

# Ghidra Tutorial - Using Entropy To Locate a Cobalt Strike Decryption Function

embee-research.ghost.io/ghidra-entropy-analysis-locating-decryption-functions/

Matthew

October 18, 2023

Last updated on Oct 18, 2023

```
19
20 puVar4 = auStack_68;
21 for (lVar3 = 0x18; lVar3 != 0x0; lVar3 = lVar3 + -0x1) {
22     *puVar4 = 0xc0000000;
23     puVar4 = puVar4 + 0x1;
24 }
25 if ((param_2 == 0x10) || (param_2 == 0x18) || (param_2 == 0x20)) {
26     if ((param_3 == 0x0) ||
27         (param_3 == ((int)(param_2 + (param_2 >> 0x1f & 0x7U)) >> 0x3) * 0x2 + 0x6)) {
28         param_4[0x78] = ((int)(param_2 + (param_2 >> 0x1f & 0x7U)) >> 0x3) * 0x2 + 0x6;
29         local_48 = 0x0;
30         *param_4 = (uint)*param_1 << 0x18 | (uint)param_1[0x1] << 0x10 | (uint)param_1[0x2] << 0x8
31         |
32         (uint)param_1[0x3];
33         param_4[0x1] = (uint)param_1[0x4] << 0x18 | (uint)param_1[0x5] << 0x10 |
34         (uint)param_1[0x6] << 0x8 | (uint)param_1[0x7];
35         param_4[0x2] = (uint)param_1[0x8] << 0x18 | (uint)param_1[0x9] << 0x10 |
36         (uint)param_1[0xa] << 0x8 | (uint)param_1[0xb];
37         param_4[0x3] = (uint)param_1[0xc] << 0x18 | (uint)param_1[0xd] << 0x10 |
38         (uint)param_1[0xe] << 0x8 | (uint)param_1[0xf];
39         local_38 = param_4;
40         if (param_2 == 0x10) {
41             local_44 = 0x2c;
42             while( true ) {
43                 local_40 = local_38[0x3];
44                 local_28 = 0x0;
45                 uVar1 = FUN_18002a060(local_40);
46                 local_38[0x4] =
47                 *(uint *) ((longlong)local_38 + local_28) ^ uVar1 ^
48                 *(uint *) (&DAT_180037fa0 + (longlong)local_48 * 0x4);
49                 local_38[0x5] = local_38[0x1] ^ local_38[0x4];
50                 local_38[0x6] = local_38[0x2] ^ local_38[0x5];
51                 local_38[0x7] = local_38[0x3] ^ local_38[0x6];
52                 local_48 = local_48 + 0x4;
53                 if (local_48 == 0xa) break;
54                 local_38 = local_38 + 0x4;
55             }
56         }
57     }
58 }
```

Lots of bitwise operators (>> / shr) and (^ / xor).

A strong indicator that this function is doing some kind of Encryption/Decryption.

Using Ghidra to analyse malware can be a difficult and daunting task. This task is often complicated through the use of encryption and the general complexity of using Ghidra for the first time.

In this blog, I will demonstrate a simple workflow that you can use to speed up this process.

By using the entropy view within Ghidra, you can quickly hone in on functions related to encryption, and use this to identify areas that you can analyse in a debugger or develop into Yara rules.

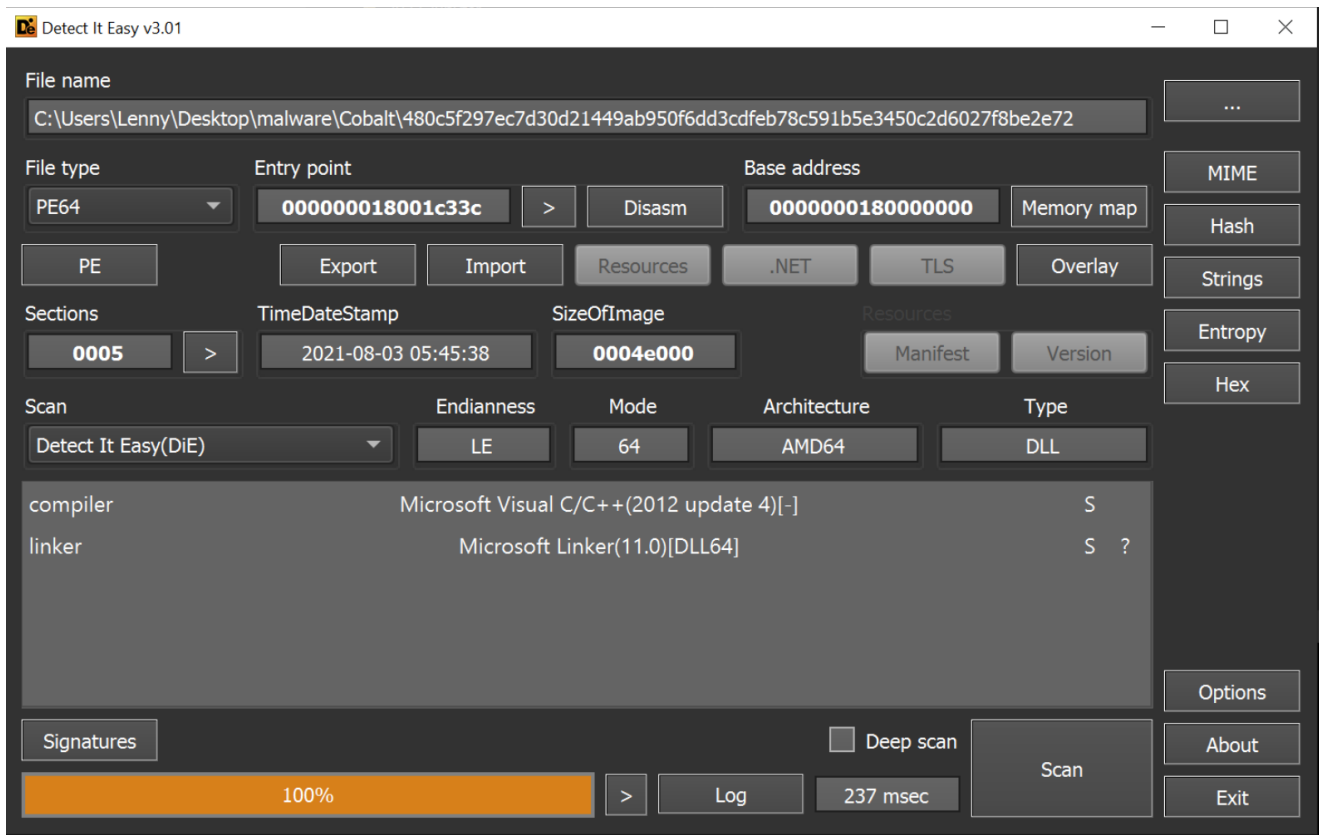
In this short blog, I will be using the sample

