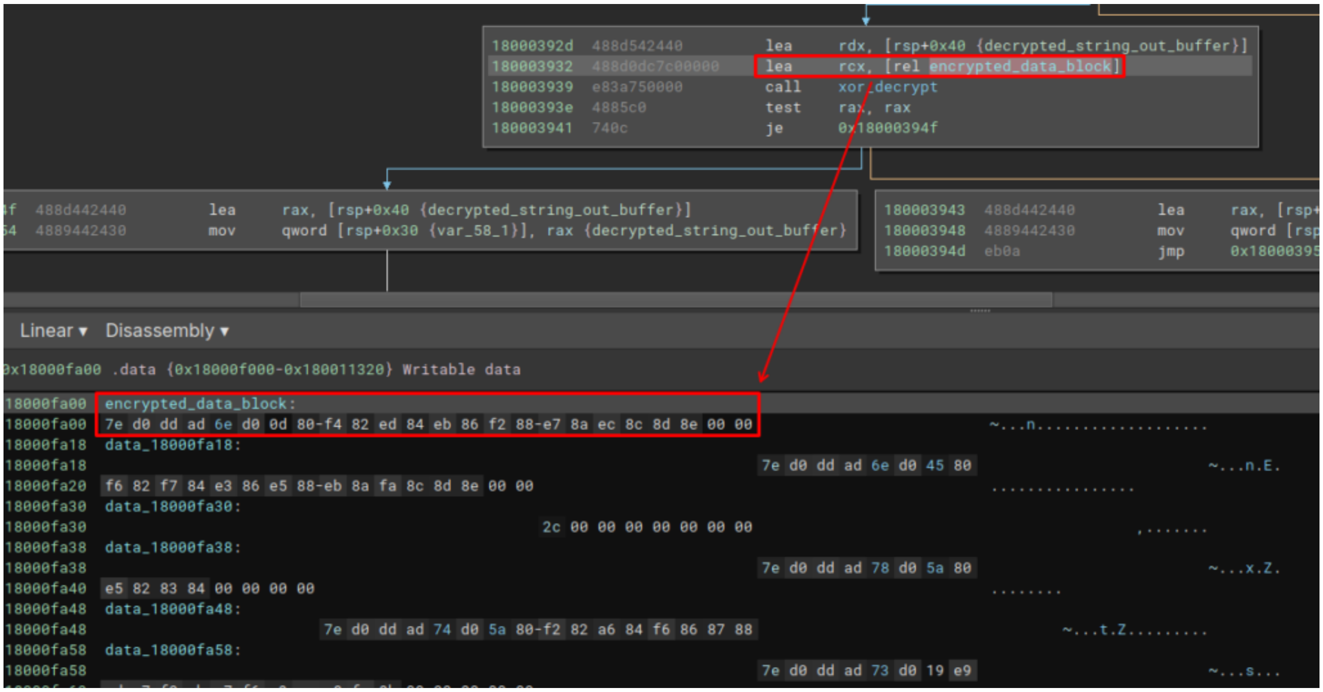
In the decompiler, we can see below the exact moment when our decryption function is called for the first time in the code.



