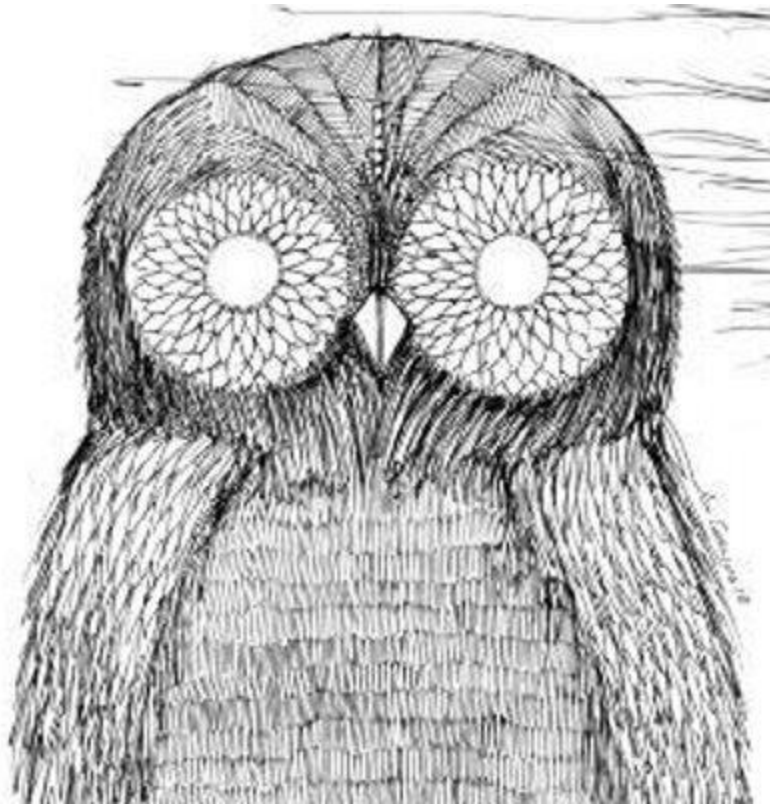


Reverse Engineering Gootkit with Ghidra Part I

dannyquist.github.io/gootkit-reversing-ghidra/



Open Malware - Danny Quist

Reverse Engineering and 3D Printing - Danny Quist

[Blog About](#)

Danny Quist

March 23, 2019

Ghidra is pretty handy for looking at malware. This series of post is an informal overview of what I do. Gootkit is a great implant to learn the functionality of Ghidra. Gootkit is a NodeJS server with packaged Javascript implementing the implant functionality. There are lots of libraries linked into the main executable including Node, OpenSSL, and many more. As a reverse engineer it is difficult to identify and identify open libraries. In this post, I will go through my analysis process to use and understand Ghidra's functionality.

I will first begin by basic code analysis, and understanding how to rename variables and types. I am going to avoid dynamic analysis initially, because dynamic analysis is something that you can buy or implement cheaply enough. In a real-world scenario I typically start dynamic analysis using a range of tools, then delve into the code as a secondary step.

The purpose is to learn Ghidra, not to do a great job at reverse engineering all of Gootkit. It is highly informal, and meant to be that way.

Ghidra All the Things!

There are now a few tutorials available on [installing and configuring Ghidra](#). Create a new project, and then import the decrypted rbody32 sample into the project. The sample I will be using is:

```
$ shasum rbody32.x.dec  
6170e1658404a9c2655c13acbe1a2ad17b17feae
```

It is a decoded version of the file downloaded from a compromised Gootkit site. While Gootkit is the topic for this blog, this process can be applied generally to anything else.

Your import summary should look a lot like this:

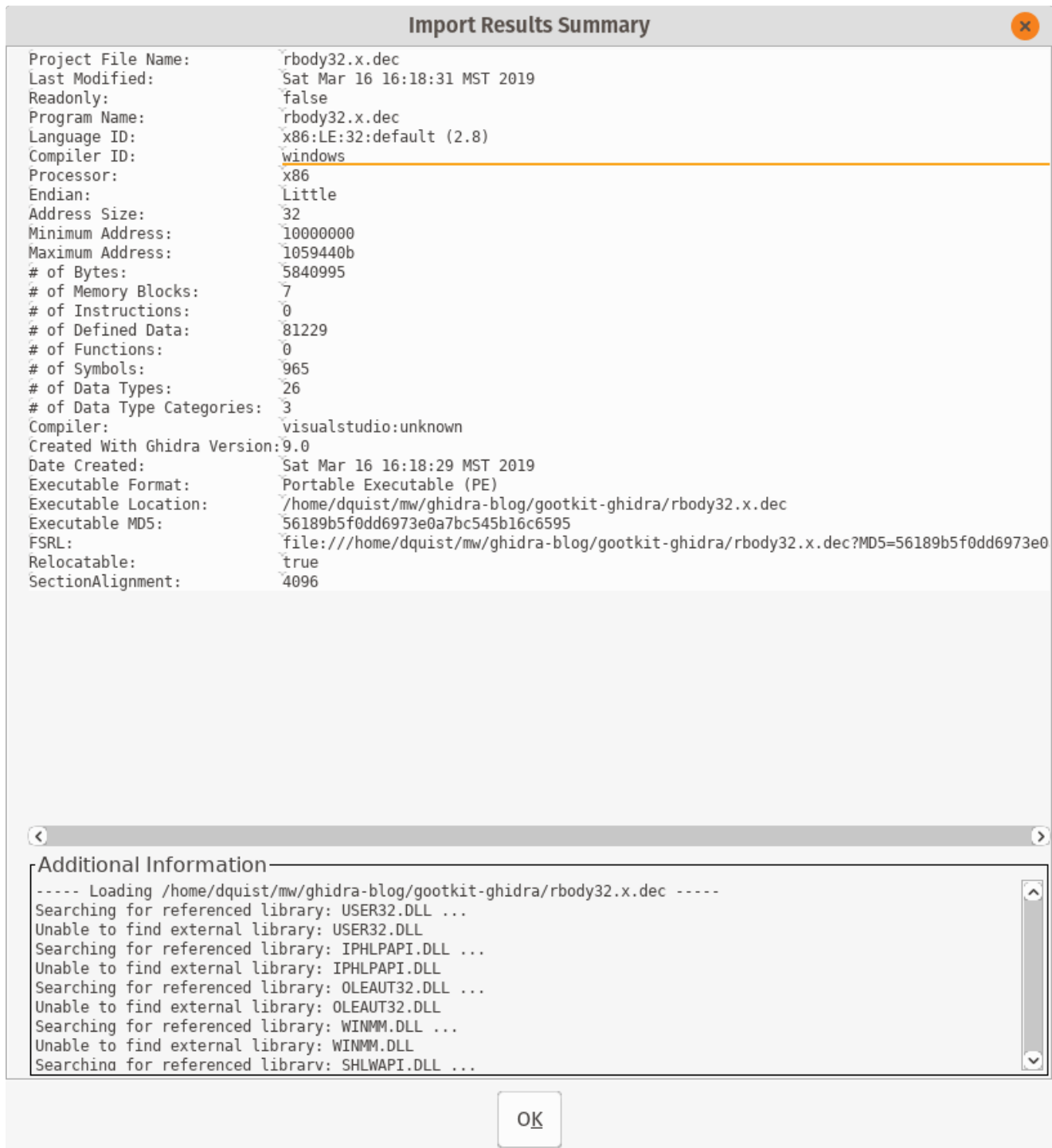


Figure 1: Ghidra import summary for the relevant Gootkit example

Ghidra Import Summaries

Import summaries tell you critically important facts about the sample that you're looking at. The key thing to remember is that Ghidra is primarily a source code reverse engineering tool. There are a few salient bits to draw your attention to:

First, compiler identification. In this case Ghidra identifies `VisualStudio:unknown` as the likely compiler. This makes sense, as it is based off of a NodeJS, which is a C++ program, and Visual Studio is the compiler of choice for Windows. Knowing the compiler is important later when you're puzzling through some obtuse assembly code, trying to figure out if the compiler generated some weird code, or the malware author was being tricky. Ghidra is excellent about identifying and categorizing compiler generated nonsense, and saves a bunch of time.

Second, `Compiler ID` appears to be the the platform that the compiler was run on. As you look at more assembly code, you'll get a good idea of how each of them generate code for standard C and C++ programming patterns. My indicator when looking at code is whether or not it was hand-rolled assembly, or is compiler generated. Typically hand-rolled, artisinally crafted assembly is a good indication that there are shenanigans afoot. Hand coded assembly can be significantly more difficult to understand, where a compiler will try to do things the same way.

Why do I care so much about compiler produced versus hand-coded assembly? As an analyst, you have a budget of time and attention that you can focus on every bit of code. During an investigation I tend to hit a point of diminishing returns where fatigue sets in, and I start to miss critically important details. The code placed around checking return values and stack canaries is something I spend way too much time classifying in a sample. If a tool can identify that, I can label it as not important and go on with life. If the tool does not identify that, or more likely I get drawn in anyway, there are all sorts of suspicious APIs that are very distracting. `ExitProcess`, anything thread related, etc.

