Reverse Engineering - Level 3

etherealshop

Step 0: Introduction

Background:

This is one of the first challenges in the ESCALATE program; it is designed to teach the fundamentals of reverse engineering to those with little exposure or experience with programming. The etherealshop challenge is designed to solidify the lessons learned in previous challenges, improve your skills with a disassembler, and retry your skills with a debugger and other tools used to disassemble an executable file and find the hidden or secured flag. This is a developmental step in understanding the contents of a binary file using a process called *live digital forensics analysis* or *dynamic analysis.*

Why Reverse Engineer?

During this exercise, an operator is in possession of a binary and has the need to better understand the application's access control function. **Live digital forensics,** also known as dynamic analysis, requires the use of a *debugger.* We will continue to hone our skills and systematic methodology for analyzing the design of an existing program or system. Whereas previous challenges required the operator to conduct static analysis, this challenge revisits skills needed for dynamic analysis, which is studying components of an executable while it is running to accurately determine how it uses memory in real time. An operator will leverage this type of analysis to bypass controls, discover vulnerabilities, manipulate the code for malicious purposes, or to otherwise redirect regular processes to facilitate their set objectives. Reverse engineering exercises like these, pick apart a .bin file to find a vulnerability allowing the operator to take additional steps to obtain a hidden and/or guarded flag.

Components needed:

Generated Challenge Piece: (generated in Step 1)

Disassembler: a computer program that translates machine language into assembly language.
(Popular options include)

Binary Ninja	IDA Pro	Radare2	Objdump

Windows Command line Emulator: a program that mimics a video **terminal** within some other display architecture. A **terminal window** allows the user access to a text **terminal** and all its applications such as **command-line** interfaces (CLI) and text user interface (TUI) applications not native to the host system. Popular options include:

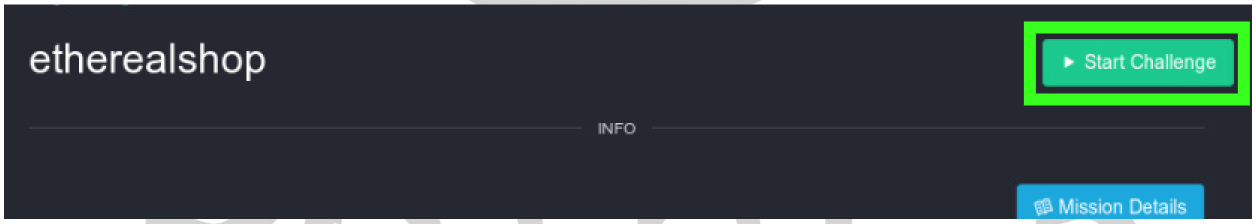
Wineconsole	cmder	ConEmu	Zoc Terminal

Debugger or Debugging Tool: a computer program used to test and pick apart other programs to find vulnerabilities, isolate bugs, and possibly leverage them while it is operating and executing code. Popular options include:

GDB	WinDbg	OllyDbg	Immunity

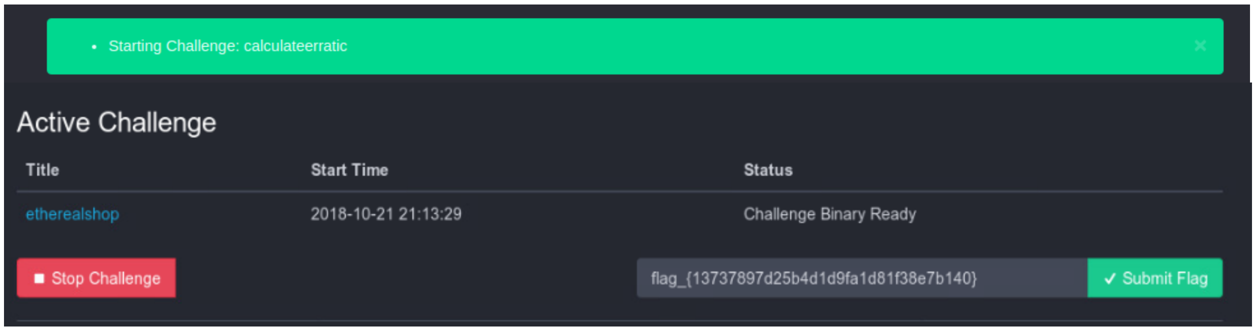
Step 1: Hit Start to Generate Challenge Piece

Click the green 'Start Challenge' button to get started.