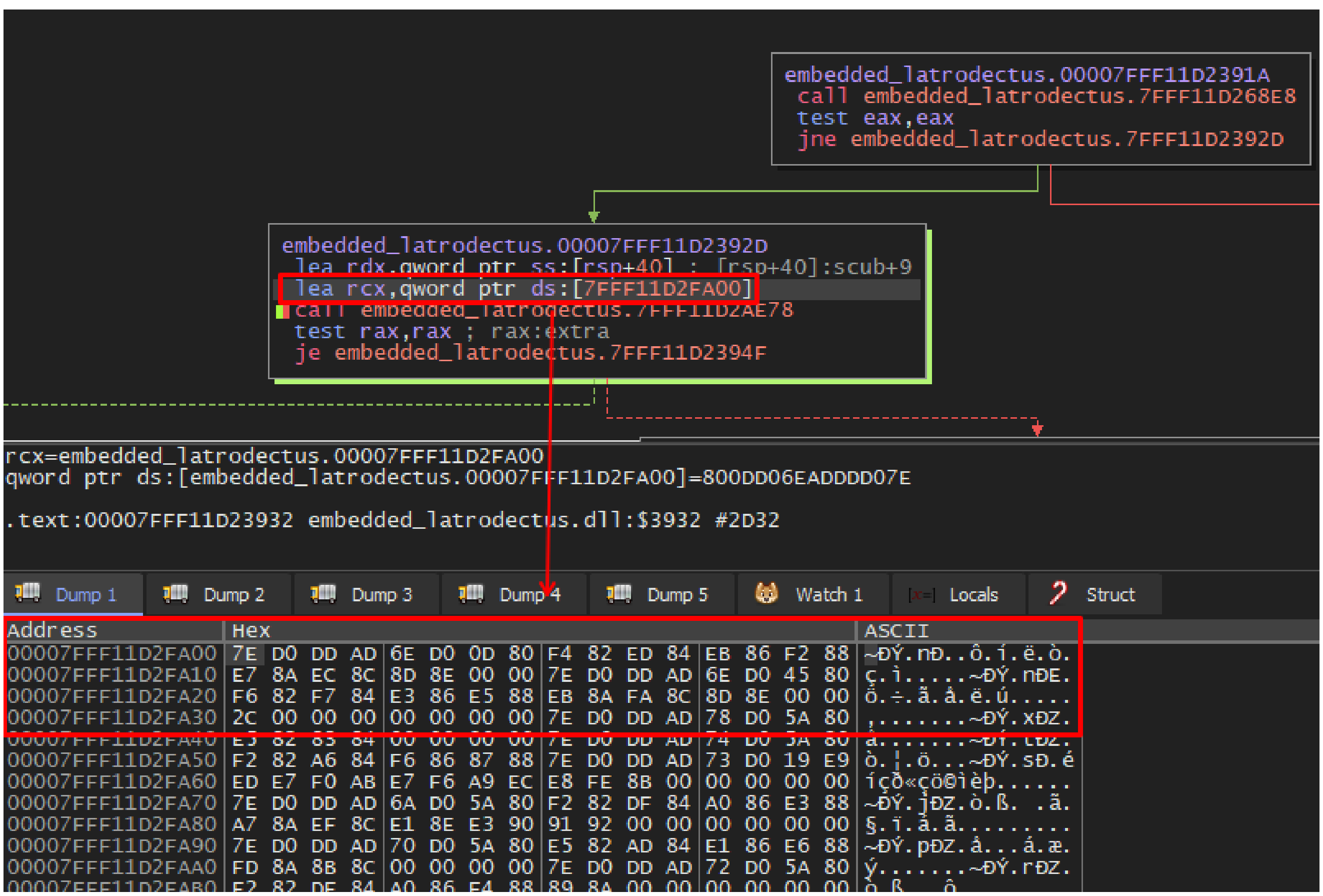
And below, we can observe the same moment. As our notes will not be present in the debugger, it is recommended that you set seven breakpoints and write comments, to remember where each action is done.



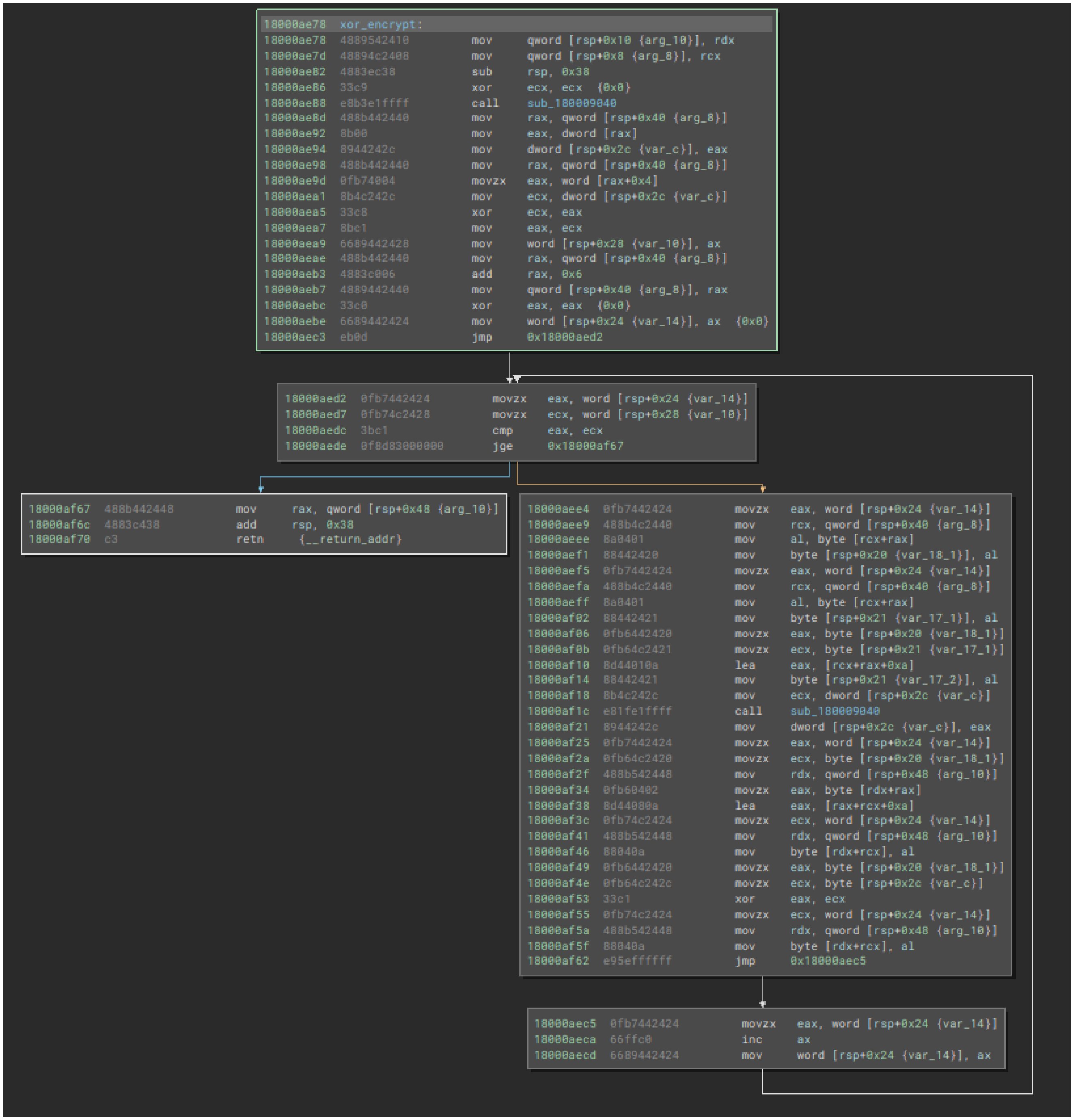
To validate where we are, below we can see the content of the encrypted data block that the xor_decrypt function (renamed by me, for documentation purposes) receives as an argument.