[480c5f297ec7d30d21449ab950f6dd3cdfef78c591b5e3450c2d6027f8be2e72](#)

[Link to File Here](#)

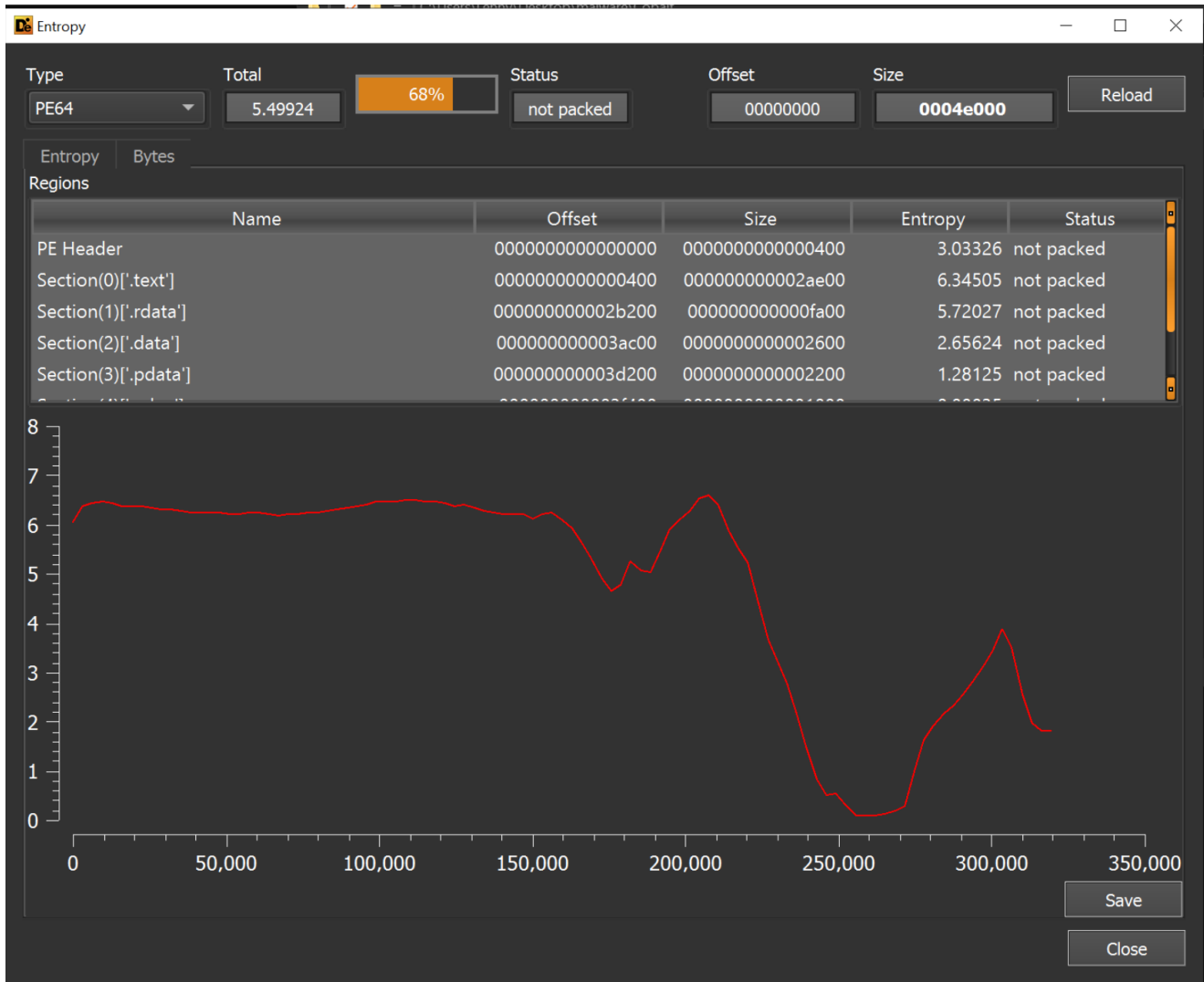
## Initial Analysis

The initial file I'll be using is a 64-bit dll file that was initially marked as cobalt strike.



