Hacktober CTF 2018 – Binary Analysis – Larry

A basic reverse engineering challenge for a CTF and a mini intro to RE.

By **emtuls** in **Binary Analysis**, **Exploit Development**, **Reverse Engineering** on **October 19, 2018**

No comments

Hey Everyone! This past week, a couple of us from VetSec participated in the CTF called Hacktober. I would definitely say that it was a very good CTF for people who are beginner to mediocre at anything related to CTF's, so if you missed out and get a chance to try it next year, give it a shot!

Most of the challenges were very doable for people who were new with no experience, as well as a few challenges that would stump some seasoned players. It involved a very wide variety of challenges, such as the typical Forensics, Steganography, SQL, Binary Analysis, Web Exploitation, Trivia type challenges, as well as a few other uncommon types. This post will focus on one of the Binary Analysis challenges that I found to be the trickiest of the bunch and it could

take some time if attempted solely via **Static Analysis** without a combination of **Dynamic Analysis**.

Takeaways:

1. Basic Static Analysis Techniques for Binary Analysis.
2. Basic Dynamic Analysis Techniques for Binary Analysis.

Tools to have:

1. A **disassembler** with x64 (64-bit) and ELFcapability for **Static Analysis**, since that was what the challenge at hand was. I recommend IDA Pro Free (free version for this, since the paid is $$$$$$$). Another good option that I recommend would be **Binary Ninja** ($150), though you would need the paid version for this challenge, the demo one only supports 32-bit x86.
2. A **debugger** for **Dynamic Analysis**. I used **GDB**, a Linux debugger that comes standard on pretty much all distributions of Linux. I chose this one due to familiarity, though I'm sure other debuggers would work just fine.

Disclaimer:

I'll try not to make this a course on Computer Architecture, since that would be way too in depth for this challenge, but it would be good to have a small background in how a computers inner components work, particularly the processor. Again, this will not be too in depth, just a high level of the useful parts you will need to know for the challenge. I am by no means a professional at any of this, just merely a hobbyist/beginner who is learning as I go and probably wrong about a lot of things.

Where to start:

For most Binary Reverse Engineering challenges, I start on Linux and run '**strings**' just to see what hints I might get, which in this case showed a string called '**flag{thisNOTtheflag}**' and another '**flag{NOTtheflag}**'. Of course, I tried these and they were surely *NOT* the flag. I then typically run '**file**' against it, to see what it thinks the file type is, which gave me the following output:

```
Larry.out: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
```

From here, you can see that this is a **64-bit ELF file**, with **x86-64 architecture**. **ELF** is the standard executable file type for Linux, analogous to an .exe file for Windows. With this information, I select a disassembler that I

have on hand that might help assist me with the binary analysis. A **disassembler** is a tool that will take our compiled **binary** (ELF in this case) and convert it from the 1's and 0's our computer reads, to something that is more human readable. This readable format is known as **assembly language**, and in particular, we will be dealing with the **x86-64 assembly language**in the **Intel syntax**, versus AT&T syntax, for ease of readability.

Now, this is where it could get very confusing, since assembly language is not exactly straight forward. I do not intend for this to be an extensive x86 assembly language tutorial/course, so I will again, only limit this to the information required to solve this challenge.

x86 Computer Architecture:

Reference: x86 Assembly Tutorial

Some background on computer architecture that would be useful to know, at a high level, so bear with me:

In a processor, things are all dealt with inside of memory and passed around via something called a '**register**' and also sometimes pushed onto a '**stack**'. Each processor has their own set of **general purpose registers** as well as proprietary registers that are highly specific, but we will only worry about the main set of general purpose registers, specifically those in x86. These registers are as follows:

x86 Registers:

Resource: X86_Architecture and this

32 bit will start with E, which stands for **Extended**(it **extended 16 bit registers**, which follow the same convention, just **drop the E or R at the front, ie, AX, BX, CX, DX, SP, BP, etc**)

64 bit will start with R instead of E, which has no historical significance.