Additional information is an excellent resource too. Looking at the high-level DLLs the sample is using can give you an idea of what the functionality is going to be.

Existing Gootkit Research

Largely this document will consist of reproducing the already existing Gootkit analyses. Gootkit is served from a compromised host and runs a small command and control server. The user is tricked/hacked into downloading a compromised PDF/DOC/implant, which then contacts the call-home server. Generally if you see `.* /rbody32` or `.* /rbody320` in the URL, you've most likely got the right sample.

@jgegeny has a copy of the [extracted JavaScript files](#). The functionality signatures, and overall path to success depends on understanding the JavaScript. I will focus on trying to extract them.

In general the things you need to know about Gootkit:

1. It's based on a all-in-one compiled version of a NodeJS application. If you ever needed a more clear and present indication that Node is evil, look no further

2. It has a second DLL inside of it to handle password and credential harvesting.
3. All of the functionality exists as JavaScript files, which we would like to decode and obtain.

Analyzing Gootkit


Analysis Goals

1. Generate new indicators of compromise
2. Find attribution information for the authors
3. Show the functionality of Ghidra
4. Extract all the Javascript code

Assumptions

1. There is Javascript hiding inside Gootkit, and is a good source for IOCs.
2. The JavaScript files are probably compressed or encrypted.
3. The Password Grabber DLL is also embedded in this binary

Double-click the `rbody32.x.dec` inside of the project view and enjoy the 1337 dragon graphic animation. The answer to “would you like to analyze now?” is always yes.

 The Answer is Always Yes *Figure 2: An exercise in clicking the Yes button until something happens*