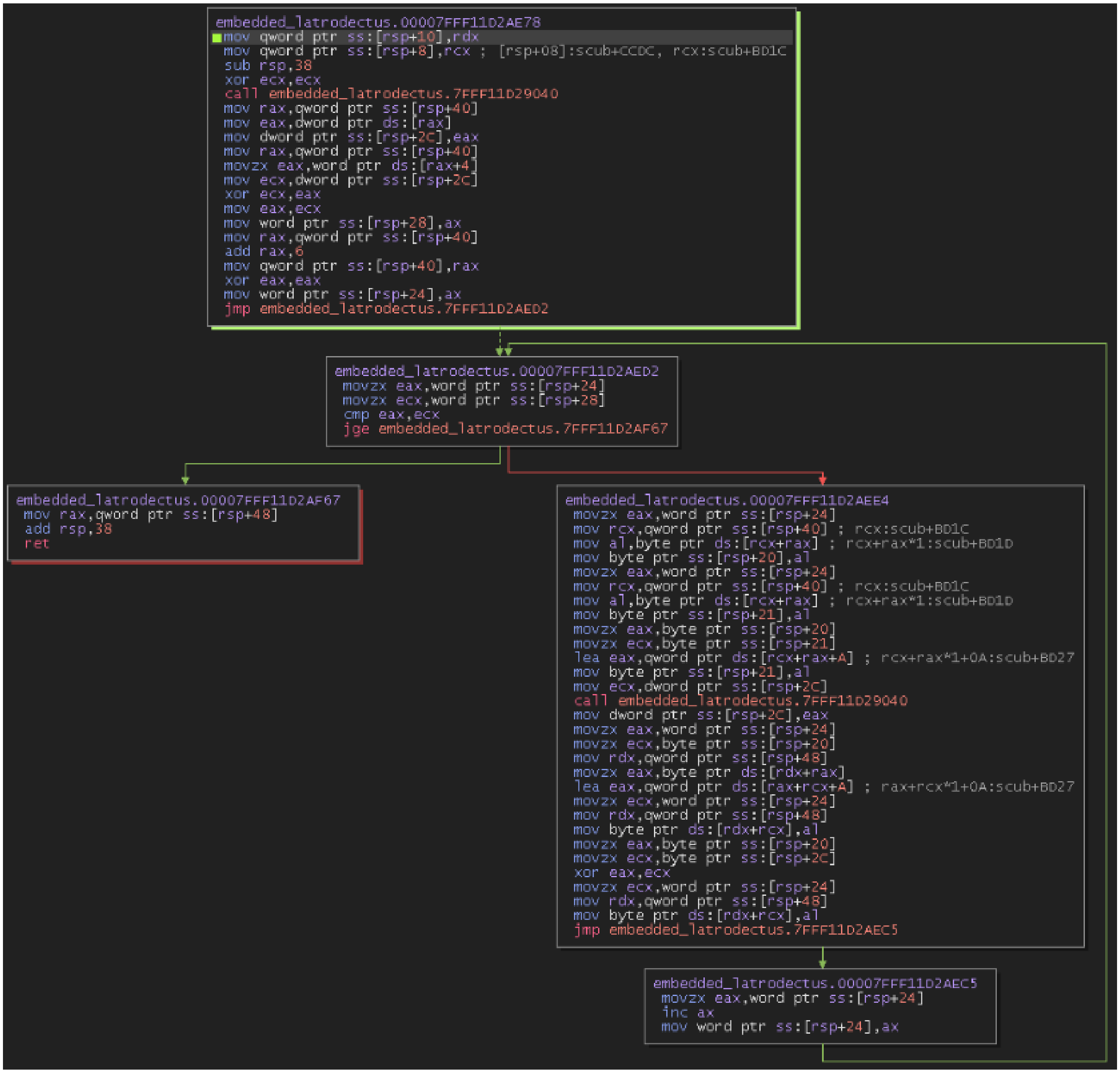


If we do the same thing with address 7FFF11D2FA00, we will observe the same data.



When we enter the function (Step-In in the debugger), we can also validate that the execution flow through graphical mode is the same. You can observe the comparison in the sequence of images, and see that we are in fact in the correct function.



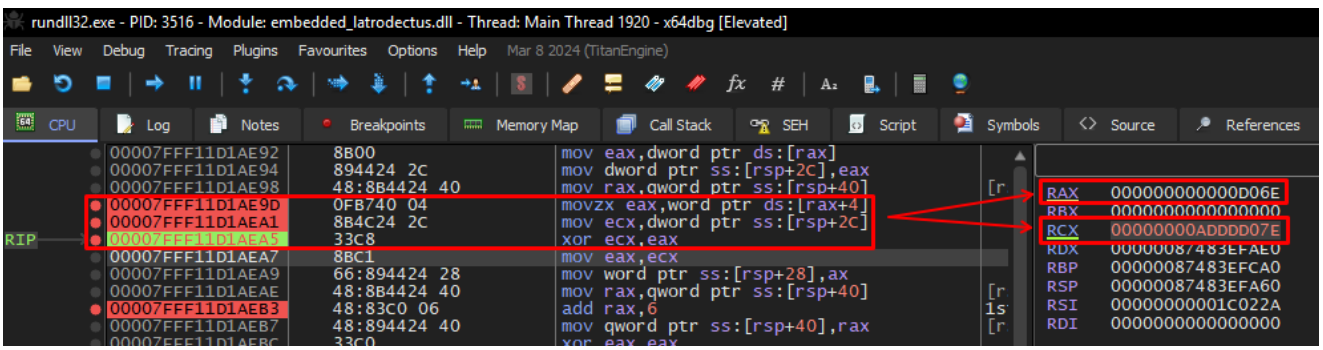


We can also use Decompiler to give us a hand in analyzing this algorithm. In our pseudo-code, we can see that first there is an *XOR* operation between some bytes within the data block itself. Then, **rcx_1** is used as a conditional for the *while loop* to continue executing, as long as **var_14** (set to **0**) is less than **rcx_1**. This is where we can assume from experience that right now the algorithm is calculating the value of the block of data that will be decrypted. After all, the block needs to have an end.