2: Download and Save the Challenge Piece

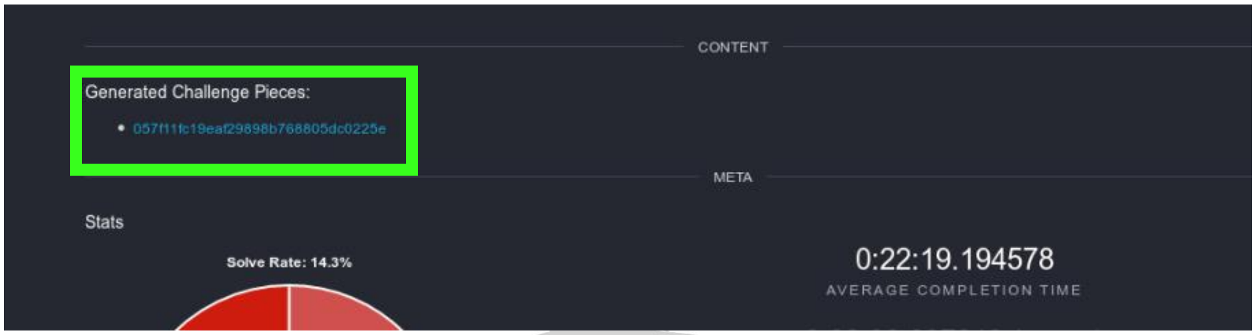
The binary file generates and the challenge page displays a green active banner.


- Starting Challenge: calculateerratic

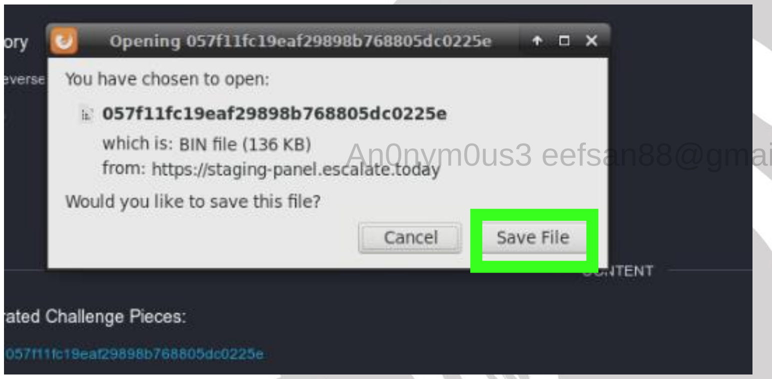
Active Challenge

Title	Start Time	Status
etherealshop	2018-10-21 21:13:29	Challenge Binary Ready

Scroll down to the 'Content' section and you will see the download link under 'Generated Challenge Pieces' .



Save the file to your Downloads folder.


Opening 057f11fc19eaf29898b768805dc0225e

You have chosen to open:

057f11fc19eaf29898b768805dc0225e

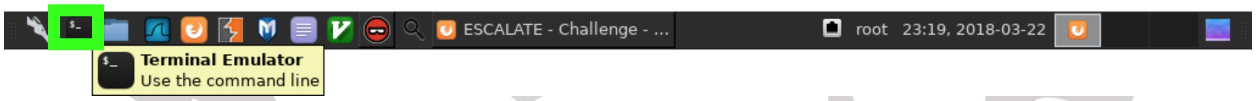
which is: BIN file (136 KB)

from: https://staging-panel.escalate.today

Would you like to save this file?

Step 3: Navigate to the Downloaded .bin File

Open the terminal by clicking the icon square icon with a '$_' in it. Within the ESCALATE browser VM, the terminal icon is located on the desktop's top banner bar.