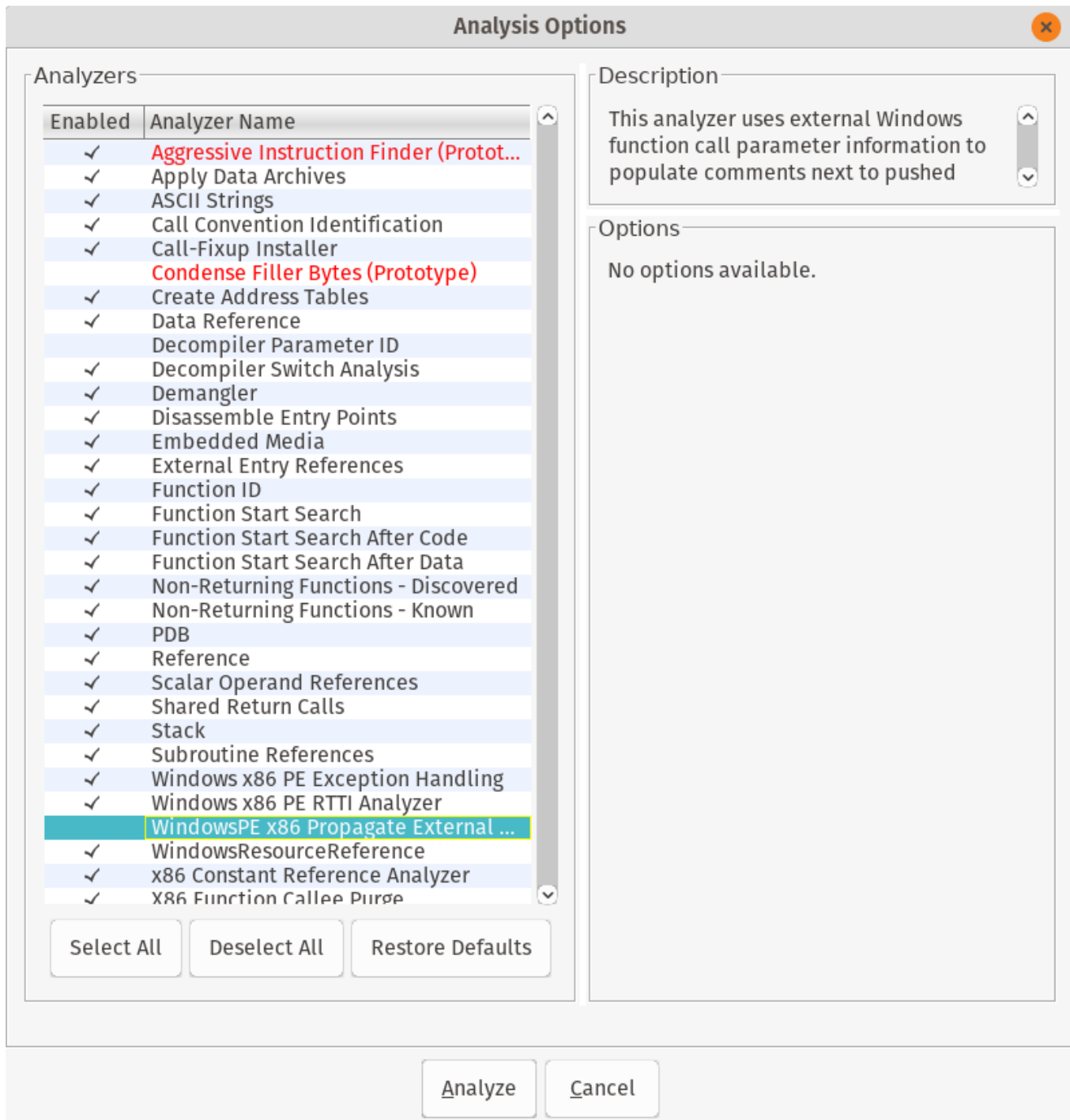


Figure 3: Be sure to select 'Aggressive Instruction Finder' and bravely ignore all the warnings.

Ghidra Analysis Options

Figure 3 shows the analysis options that Ghidra has available. Similar to IDA, you should most likely ignore these individual settings and just accept the defaults. (The exception being **Aggressive Instruction Finder**)

Looking at some of the default options, there are all sorts of goodies available. I'll go through my favorites so far:

1. Apply Data Archives - Search for embedded archive formats, and display information about them. Have a blob of zip/base64/lznt1 data you find? Ghidra looks for these as well and calls them out.
2. Embedded Media - More often than not, especially if your sample is trying to impersonate a benign program, you'll find media or other sheisty information embedded. This will create bookmarks for you to later use and analyze.
3. Windows .* - All of the internal things that Windows compilers use to make life difficult. Previously these all had to be waded through individually. Now Ghidra will figure them out, add salient information to the analysis, and generally save you time.

Hopefully in the time it took you to read the above, your analysis is finished. Let's jump right into analyzing the GUI and starting to use our workflow.

GUI Overview

After all the analysis is completed, you should be presented with the business end of Ghidra, it's GUI. Take in the Windows 95 era Java Swing GUI, and remember a time when you could hot-patch the page fault handler without the Windows kernel immediately labeling you as a malcontent.