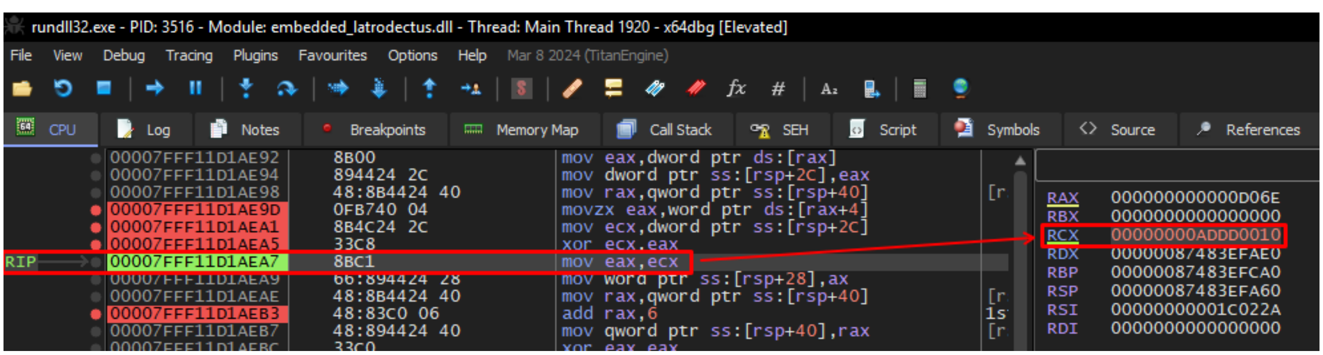
```
18000ae78  int64_t xor_encrypt(int32_t* arg1, int64_t arg2)

18000ae94      int32_t var_c = *arg1
18000aea5      int16_t rcx_1 = var_c.w ^ arg1[1].w
18000aebe      int16_t var_14 = 0
18000aede      while (zx.d(var_14) s< zx.d(rcx_1))
18000aeee          uint64_t rax_9
18000aeee          rax_9.b = *(arg1 + 6 + zx.q(var_14))
18000aef1          char var_18_1 = rax_9.b
18000aeff          uint64_t rax_10
18000aeff          rax_10.b = *(arg1 + 6 + zx.q(var_14))
18000af14          char var_17_2 = rax_10.b + var_18_1 + 0xa
18000af21          var_c = sub_180009040(var_c)
18000af46          *(arg2 + zx.q(var_14)) = *(arg2 + zx.q(var_14)) + var_18_1 + 0xa
18000af5f          *(arg2 + zx.q(var_14)) = var_18_1 ^ var_c.b
18000aecd          var_14 = var_14 + 1
18000af70      return arg2
```