NOTE: For this, I will use mainly 32 bit registers, since this challenge mainly used 32 bit registers, though there are a couple of 64 bit registers used, which are denoted with R in the front.

EAX (32 bit Accumulator register): Used in arithmetic operations and also for **return values**from function calls in certain calling conventions, such as **cdecl** (this might be useful to help understand a bit more), a common convention in x86.

EBX (32 bit Base register): Sometimes used as a pointer to data. No specific uses, but is often set to a commonly used value (such as 0) throughout a

function to speed up calculations

ECX (32 bit Counter register): Used in shift/rotate instructions and loops as a loop counter (like i in for loops).

EDX (32 bit Data register): Used in arithmetic operations and I/O operations, and generally used for storing short-term variables within a function.

RSI/ESI (RSI is 64bit, ESI is 32 bit Source Index register): Used as a pointer to a source in stream operations (like manipulating strings, which are pointers to char arrays).

RDI/EDI (RDI is 64bit, EDI is 32 bit Destination Index register): Used as a pointer to a destination in stream operations. (also for manipulating strings, see above register).

EFLAGS – a 32-bit register used as a collection of bits representing Boolean values to store the results of operations and the state of the processor

Memory:

Although it is not needed for this challenge in particular, it would be valuable to know how memory works and the organization of it, such as little endian and big endian. In our case, x86 uses **little endian format.**

x86 Instruction Layout:

When dealing with x86 instruction with the **Intel syntax**, instructions are usually written in the form:

```
instruction destination, source
```

for example:

```
mov eax, [ecx]
```

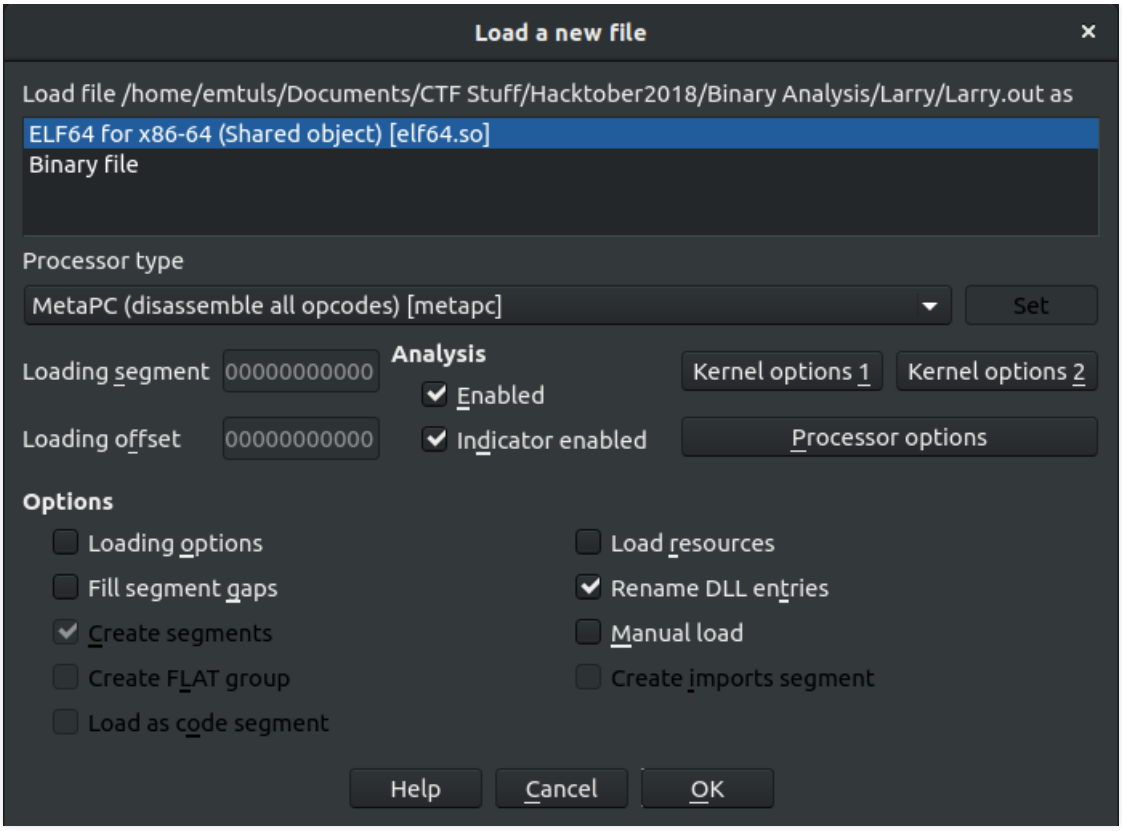
which means, move what is pointed to by the address in the register **ECX** and put it into the register **EAX**. If something is in **brackets** (like how **ECX**is), we use the value that is within these brackets **as a pointer to a memory address** and use the **value**that is **pointed to by this address**. We won't worry about this much right now, since this is not an x86 assembly language course, this was just an