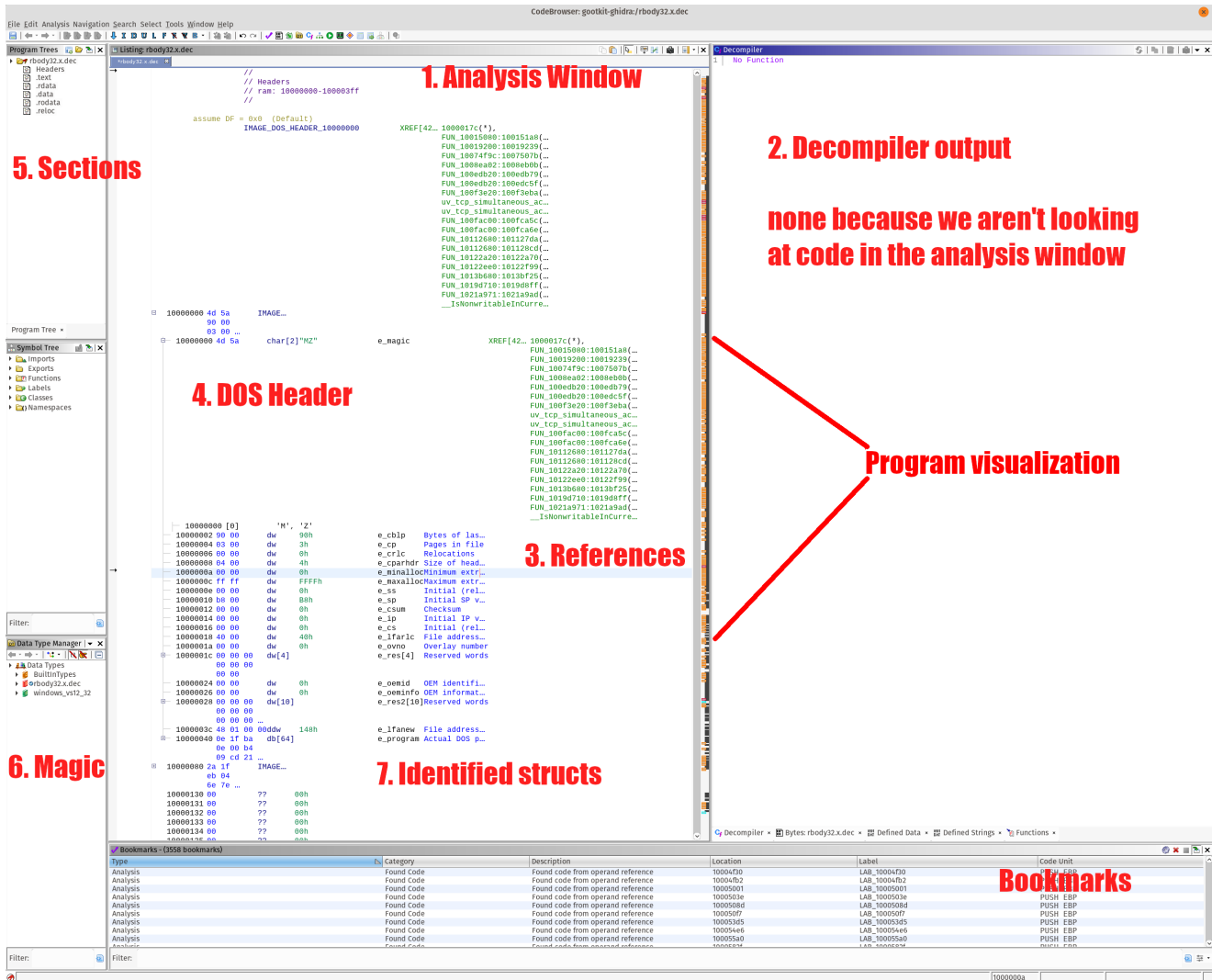


Figure 4: First view of the GUI with annotations. Clean version without the annotations can be found here

Enable Entropy Visualization

This is a cool trick that saved me a lot of time. Enable entropy visualization. Click the drop down menu on the top right of the Listing view, and select “Show Entropy.”

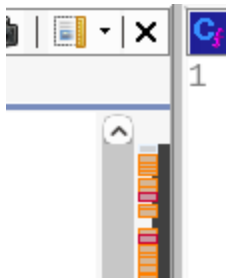


Figure 5: Click the pulldown to enable entropy visualization

Entropy, or the measure of randomness is useful for identifying encrypted or compressed portions of the executable. This is probably a good time for you to learn some math if you're not already familiar. [Wikipedia provides a good overview of Entropy](#) if you're into that sort of thing. All you need to know is that the higher the entropy (red in this case) means that there is likely a compressed, or encrypted blob of data. Goal 4 of our analysis goals is to extract the compressed JavaScript, so this is a good place to start looking.

Entropy does not always mean compressed or encoded data, nor does it mean that all encoded or compressed data is high entropy. All things being equal, it does mean something you should take a look at. In general, it's a good place to start looking and I appreciate that Ghidra includes this as a default option.



Figure 6: The code listing with the high-entropy portions

Analysis: Find the Embedded Code Part 1 - A Failure

Now that we have a good entropy visualization, let's try and take a shortcut to finding the compressed code.

Inspect the High Entropy Areas

If you click next to the red area in the executable, you should see a reference to the entropy being somewhere close to 8 in the tool-tip pop up. Select as close to the top as you can, then scroll the code view up until you see references to functions. Why functions? Because the address cross-references (XREFs) can contain random data, and not necessarily what you're looking for. Code references are where the executable is looking at that specific address. From here we will inspect all of the XREFs and look for anything that looks like encryption.

What does encryption code look like? This is a hard question. One way to answer that is to compile a bunch of encryption reference code, and look at what code is generated. In the end a couple of rules-of-thumb apply:

How to find encoding, encryption, and obfuscation the hard way

1. Is there an xor with differing operands? `xor eax, 0x42` would be an example, and `xor eax, eax` would not.
2. Are there lots of shift instructions in the same code? The `shl` and `shr` instructions being the most notable
3. There's a noticeable loop structure
4. Data is modified, and stored somewhere else in the program

With an eye on those details, I will inspect each of the listed cross references to see if I can infer what the compressed code is.