During initial analysis, I typically view use the entropy view of Detect-it-easy to identify if there are any large areas of high entropy, which typically indicates encrypted content. These areas are something that I tend to hone in on in my next step of analysis.

In this case, there are no indications of high entropy or packing.



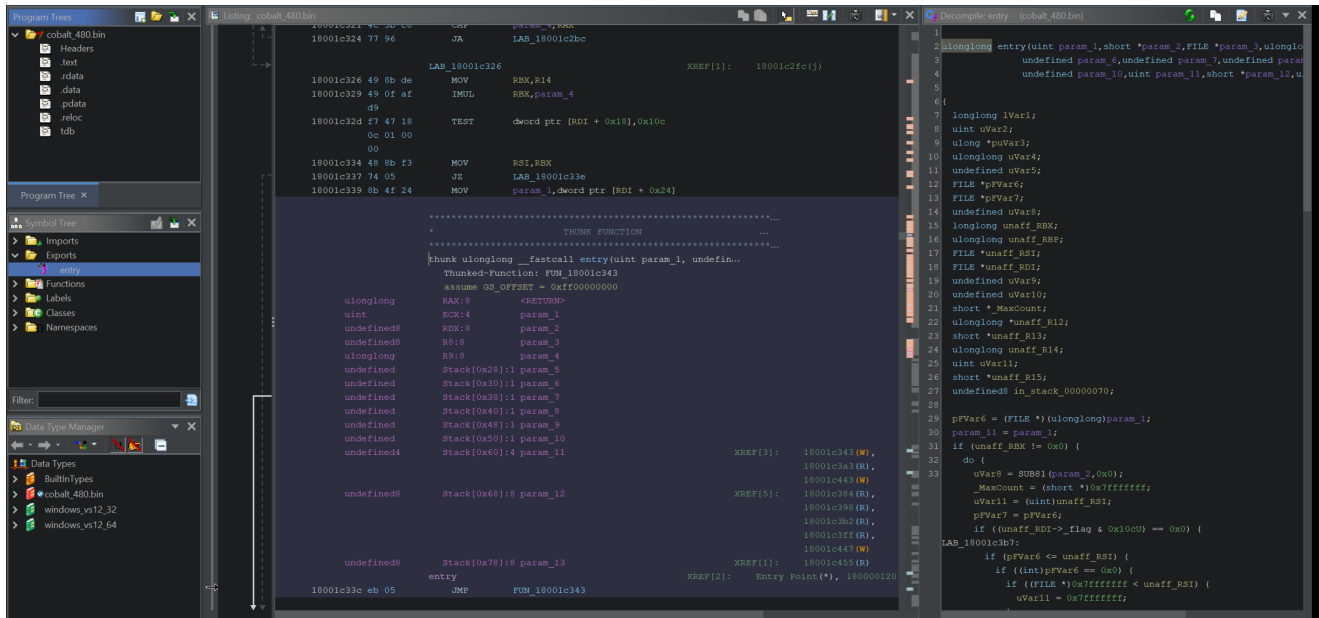
Since there are no significant sections of high-entropy, I will instead use Ghidra to hone in further.

The lack of large high-entropy areas suggests that there are no embedded payloads. However, there may be smaller areas of high entropy that contain configuration data (c2's, url's) or otherwise useful information.

We can try and use Ghidra to determine this further.

## Cobalt Strike Analysis With Ghidra

After loading the file inside of Ghidra, a screen like this is presented. This is a lot of information and generally a difficult place to start.