What is the Terminal and Command line used for?

The Window, often called the command line or command-line interface, is a text-based application for viewing, handling and manipulating files on your computer. It is used to run common operations, like search, change folder, run, copy, and paste, but without using a graphical interface (GUI) like a search bar or mouse clicks. Other names for the command line are: cmd, CLI, prompt, console, or terminal.

Navigate to the downloaded .bin file to begin analysis. To do so, type the following command in the open terminal window and hit Enter.

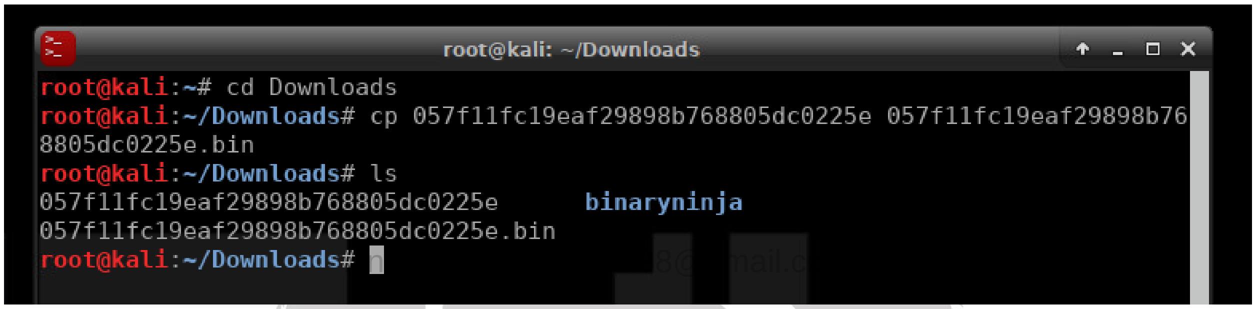
```
$ cd Downloads/
```

Step 4: Version Control the .bin Download

Maintain a pristine copy before starting the analysis. Copy the binary file we downloaded and give the copy we want to work with a .bin extension. To do this type in the open terminal

```
$ cp 057f11fc19eaf29898b768805dc0225e 057f11fc19eaf29898b768805dc0225e.bin
```

hit Enter. Using the `$ ls` command will list the contents of the Download folder to confirm the file was copied.



```
root@kali:~# cd Downloads
root@kali:~/Downloads# cp 057f11fc19eaf29898b768805dc0225e 057f11fc19eaf29898b76
8805dc0225e.bin
root@kali:~/Downloads# ls
057f11fc19eaf29898b768805dc0225e      binaryninja
057f11fc19eaf29898b768805dc0225e.bin
root@kali:~/Downloads# 
```

ⓘ The dollar sign is a character often used in online step-by-step instructions indicating the beginning of typed command within a terminal window and is not intended to be typed by the user. The pound symbol (#) is used in a similar fashion, but indicates that the command should be executed with root permissions.

Step 5: Determine File Type

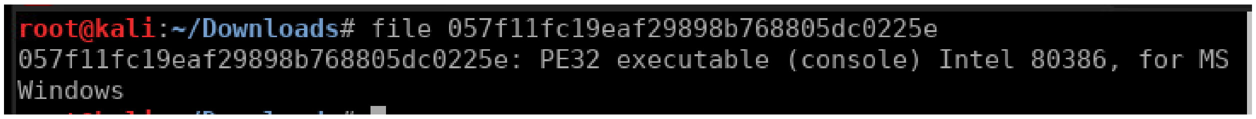
Next, we will take the first step in any reversing operation and determine what type of file it is by running the file command. To do this, we will simply type in the terminal window:

```
$ file 057f11fc19eaf29898b768805dc0225e.bin
```

What is the purpose of determining the file type?

Often identified by a file's extension (.doc, .pdf, .xls), determining the file type informs the operator in what environment the binary was designed to run and operate. This information can include both the Operating System (OS) and/or the application. The file type specifies how bits are used to encode information within the file. Formats can be proprietary or free and may be either unpublished or open, making file identification a valuable skill for future operations. It is necessary to identify the file type to conduct any further analysis.