The first reference occurs at address 0x100f56f9 inside of FUN_100f56b0, and is a good example of what we are not looking for.

```

Listing: rbody32.x.dec
*body32.x.dec
100f56af ccc ??? CLD
.....
FUNCTION
.....
undefined FUN_100f56b0()
AL1 -RETURN-
undefined1 Stack[-0x_100af_1] XREF[2]: 100f56e8(W), 100f5767(R)
FUN_100f56b0 XREF[1]: uv_tty_init:100f5548(c...
100f56b0 51 PUSH ECX
100f56b1 83 3d CMP dword ptr [DAT_10567438],0x0 = ??
38 74
56 19 00
100f56b8 0f 85 JNZ LAB_100f5775
b7 00
00 00
100f56bc 66 0b MOV AX,word ptr [ECX + 0xb]
41 08
100f56c2 66 a3 MOV [DAT_104e7ccc],AX = 0007h
cc 7c
4e 10
100f56c8 66 85 c0 TEST AX,AX
100f56cb 75 0b JNZ LAB_100f56d8
100f56cd ba 07 MOV EAX,0x7
00 00 00
100f56d2 66 a3 MOV [DAT_104e7ccc],AX = 0007h
cc 7c
4e 10
LAB_100f56d8 XREF[1]: 100f56cb(j)
100f56d8 8a 0d MOV CL,byte ptr [DAT_104e7ccc] = 0007h
cc 7c
4e 10
100f56de 32 c0 XOR AL,AL
100f56e0 53 PUSH EBX
100f56e1 32 ed XOR CH,CH
100f56e3 0f b6 c0 MOVZX EAX,AL
100f56e6 32 db XOR BL,BL
100f56e8 89 6c MOV byte ptr [ESP + local_1],CH
24 07
100f56ec 32 ff XOR BH,BH
100f56ee ba 01 MOV EDX,0x1
00 00 00
100f56f3 f6 c1 04 TEST CL,0x4
100f56f6 0f 45 c2 CMOVB EAX,EDX
100f56f9 a2 ed MOV [DAT_104af3ed],AL = 07h
f3 4a 10
100f56fe f6 c1 02 TEST CL,0x2
100f5701 74 01 JZ LAB_100f570a
100f5703 0c 02 OR AL,0x2
100f5705 a2 ed MOV [DAT_104af3ed],AL = 07h
f3 4a 10
LAB_100f570a XREF[1]: 100f5701(j)
100f570a 84 ca TEST DL,CL
100f570c 74 07 JZ LAB_100f5715
100f570e 0c 04 OR AL,0x4
100f5710 a2 ed MOV [DAT_104af3ed],AL = 07h
f3 4a 10
Decompile: FUN_100f56b0 - (rbody32.x.dec)
1 /* WARNING: Globals starting with '_' overlap smaller symbols at the same address */
2
3 void __fastcall FUN_100f56b0(int iParam1)
4
5 {
6     uint uVar1;
7
8     if (_DAT_10567438 == 0) {
9         uVar1 = _DAT_104e7ccc & 0xffff0000;
10        _DAT_104e7ccc = uVar1 | (uint)(ushort *){iParam1 + 8};
11        if (*(ushort *){iParam1 + 8} == 0) {
12            [DAT_104e7ccc] = CONCAT22((short){uVar1 >> 0x10},7);
13        }
14        DAT_104af3ed = (_DAT_104e7ccc & 4) != 0;
15        if ((_DAT_104e7ccc & 2) != 0) {
16            DAT_104af3ed = (bool){DAT_104af3ed | 2};
17        }
18        if ((_DAT_104e7ccc & 1) != 0) {
19            DAT_104af3ed = (bool){DAT_104af3ed | 4};
20        }
21        DAT_10567431 = (_DAT_104e7ccc & 0x40) != 0;
22        if ((_DAT_104e7ccc & 0x20) != 0) {
23            DAT_10567431 = (bool){DAT_10567431 | 2};
24        }
25        if ((_DAT_104e7ccc & 0x10) != 0) {
26            DAT_10567431 = (bool){DAT_10567431 | 4};
27        }
28        DAT_10567432 = (_DAT_104e7ccc & 8) != 0;
29        _DAT_10567438 = 1;
30        DAT_1056743c = (_DAT_104e7ccc & 0x80) != 0;
31        DAT_10567433 = (_DAT_104e7ccc & 0x4000) != 0;
32    }
33    return;
34 }
35
36
  
```

Figure 7: FUN_100f56b0 assembly view and its decompilation

Rename Global Variables and Functions Using ADD

The first thing to do is to change the name of `DAT_104af3ed` to something more noticeable. Since reverse engineering is all about abductive reasoning, I'm going to assume (abduct) that this is compressed or encrypted code. If any facts present themselves that contradict this assumption, I will modify my assumption and subsequently change the variable name to match my new assumption. Abductive reasoning is a good lifestyle choice, but that's a highly personal matter. In the grand effort to increase global information entropy, confusion, and make a slightly offensive joke I call it Abductive Data Descriptor (ADD) workflow.