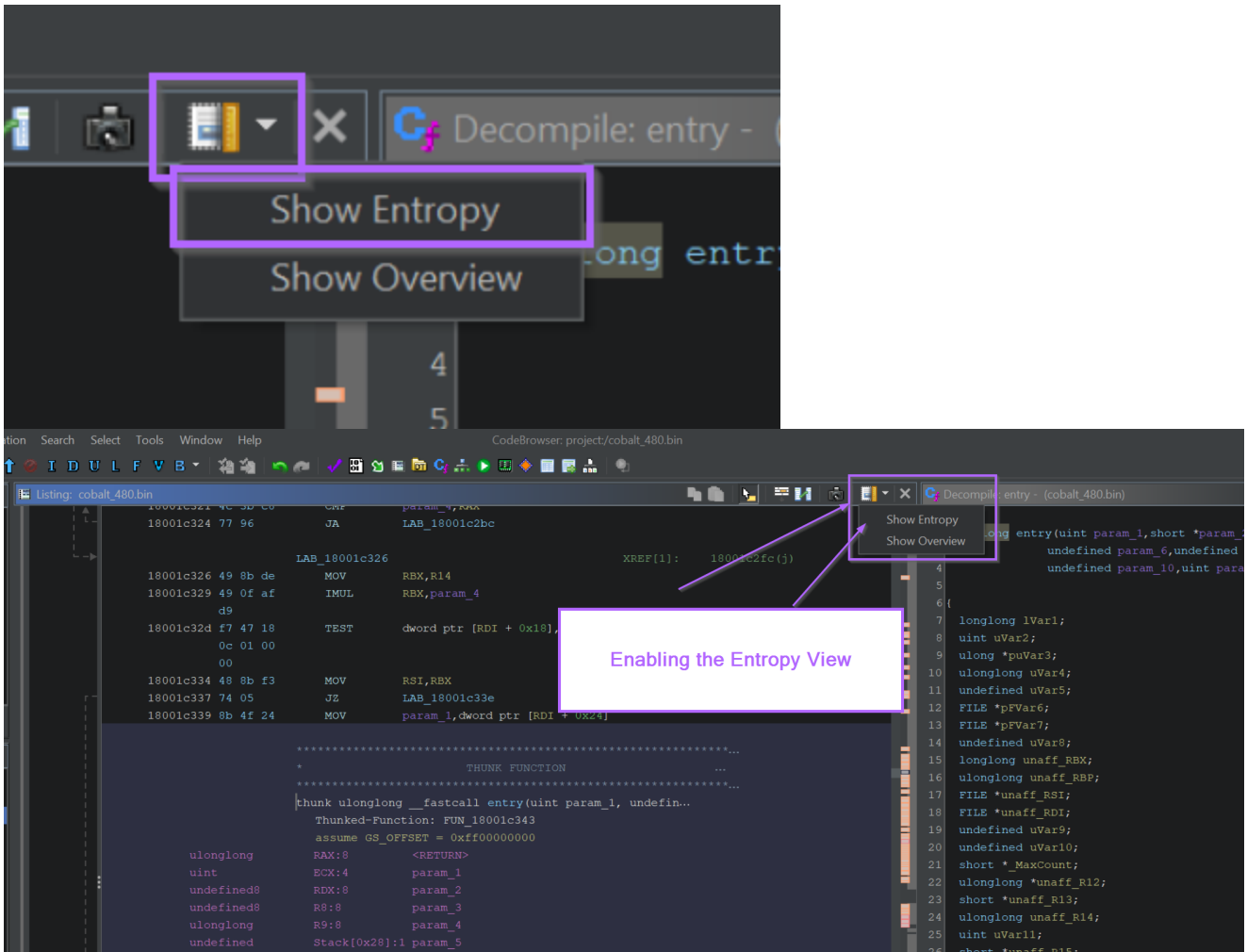
There are lots of things you can do from here, but for the purpose of this blog I will be honing in on Entropy, and using the entropy to identify decryption functions that can be used in a Yara rule.

To achieve this, a few steps need to be taken

- Enable the entropy view
- Locate any areas of High-entropy (typically indicated by red blobs)
- Use the "most recent label" to locate the beginning of high-entropy areas
- Observe any cross-references to the start of the high-entropy area (This shows any function that is acting on the entropy, typically this will be a decryption method)
- If a decryption method is found, look for unique instructions that can be used in a Yara rule.
- Additionally - Use a debugger (like x64dbg) to analyse the decryption function.