example, but if you would like to understand it better, **this**might help. This format is almost always the case for Intel syntax, unless stated otherwise.

Static Analysis: IDA Pro (or your disassembler of choice):

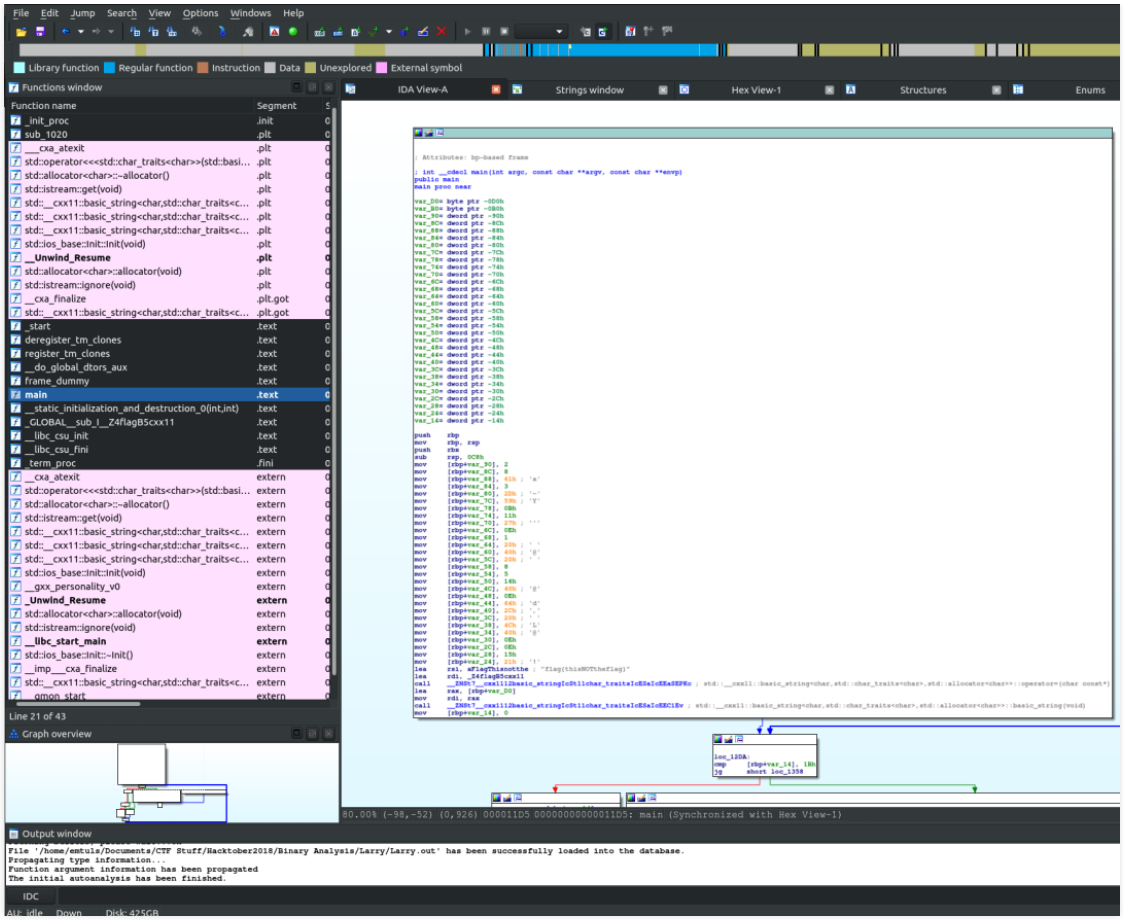
IDA is by far the industry standard when it comes to reverse engineering software. It has a plethora of capabilities and scripts combined with one of the best **decompilers** (sold separately) to date. A **decompiler** attempts to take our compiled binary and turn it into **source code** (what a developer/programmer would write), such as C programming, for instance. They are not perfect, but they can do a great job at assisting with reverse engineering. Though, for this challenge, we will stick with all free tools, so we will just be using the **IDA Pro Free version**, which is just a **disassembler**.