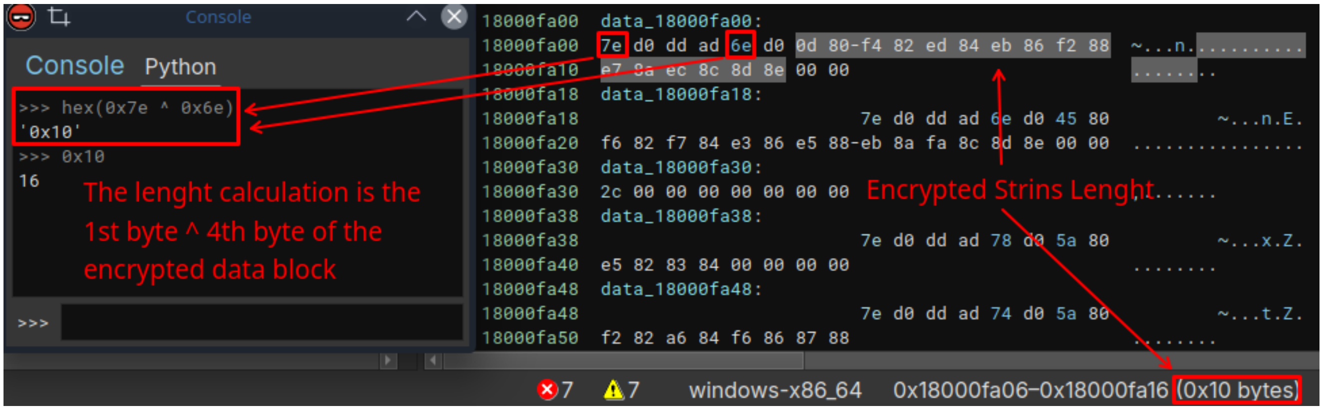
To validate, we can check in the debugger. Below, we can see the suspicions of what we saw in the pseudo-code above. The algorithm selected two bytes present in the data block, **0x7e** and **0x6e**, and performed an **XOR** operation between these two values.



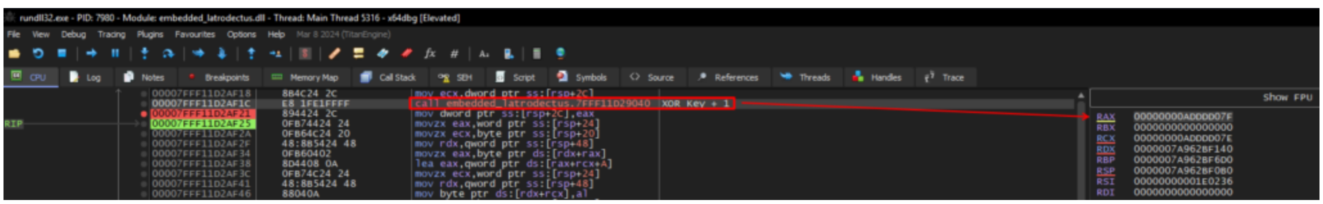
The value of this **XOR** operation was **0x10**, as we can see in the **RCX** register.



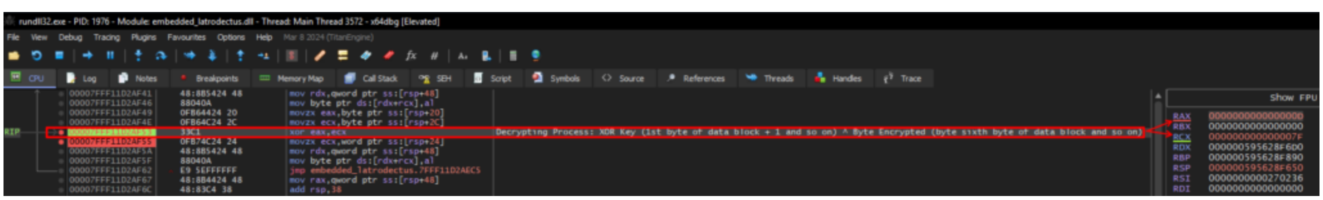
If we check in our Disassembler, byte **0x7e** is the first byte of the every data block, and **0x6e** is the fifth byte of this specific data block. In the image below, we can also redo the operation through the *Binary Ninja Python console*, where the value will also give **0x10**, which in decimal is **16**. And if we further analyze the block of data in question, we will also be able to observe that **0x10** is the *exact size of this specific data block*, before null values. In other words, in fact, the algorithm sets the size of the current data block that will be decrypted, and uses the value of its size as a conditional for the while loop.