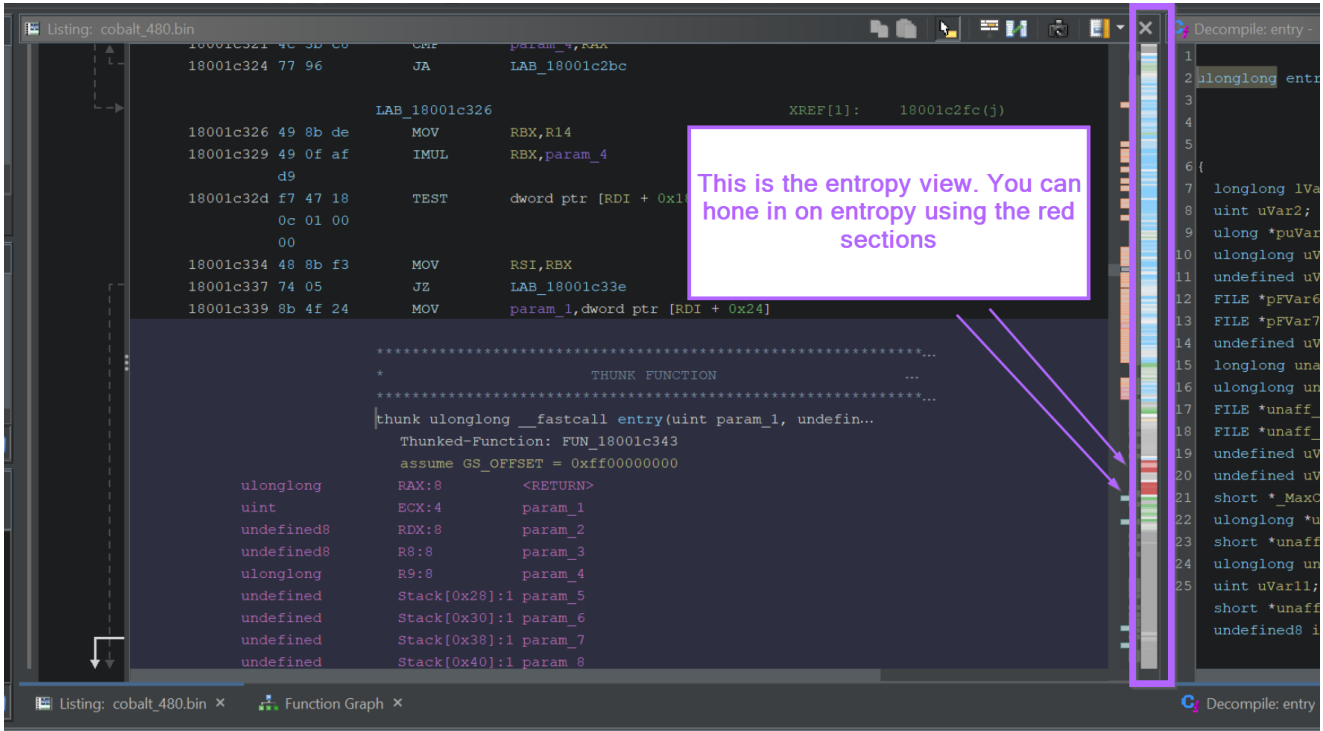
## Enabling the Entropy View in Ghidra

Enabling the entropy view is simple. You can use the top-right box to enable a dropdown menu that contains the "Show Entropy" setting.



With the entropy view enabled, a small window shows up that enables you to view entropy within the file.

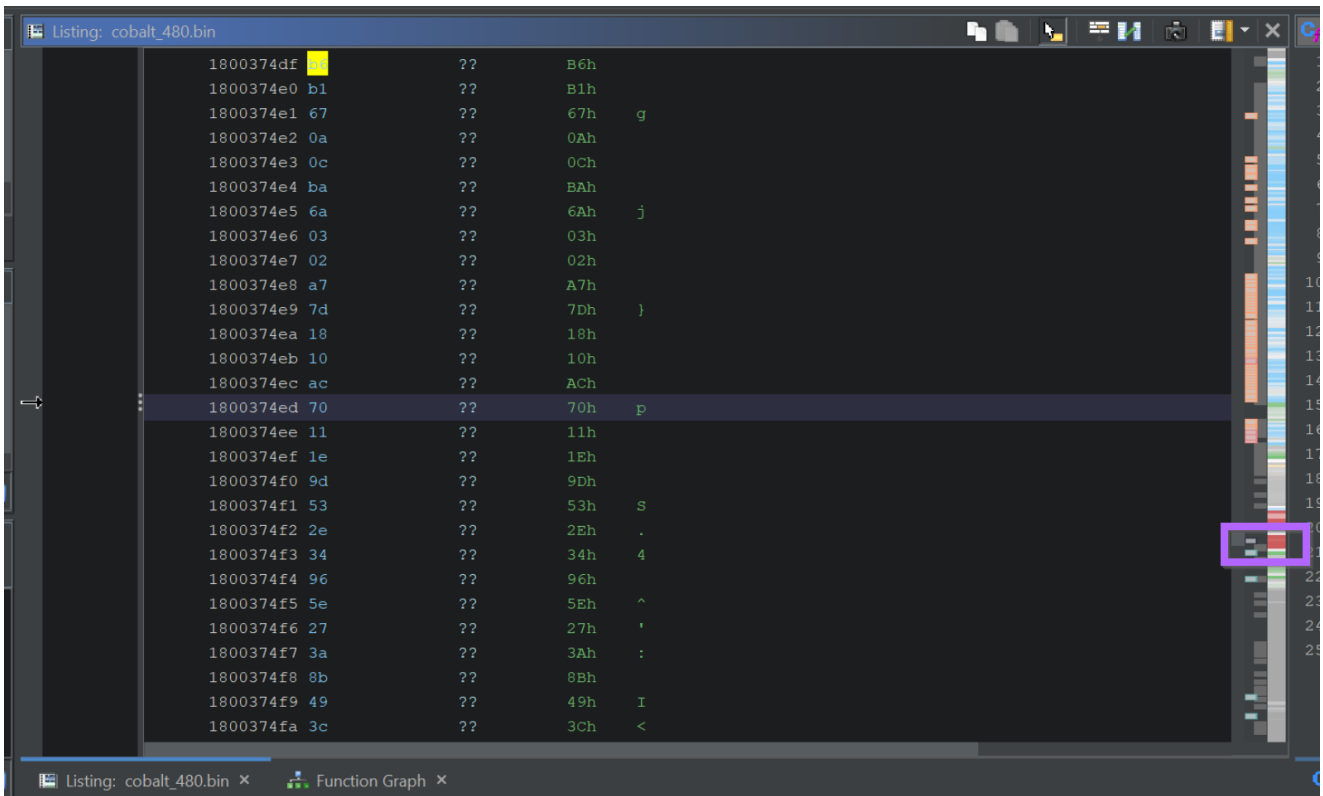
High Entropy Areas are indicated by Red blobs. The red blobs can be clicked to jump straight to the high-entropy section.