As seen below, this operation shows us that the file is a PE (Portable Executable). PEs are the format of executable images commonly found on the Microsoft Windows operating system.



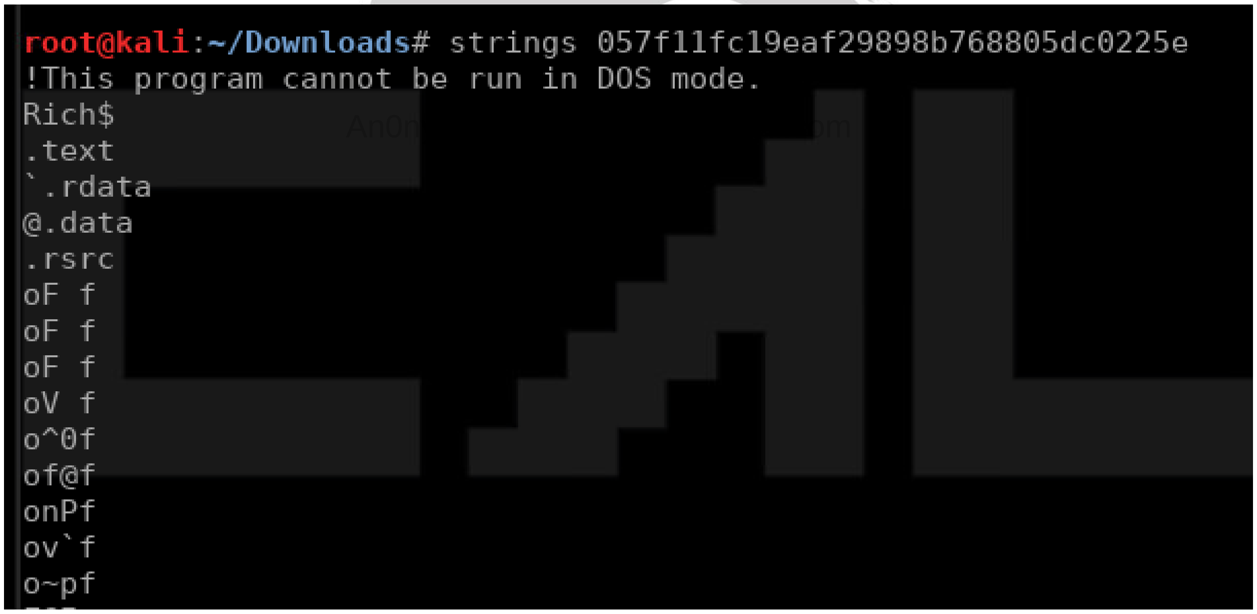
```
root@kali:~/Downloads# file 057f11fc19eaf29898b768805dc0225e
057f11fc19eaf29898b768805dc0225e: PE32 executable (console) Intel 80386, for MS
Windows
```

ⓘ The **Portable Executable** (PE) format is a file format for executables, object code, DLLs, FON Font files, and others used in 32-bit and 64-bit versions of Windows operating systems. The PE format is a data structure that encapsulates the information necessary for the Windows OS loader to manage the wrapped executable code.

Step 6: Parcing Strings

In earlier challenges, we used Binary Ninja to look for readable text strings inside a binary. Let's try another quick command to read a program's strings that produces an initial examination of a sample. This is the strings command in Linux and we can put it to use here by typing `strings 057f11fc19eaf29898b768805dc0225e.bin` into our terminal The output is a long list of the strings located in the program.

```
$ strings 057f11fc19eaf29898b768805dc0225e.bin
```



```
root@kali:~/Downloads# strings 057f11fc19eaf29898b768805dc0225e
!This program cannot be run in DOS mode.
Rich$
.text
`.rdata
@.data
.rsrc
oF f
oF f
oF f
oV f
o^0f
of@f
onPf
ov`f
o~pf
```

As you scroll down the list there are some interesting strings presented. Strings like <password>, You achieved level3!, You are not leet enough, and Compare String all may be something to track when breaking down the binary duirng static analysis.