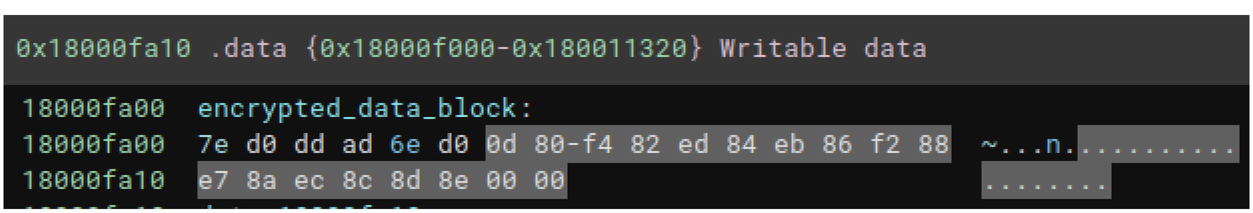
As we proceed, the decryption algorithm calls a function that we can also observe in the pseudo-code. This function simply adds *1 byte* (going from **0x7e** to **0x7f**) to the first byte of the encrypted data block.



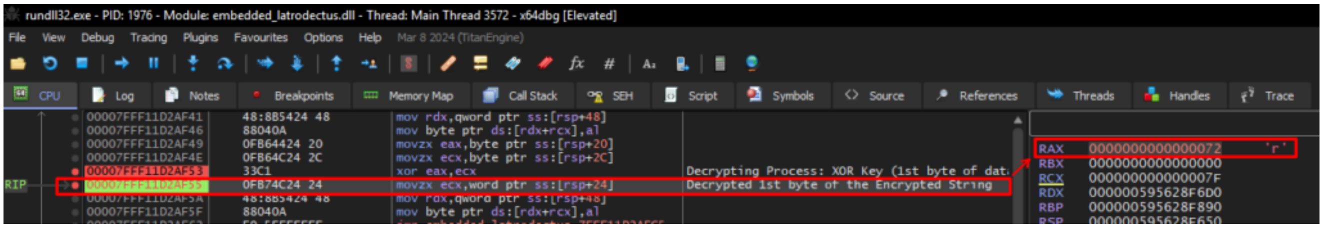
Next, the algorithm will perform an **XOR** operation with the byte appended to *1* (**0x7f**) and with byte **0x0d**, which is the *seventh byte of the encrypted data block*.



We can validate this information in our Disassembler, where it is possible to observe that the algorithm skips the initial *6 bytes* of the encrypted data block.