When we first open the file up in **IDA**, we are presented with a screen (shown in the image below) that asks us how to load the file, as in, which file type. IDA is pretty smart and it already recognizes that the file we provided it was an **ELFbinary** and is **64 bit,** so we will keep that selected. It then asks us which processor type we intend to use to decode the instructions in this disassembly. I recommend leaving this in **MetaPC**, since IDA is usually smart enough to determine this correctly.



Options when opening file in IDA Pro

After hitting '**OK**', IDA attempts to disassemble the program and leaves with a large jumbled mess of instructions and inside of the '**main**' function. This isn't an IDA tutorial, so I won't go too in depth as to how to actually use IDA to the full effect. Though if you are interested, you can definitely check out the book on it called, The IDA Pro Book. What you should see on your screen, should look similar to what is in the picture below.



Opening IDA on Larry.out

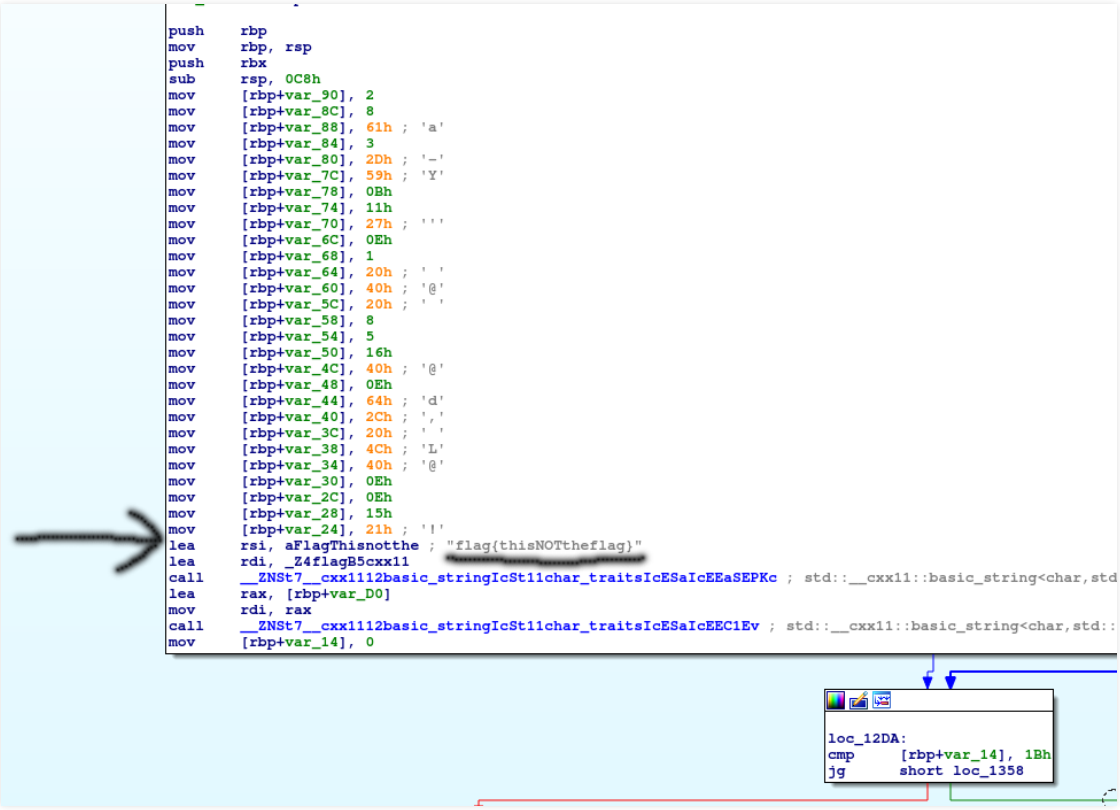
Getting into the disassembly:

The first screen we start in is the graph view, it allows us to see the flow of the program in a visual perspective, which I find very useful. To navigate on this screen, you can click and hold somewhere outside of one of the boxes and move the mouse around to shift the screen. You can also scroll up and down, or scroll in or out to view the bigger picture if you would like.

Focus! Don’t get too distracted:

Now, the main focus for reverse engineering is to not try and get into the fine details of every last line of assembly in the program (unless that is your actual intention of course), but rather, to find the details that matter to you. For our case, we are trying to complete a challenge, so we would likely want to find a function that handles/returns a ‘**flag**‘, which is sometimes a function labeled as ‘**win**‘, but not for this particular challenge. So if you remember, when I ran the ‘**strings**‘ command against the file, I saw a string that said ‘**flag{thisNOTtheflag}**‘ and another that said ‘**flag{NOTtheflag}**‘. We can confirm this by also running strings via **IDA Pro**, which can be found in **View -> Open subviews -> Strings** or ‘**Shift + F12**‘ as a shortcut. With this in mind, and an attempt to find something that might give me any idea of a ‘**flag**‘, I began

searching. What stuck out to me almost immediately was the string we saw before being stored into a variable. IDA is able to determine that the address in the variable ‘**aFlagThisnotthe**‘ points to an **ASCII string** which comes to be ‘**flag{thisNOTtheflag}**‘, and displays it for us as a comment automatically, as seen below.



Flag String in IDA

Now, for x86 assembly, the ‘**lea**‘ instruction means ‘**load effective address**‘. What this does is **store**the address of the **source operand** and put it into the **destination operand**, which if you recall from before, the format is:

instruction destination, source.

Function calls:

From the picture above, at the bottom right, we see ‘**lea rsi, aFlagThisnotthe**‘, which means that we load the address of the variable ‘**aFlagThisnotthe**‘ and store it into the ‘**RSI**‘ register. Then it looks like we store another variables address into the ‘**RDI**‘ register, from looking at the next instruction, which then leads to a **call**instruction. A ‘**call**‘ instruction means that we will be **calling a function** with the arguments given, and how those arguments are set up is based on the type of function being called and it’s **calling convention**. So with this knowledge, it looks to be **setting up arguments for a function**to work on them, which I originally had assumed would be a function that manipulated the string in memory. Though this assumption was **incorrect**, I will give a little

more detail into this. We know the address for the string, '**flag{thisNOTtheflag}**', was loaded into **RSI** via an '**lea**' instruction, and another address was loaded into **RDI,** thus, armed with this knowledge, we can assume that the '**call**' instruction located after these '**lea**' instructions, operates on these strings in some manner. It likely is some sort of memory allocation function for a new string. I won't go into any more detail on that, since again, it is not important for this challenge, and I already determined that this function is not the one we're worried about in the next few discoveries.