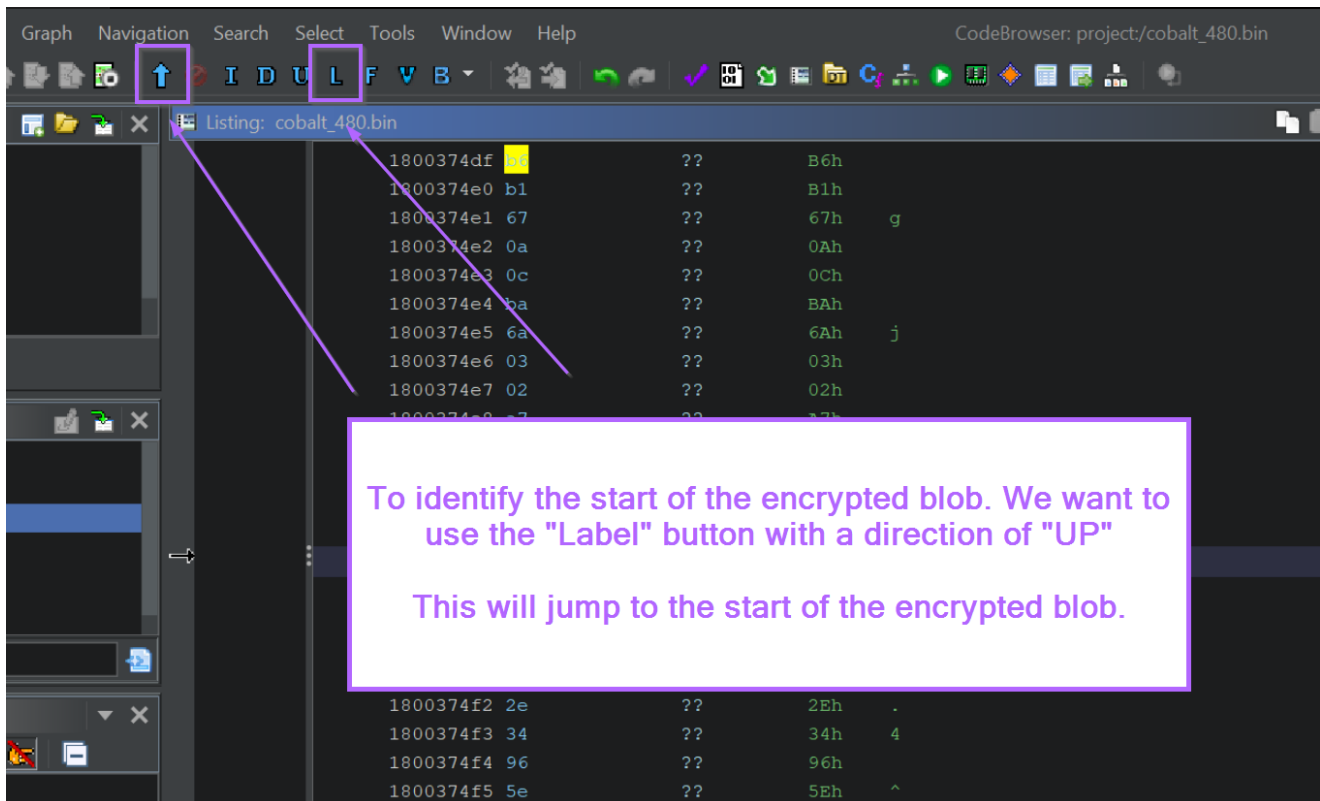
## Locating A Decryption Function With Ghidra

I tend to start with the largest red blob first. Clicking on the larger red blob shows the following view.

Initially this is just a blob of encrypted bytes. More information can be extracted by jumping to the beginning of the blob area.

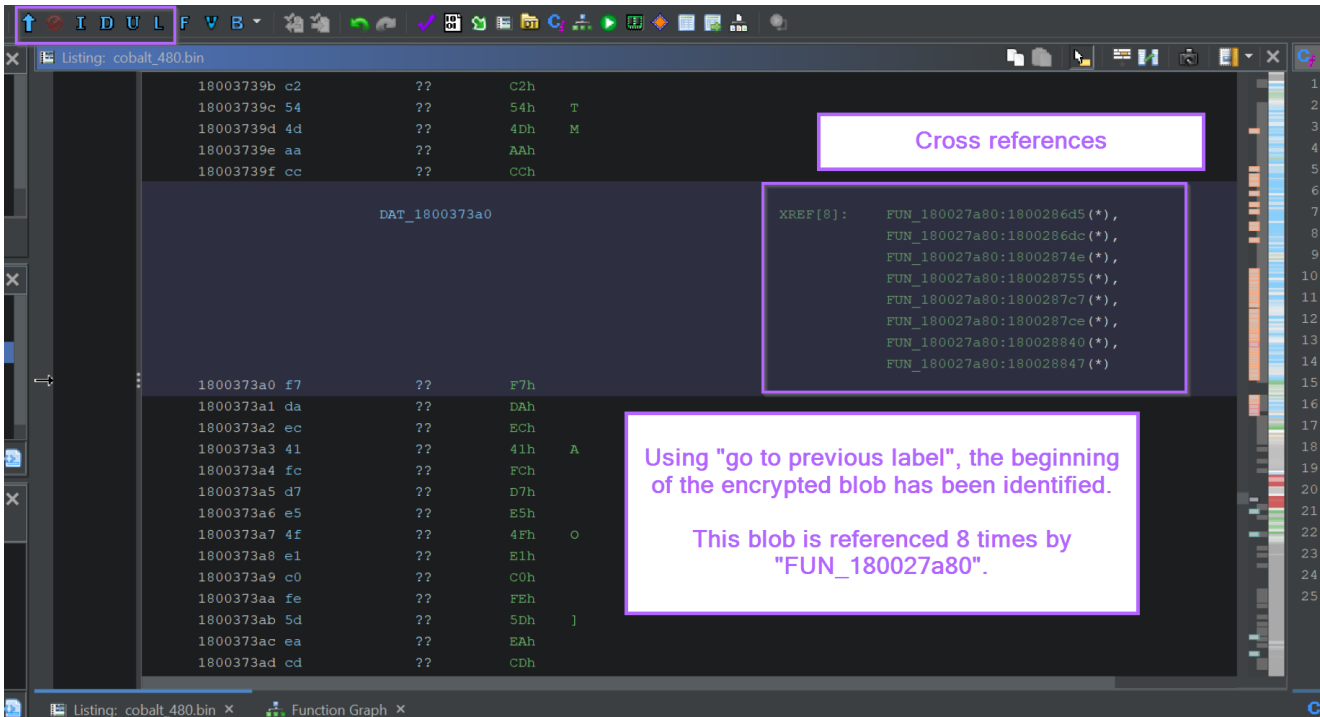


To locate the start of the encrypted blob, we can use the "Go To Previous Label" button, making sure to set the arrow direction to "UP".