Gaze Upon the Magnificence of the Decompiler

You should notice that the decompilation window now has code in it. You may also notice that there are no `goto` s in this code. Further inspection will reveal that aside from automatically assigned labels, the code looks more or less reasonable. When I first reversed Gootkit with Ghidra and saw this decompilation, I had a very Jodie Foster in Contact moment when I first saw the decompiler working. Decompiler quality is informally judged by how many `goto` s produced instead of the more common if/else/switch/throw/catch statements. C and C++ developers are threatened from birth against using `goto` s, except in some very narrow circumstances, so a decompiler using them is akin to taking a shortcut. In practice I have found that once you fully fill out the types of all the variables, the decompiler outputs legible C code. Programming idioms and patterns matter, so it's a good idea to study them.

Let's rename a variable using our ADD workflow:

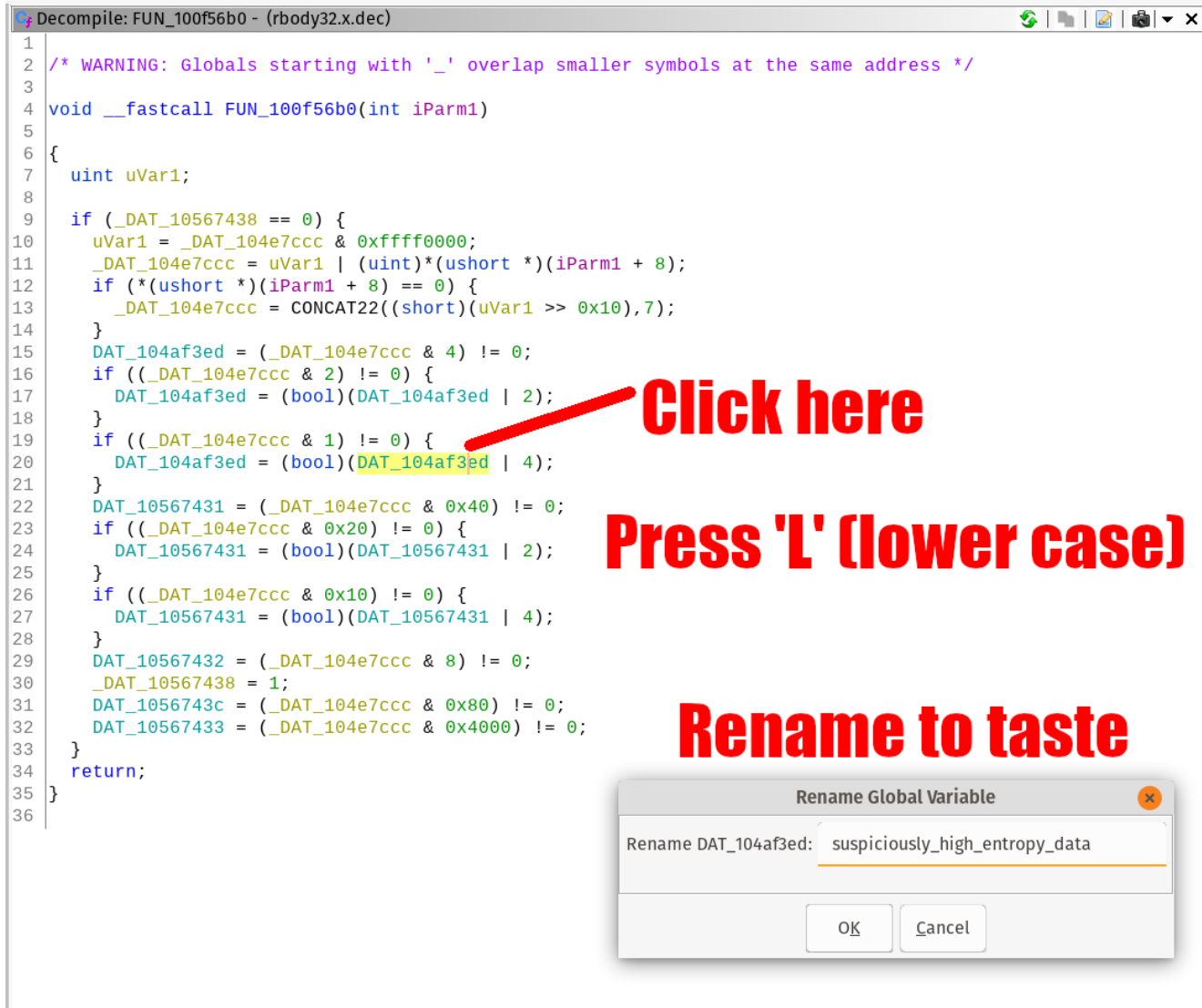


Figure 8: Rename the variable pointing to the high-entropy code to something more descriptive

Rename your Functions

This function is most likely *not* what we are looking for, however we have invested some time in looking at it. It's a good idea to rename the function any time you have a high-level concept you're looking for. My names tend to be pretty descriptive, and describe both my confidence in and the contents of the function. I use uncertain names like `some_xors_and_bitshifts` to imply how much time I've spent on it. Later I'll change it to something more specific if I spend more time on it, like `high_entropy_flag_mod()` and actually know what it's function is.