After that first function call, there is another set up of some function, which also gets called, but we won't worry about that for now, since it looks like the more interesting stuff comes after it. At the end of the box, we see a **0** being loaded into where '**[rbp+var_14]**' is **pointing** to (recall that if something is in **brackets**, this means we are **not worried about the value** which is evaluated by what is in the brackets, **but rather what that value as an address, points to in memory**), but this was only significant to me after I had looked ahead at the **next** box, which showed that **something was being compared to what was at at that location**, causing me to look back at what might be there. My assumption at this point is that it could be a **counter** that gets used for a **loop**, which is confirmed later, when we notice the large **bluearrow** that **circles** back around into the next box. When we see a **register** (typically **RBP**) has something **added to it** continuously, it typically signifies that we are dealing with a **string/array/struct** and **RBP** is pointing to the **baseof this structure**, with **whatever is being added to it** as the **offset** within this **structure**.

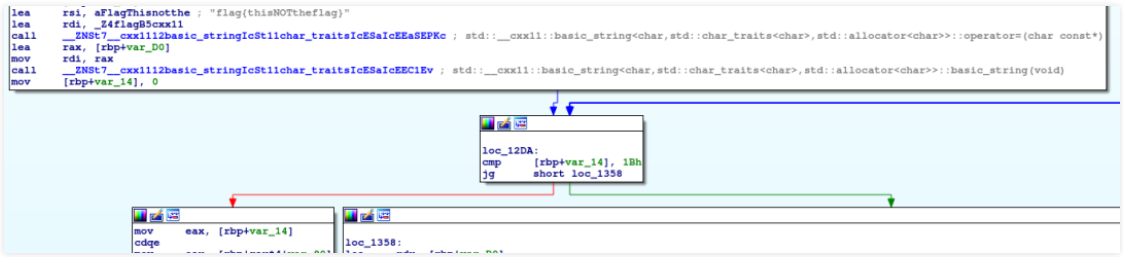
Comparison and Jump Instructions:

Now if we take a look at what the next set of blocks look to be doing, we can see a bunch of **arrows** jumping to other boxes, which symbolizes **code/control flow**. Let's examine this next box a little more. We can see that **2** different directions can be taken, but now we need to figure out **what** determines the path to take. In assembly, when it comes to just **2** paths, the path taken is determined by different **flags** that have been set and stored in the **EFLAGS** register, typically by a **comparison instruction** (such as **<, >, ==, <=, >=,**etc) **and a jump instruction** (such as **jg, jgr, jl, jle, je, jz,** etc). Based on this **comparison and jump flag**, you take either the **red** or the **green**arrow. The instruction that sets the flags in this case is the '**cmp [rbp+var_14], 1Bh**'. If you couldn't guess already, the '**cmp**' instruction means compare, and we're comparing the value '**1B**' as a **hexadecimal** value (which is what the '**h**' means after the **1B**) to the value stored at what is being pointed to by the address located at '**rbp + var_14**' (we can tell because when something is in **brackets**, it means a **pointer** to an **address**, that address being what is in the brackets).

With a **comparison** instruction, certain flags get set in the '**EFLAGS**' register, based on the outcome of this comparison. For example, the '**cmp**' instruction will do a **subtraction between the two instructions**, the **source** from the **destination**(**right side** subtracted from **left side** in Intel syntax) and set **ZF** (Zero Flag) and **SF** (Sign Flag) accordingly. If our subtraction leads to **0**, we will set **ZF** to **1** and the **SF** will be set to **0**. If the subtraction leads