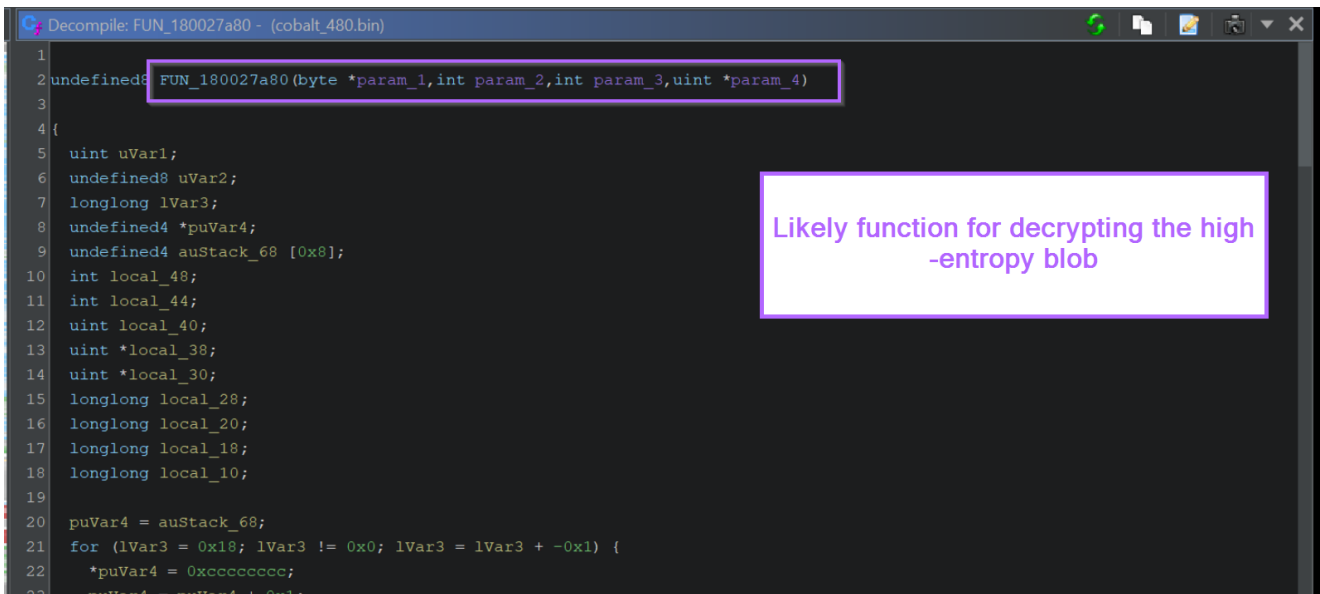


This will move the screen to the beginning of the encrypted blob, in this case the beginning was found at `DAT_1800373a0`. We can also see that this location is referenced 8 times by the function `FUN_180027a80`.

In most cases, this is a very strong indicator that `FUN_180027a80` is the function responsible for decrypting the blob.



Clicking on any of the references takes us to the responsible function.



Scrolling down slightly reveals a significant number of bitwise operators such as XOR  $\wedge$  and SHR  $\gg$ .

This is generally a strong indicator of an encryption/decryption function.



```

19
20 puVar4 = auStack_68;
21 for (lVar3 = 0x18; lVar3 != 0x0; lVar3 = lVar3 + -0x1) {
22     *puVar4 = 0xcccccccc;
23     puVar4 = puVar4 + 0x1;
24 }
25 if ((param_2 == 0x10) || (param_2 == 0x18) || (param_2 == 0x20)) {
26     if ((param_3 == 0x0) ||
27         (param_3 == (int)(param_2 + (param_2 >> 0x1f & 0x7U)) >> 0x3) * 0x2 + 0x6) {
28         param_4[0x78] = (int)(param_2 + (param_2 >> 0x1f & 0x7U)) >> 0x3) * 0x2 + 0x6;
29         local_48 = 0x0;
30         *param_4 = (uint)*param_1 << 0x18 | (uint)param_1[0x1] << 0x10 | (uint)param_1[0x2] << 0x8
31         |
32         (uint)param_1[0x3];
33         param_4[0x1] = (uint)param_1[0x4] << 0x18 | (uint)param_1[0x5] << 0x10 |
34         (uint)param_1[0x6] << 0x8 | (uint)param_1[0x7];
35         param_4[0x2] = (uint)param_1[0x8] << 0x18 | (uint)param_1[0x9] << 0x10 |
36         (uint)param_1[0xa] << 0x8 | (uint)param_1[0xb];
37         param_4[0x3] = (uint)param_1[0xc] << 0x18 | (uint)param_1[0xd] << 0x10 |
38         (uint)param_1[0xe] << 0x8 | (uint)param_1[0xf];
39     }
40     local_38 = param_4;
41     if (param_2 == 0x10) {
42         local_44 = 0x2c;
43         while( true ) {
44             local_40 = local_38[0x3];
45             local_28 = 0x0;
46             uVar1 = FUN_18002a060(local_40);
47             local_38[0x4] =
48             *(uint *) ((longlong)local_38 + local_28) ^ uVar1 ^
49             *(uint *) (&DAT_180037Fa0 + (longlong)local_48 * 0x4);
50             local_38[0x5] = local_38[0x1] ^ local_38[0x4];
51             local_38[0x6] = local_38[0x2] ^ local_38[0x5];
52             local_38[0x7] = local_38[0x3] ^ local_38[0x6];
53             local_48 = local_48 + 0x1;
54             if (local_48 == 0xa) break;
55             local_38 = local_38 + 0x4;
56         }
57     }
58 }