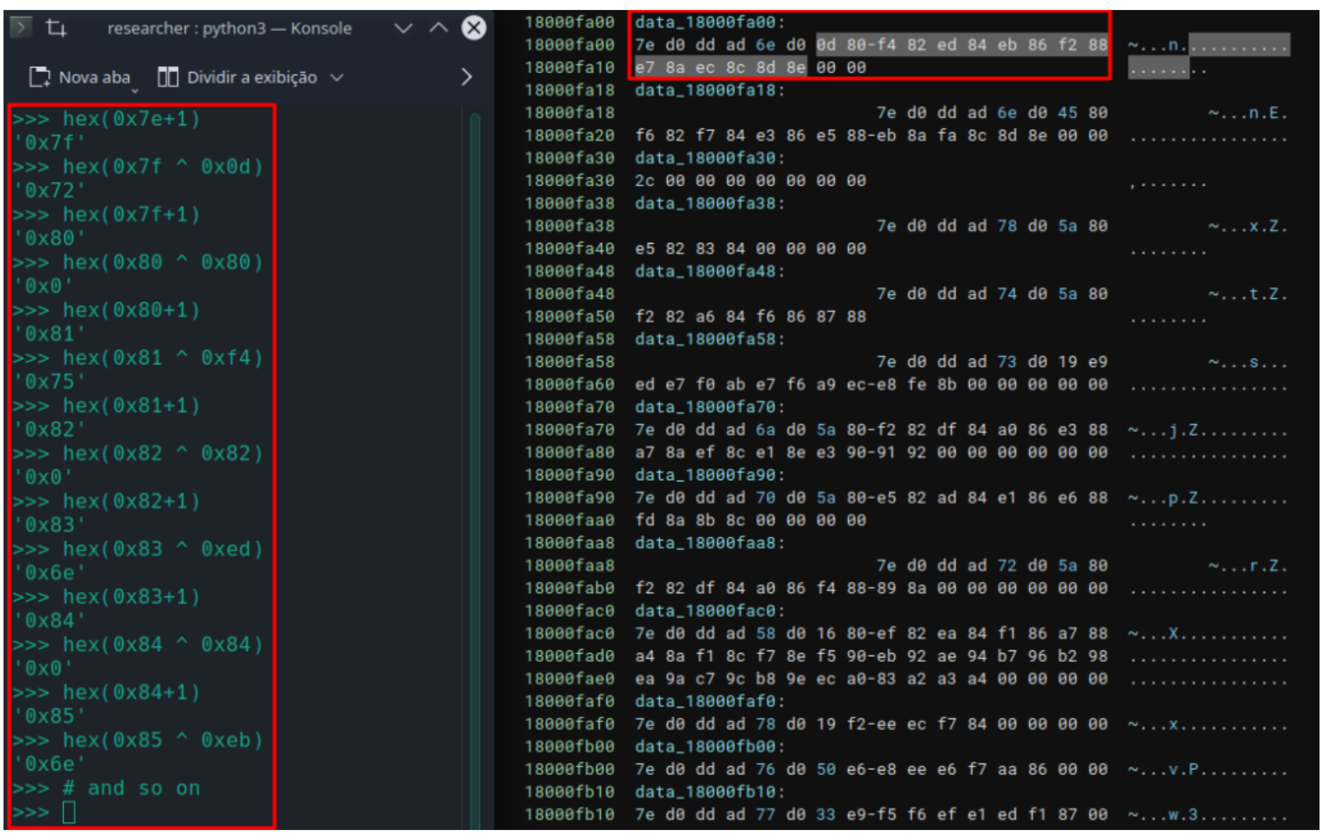
When we perform the **XOR** operation between the values **0x7f** and **0x0d**, the result (stored in **RAX**) will be a string identified as '*r*'.



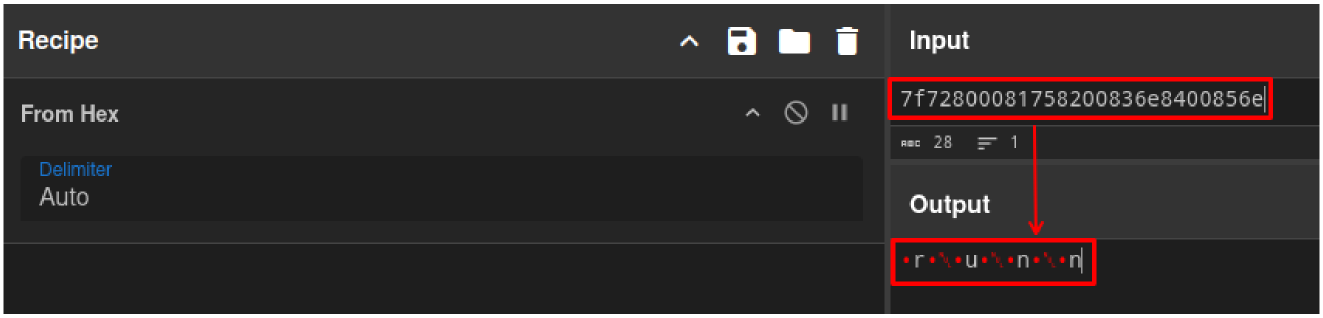
Having analyzed this behavior, we reached the following conclusion:

- The first byte of all blocks to be decrypted is the initial byte, which will always have its value increased by **1** for each subsequent byte (starting from the seventh byte of the encrypted data block) in which the **XOR** operation will be performed. That is, each byte will have a different **XOR key**.
- The *fifth byte* of each encrypted data block will be used together with the *first byte* of the block to calculate the size of the block that must be decrypted
- In other words, the *first six bytes* of each encrypted data block are not decrypted, they are what we can call the *control header*.

Below, we can validate our assumption. Below, I manually made the algorithm execution flow.