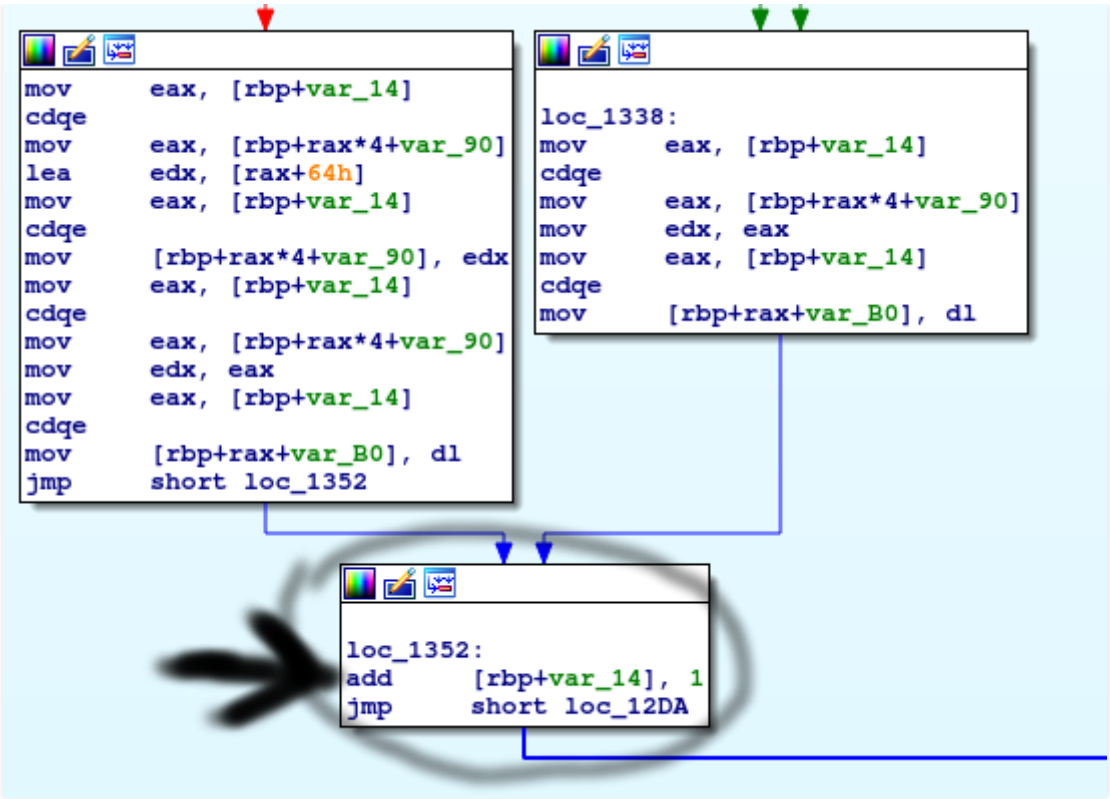
to a negative number (the **right**side was **less than** the **left** side), than we will have **SF** be set to **1** and **ZF** be set to **0**.

The next instruction is: '**jg short loc_1358**', at the bottom of the box. The '**jg**' instruction means '**jump if greater**', so it will take either the **red**arrow or the **green** arrow, based on the **flags** set in the prior **comparison instruction**. It takes the **red**arrow (meaning the jump is **NOT** taken) if the comparison showed that the **source** (right side) was **less** than the **destination** (left side) and the **green** arrow (meaning the jump **IS** taken) if the **source** was **greater** than the **destination**. What the arrows look like in IDA can be see in the picture below.



Comparison Instruction with Jump Instruction

When it comes to reverse engineering **functions**, when you spot an **interesting area** that you want to look into, it's usually best to **start at the end and work backwards**. After the block with the jump and comparison instructions, we see that if we take the **red** arrow side of code flow, we have a **blue**arrow that **loops back** around to the comparison box. This is analogous of a **loop** in code, like a **for loop** for example. Seeing this alongside the fact that we are comparing a number (**1Bh**) to a value that is pointed to by '**rbp+var_14**' (which initially started at **0**) but is **incremented by 1** just before **returning** to the start of the loop (as seen in the picture below) makes it safe to assume that we are **iterating** through something, likely an **array or string** of some sort. This loops continues until this iterator reaches '**1B**' in **hexadecimal** (**28** in **decimal**), in which we take the other path finally.