```

Lots of bitwise operators (>> / shr) and (^/ xor).

A strong indicator that this function is doing some kind of Encryption/Decryption.

At this point, we can have high confidence that the decryption method has been identified.

There are a few interesting things that can be done here.


- Ask ChatGPT to identify the type of Encryption/Decryption Used
- Try to extract bytes that can be used in a Yara rule
- Use a debugger to obtain the decoded results.

## Using ChatGPT to Identify an Encryption Function

To identify the type of decryption/encryption used, I copied out the entire decompiled function as asked ChatGPT if it could identify it.

To do this, I asked **Can you identify the type of encryption or decryption used in this ghidra decompiled code?** and then pasted in the entire function.

E

Can you identify the type of encryption or decryption used in this ghidra decompiled code? 




```
undefined8 FUN_180027a80(byte *param_1,int param_2,int param_3,uint *param_4)
```

```
{
  uint uVar1;
  undefined8 uVar2;
```

ChatGPT suggested that the encryption type was AES. I personally don't know enough about AES to confirm this, but it's a useful suggestion and probably correct.

At this point you could try to identify the decryption key and write a script to decode the encoded data using an AES library.



The decompiled code you've shared is part of the key expansion process for the AES (Advanced Encryption Standard) algorithm. The key indicators include:   

1. The various key lengths checked: 0x10 (128 bits), 0x18 (192 bits), and 0x20 (256 bits) are standard AES key sizes.
2. The use of S-box lookups in the code (referenced by the addresses like `&DAT_180036fa0`, `&DAT_1800373a0`, etc.).
3. The cyclic shifting and substitution operations.
4. The function `FUN_18002a060` likely computes the AES key schedule's round constants or involves some AES-specific transformation (like the SubWord operation followed by an Rcon addition).
5. The generation and expansion of keys, as seen by the loop structures and bitwise operations.

The code performs key expansion to generate round keys from the initial key, which will be used in the encryption or decryption process of AES.

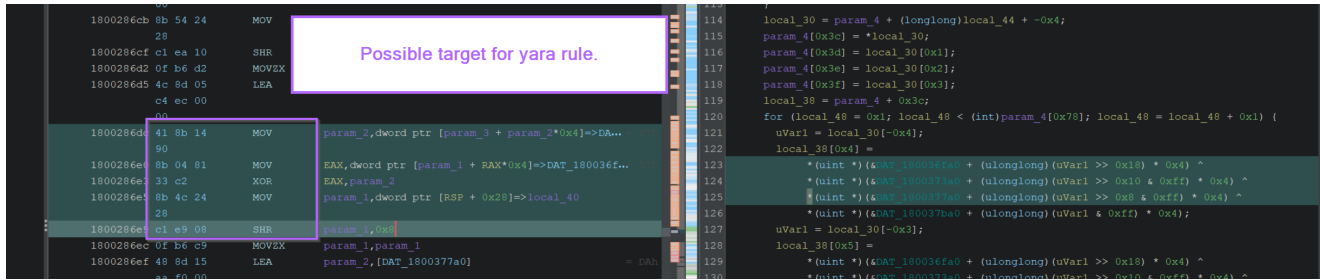
## Extracting Encryption Bytes For a Yara Rule

With the encryption function identified, you can try to find bytecodes that can be turned into a Yara rule.

This works best when the malware uses its own unique encryption/decryption function. It may not be the best here (since the AES usage may not be unique on its own), but it's something that can work for a lot of malware.

[\(Here's an example where it worked well for IcedID\)](#)

To create a Yara rule, you want to look for blobs that contain at least 2 math/bitwise operators. (XOR, SHR, SHL etc). From there you can extract bytecodes that can be used for Yara rules.



## Extracting Decoded Content With a Debugger

With knowledge of the decryption function and location of encrypted content, you can use a debugger to set breakpoints and extract information of interest.

To do this, you can either set a software breakpoint on the encryption/decryption function. Then jump to the end of the function and find the register or location containing decrypted content.

OR

With knowledge of where the encrypted content is located, you can set a hardware breakpoint on the location and receive an alert when it is acted on.

Both of these methods will achieve the same result. This is something I may write about in another blog post.

## Conclusion

In this post, we have used Ghidra to identify an encryption function present inside a Cobalt strike sample. We have identified an area of high entropy and also identified that the encryption used might be AES.

This information can be leveraged further to identify the decrypted contents via debugger, or to develop a Yara rule based on bytecodes present in the encryption.