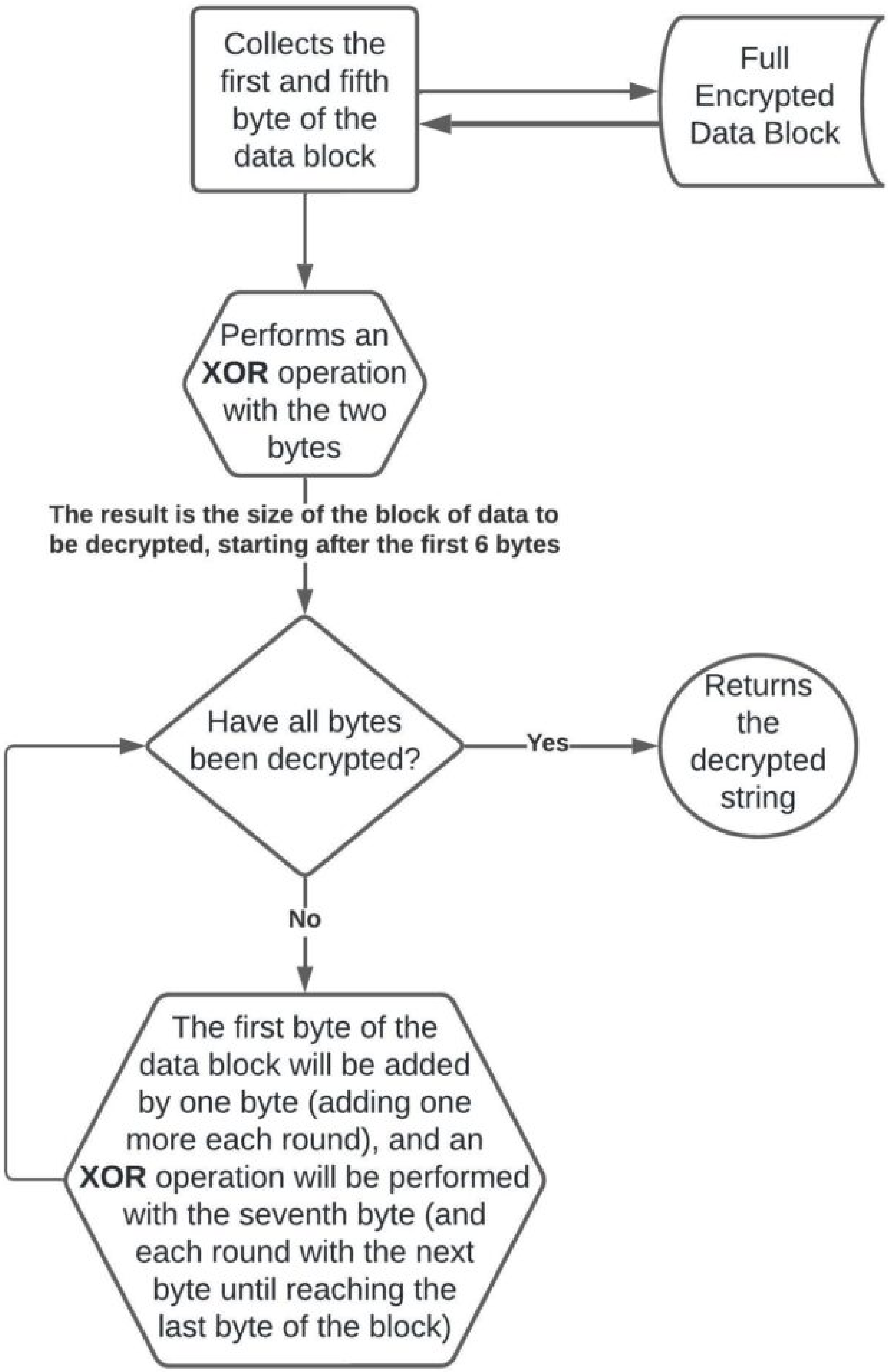
Upon obtaining a certain set of bytes, I went to **CyberChef** to transform the hex data into readable output, and… *Voilà*!



As we know from the *Latrodectus* analysis in my previous post, the string above is part (I just streamed it in a few bytes, out of laziness) of the **runnung** string, which is used to create the *Mutex* on the infected system.

Latrodectus Decryption Algorithm Flowchart

In order to improve understanding of the algorithm, below is a flowchart I made just to illustrate the flow of executing the *Latrodectus* string decryption algorithm.


Collects the first and fifth byte of the data block
Full Encrypted Data Block
Performs an XOR operation with the two bytes
The result is the size of the block of data to be decrypted, starting after the first 6 bytes
Have all bytes been decrypted?
Yes
Returns the decrypted string
No
The first byte of the data block will be added by one byte (adding one more each round), and an XOR operation will be performed with the seventh byte (and each round with the next byte until reaching the last byte of the block)

Once you understand the algorithm, you can implement this algorithm in a script, with the aim of extracting the strings from the sample you are analyzing.

Python Script for String Decryption and Extraction

I created a Python script that will run the *Latrodectus* decryption algorithm, print the entire flow of its execution for debugging and study.

Below is the video of the execution of the script.

Python-Only – Latrodectus String Extractor

The source code of the script can be found on my *github*, at the link below:

Conclusion

Well, I hope this type of article pleased you, the reader, and that you learned something new!! See you around!