Iterator being incremented before returning to start of loop

The Turning Point:

Before we move onto the other path from this original comparison box at the top of this loop, let's reexamine the **bottom** of the **red** (**left**) side and try to gather some more information as to what this might be doing. Again, working backwards, we have something being loaded into a location '**[rbp + rax + var_B0]**'. This means that something is being loaded into the offset of '**var_B0**' inside of '**rbp**', at the iterator in '**rax**'. I stopped here and began my search to find what '**var_B0**' might be (when I reference '**var_B0**' or '**var_90**' by themselves, I am talking with respect to '**rbp + rax**'), since the rest of the box isn't really necessary to understand to complete the challenge, but if you are interested, you can continue to read the next paragraph, otherwise, **you can move on to the next section**!

NOTE: These next few paragraphs **are not needed** to solve challenge, but might be useful to get a better picture of what is going on. Starting at the top of the **bottom left box** in the loop (**left box in above picture**), we see that the **iterator** (**[rbp + var_14]**) is put into '**EAX**', which is then **multiplied by 4** (via '**RAX**', which if we recall, is the **64 bit version** of '**EAX**', so the same register, just **64 bit extended**, rather than 32 bit) and added to '**var_90**' (signifying that we are moving **4 bytes** at a time through whatever is at offset '**var_90**' inside of the structure at location '**rbp**'). Then, whatever is at this location is loaded into the '**EAX**' register. Now that we know that '**var_90**'

is some offset in a **structure** that points to some **string/array** potentially, let's move on and see what the end of this box leaves us with.

Continuing the quest to get a better understanding of what is in '**var_B0**', let's examine the rest of this box. Just before leaving the box, we can see that both boxes (left and right) end with a '**mov [rbp+rax+var_B0], dl**'. Well, what exactly is '**dl**' and what's in it? A '**dl**' is the **8-bit** version of **RDX/EDX** register, specifically, the **lower 8 bits** (hence the **l**), the **upper 8 bits** would be '**dh**'. I know that probably doesn't make a whole lot of sense to you, but again, since it **isn't** exactly necessary to solve this challenge if we go this route, I won't go into detail, but you can reference **this** for a better understanding, if you would like. Now, what gets put into '**dl**'? Since '**dl**' is a part of '**EDX**', we continue going back up and see that '**EAX**' is put into '**EDX**' via the '**mov edx, eax**' instruction. Now we continue going back up and we see that '**[rbp + rax*4 + var_90]**' is put into '**EAX**'. So from this, we can assume that we are **iterating** through a **string** at **offset** '**var_90**', **4 bytes at a time**, starting at the **base address** in '**RBP**'. From here, we have the option to now figure out what exactly is in '**var_90**' (most of which can be see in the first box that IDA provided, with the **28** '**mov**' **instructions**), or we can try to just see what we end up with inside of '**var_B0**', since it looks to be potentially useful to us. I won't continue with the search of '**var_90**', since it isn't exactly straight forward or needed for this challenge.

Wrapping up the Static Analysis:

With what looks to be the **important** part of the **red arrow's** code flow, we continue our search for what is inside of (pointed to by) '**var_B0**' by checking the **green arrow's** code flow. Below shows the **green arrow** flow after the **original comparison box**.

```
loc_1358:
lea     rdx, [rbp+var_B0]
lea     rax, [rbp+var_D0]
mov     rsi, rdx
mov     rdi, rax
call    __ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEaSEPKc ; std::__cxx11::ba
lea     rsi, aINeverThoughtI ; "I never thought I was special.\nThen, o"...
lea     rdi, _ZSt4cout@@GLIBCXX_3_4
call    __ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc ; std::operator<<<std:
lea     rsi, aGoodbye    ; "\nGoodbye.\n\n"
lea     rdi, _ZSt4cout@@GLIBCXX_3_4
call    __ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc ; std::operator<<<std:
lea     rdi, _ZSt3cin@@GLIBCXX_3_4 ; this
call    __ZNSi6ignoreEv ; std::istream::ignore(void)
mov     rdi, rax        ; this
call    __ZNSi3getEv    ; std::istream::get(void)
mov     ebx, 0
lea     rax, [rbp+var_D0]
mov     rdi, rax
call    _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEED1Ev ; std::__cxx11::basic
mov     eax, ebx
jmp     short loc_13E0
```

Green Arrow Code Flow