There is no xor instructions, and there is no loop. Likely this is a helper function that is looking at the flags of the data. It's a good idea to rename functions with your best guess (ADD), so I'm going to do that. I've also relabeled this function as `high_entropy_flag_mod()`.

Rename your variables

If you figure out the types used in a code sample, you can redefine those as well using `CTRL-L`, or right-clicking and selecting 'Retype Variable'. The more correct information you provide about the types, the more accurate the decompiler output will be.

Next function! To get back to the data view, click the left arrow button until you see the view again. This works similar to the `escape` key in IDA and Binary Ninja. If you renamed the function, your listing should look like this:

```
104af3ec 00      ??      00h
          suspiciously_high_entropy_data  XREF[4]: high_entropy_flag_mod:...
                                                high_entropy_flag_mod:...
                                                high_entropy_flag_mod:...
                                                FUN_100f7680:100f7696(...)
104af3ed 07      undef... 07h
104af3ee 00      ??      00h
104af3ef 00      ??      00h
104af3f0 00      ??      00h
```

Figure 8: The updated code listing once you have renamed the referencing function

Notice that all but one of the functions has been renamed, reducing how many functions you need to analyze. There is only one remaining, `FUN100f7680` and it bears inspection. The decompiler shows that a lot of our encryption qualifications are met: xors, bit shifting, and even a `do {} while ()` loop! Upon further inspection, the only xor in the code is at the very top of the function. This is a trick that Visual Studio uses to prevent stack based buffer overflows called a Canary. If you see an xor at the beginning of a function, this is most likely what it is. Similarly, there will be a subsequent function call that reverses the process, and exits the program.

Further inspection of the function shows that this is just a flag checking algorithm inside of a loop. Rename the function (I used `high_entropy_loop_flag_check()`) and move on. A good next step is to look at the XREFs for the function, and look at the parent code. I only saw one XREF `FUN_100f7bc0` so that is the next target.

Inferring Functionality using API Calls

The first thing I noticed about `FUN_100f7bc0` are the API calls being made. These function calls give us an idea about what the program is being used for. Looking up API calls on MSDN will give you an idea about what the developer is doing.

API Call (MSDN)

Typical Usage

[WaitForSingleObject](#)

Wait until the specified object is available or times out. Typically used to implement a Mutex, Semaphore, or other multiprocess primitives

API Call (MSDN)	Typical Usage
<u>MultiByteToWideChar</u>	Convert a multi-byte character to a 'wide' character. Unicode in Windows is full of pain and misery due to an early Windows design decision to ignore Unicode
<u>WriteConsoleW</u>	Write a buffer to the console. The W stands for 'wide'. An A at the end would indicate an ascii string
<u>GetLastError</u>	Why did my last function return an error? The Linux pattern is to use <code>errno</code> then bitterly complain about <u>reentrancy</u> issues

Table 1: A listing of API calls found in FUN_100f6bc0

Conclusion: This is OpenSSL

I quickly came to the realization that despite my initial hopes, this is not a decryption function. I follow a similar renaming process for all of the referenced functions, until everything is renamed. This particular branch of code seems to focus on outputting data to the terminal.

Sometimes you win, and sometimes you lose. I figured out I was in the wrong area when I scrolled a bit further down in listing and saw this jump out at me:

```

104f00ba 00      ??      00h
104f00bb 00      ??      00h

104f00bc e4 62    PTR_s_OpenSSL_DH_Method_104f00bc XREF[1]: FUN_1010e1a0:1010e229(...
          48 10    addr   s_OpenSSL_DH_Method_104862e4     = "OpenSSL DH Method"
104f00c0 f0 1f    addr   FUN_10101ff0
          10 10
104f00c4 b0 21    addr   LAB_101021b0
          10 10
104f00c8 40 23    addr   LAB_10102340

```

Since the implant portion of Gootkit is packaged Javascript with an embedded NodeJS server, which uses OpenSSL, this is likely just a statically linked copy of the OpenSSL code. In other words, a false lead.

Next Steps

In the next post, I will go over Ghidra's binary diffing feature and see if it can help identify embedded libraries.