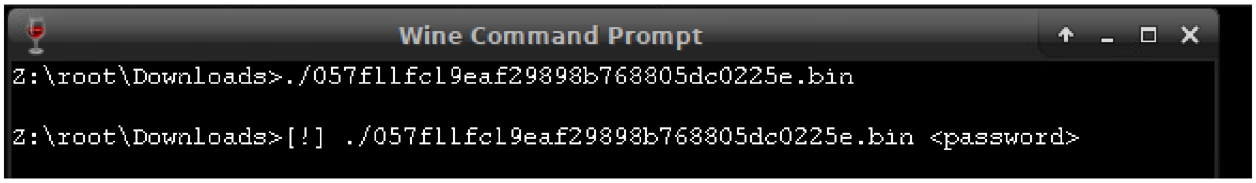
ⓘ A String is traditionally a sequence of characters that is human readable and included in the binary from the original source code. As the programmer identified variables, arrays, elements, and other structured bytes (or words) needed in programming the source code, those items were included in the compiled binary for its continued reference and operation.

Step 7: Analyze using appropriate OS

Since Windows executable files do not run natively in linux systems, we will use a tool designed to analyze foreign file types in incompatable hosts. To run the Windows executable file on Linux, open a windows command line emulator. To do this type `$ wineconsole` into a terminall window and hit enter. Run the file by typing in the terminal window:

```
$ ./057f11fc19eaf29898b768805dc0225e.bin
```

The program asks for a password and upon entering a random input, it reports that "You are not leet enough." Let's move on to static disassembly to break out the context associated with the strings we found in this step.



```
Z:\root\Downloads>./057f11fcl9eaf29898b768805dc0225e.bin

Z:\root\Downloads>[!] ./057f11fcl9eaf29898b768805dc0225e.bin <password>
```

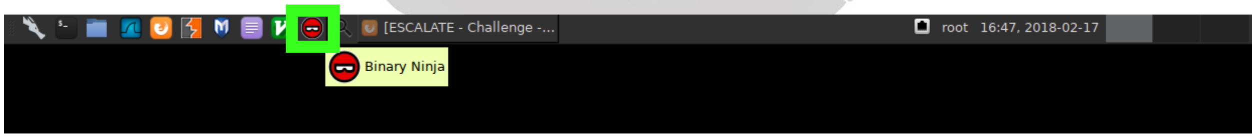
Step 8: Analyze the Binary Using a Dissassembler

Reverse Engineering basics

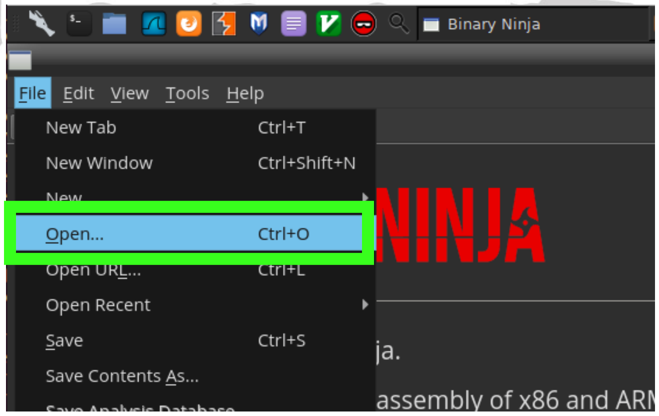
The next step requires the use of a disassembler which is a powerful tool used to do static analysis. The first step in any reversing operation is to identify the filetype (done!). A disassembler tool (binary Ninja pictured) is used to conduct the second step in any reversing project: finding and analyzing strings. Although the functionality and usage of this tool are vast, do not be intimidated; we will learn this tool one instruction at a time.

After identifying some of the program's attributes, begin the analysis to determine how the program verifies if the password you entered is correct or not. At this point, open a disassembler. This example will use open Binary Ninja,

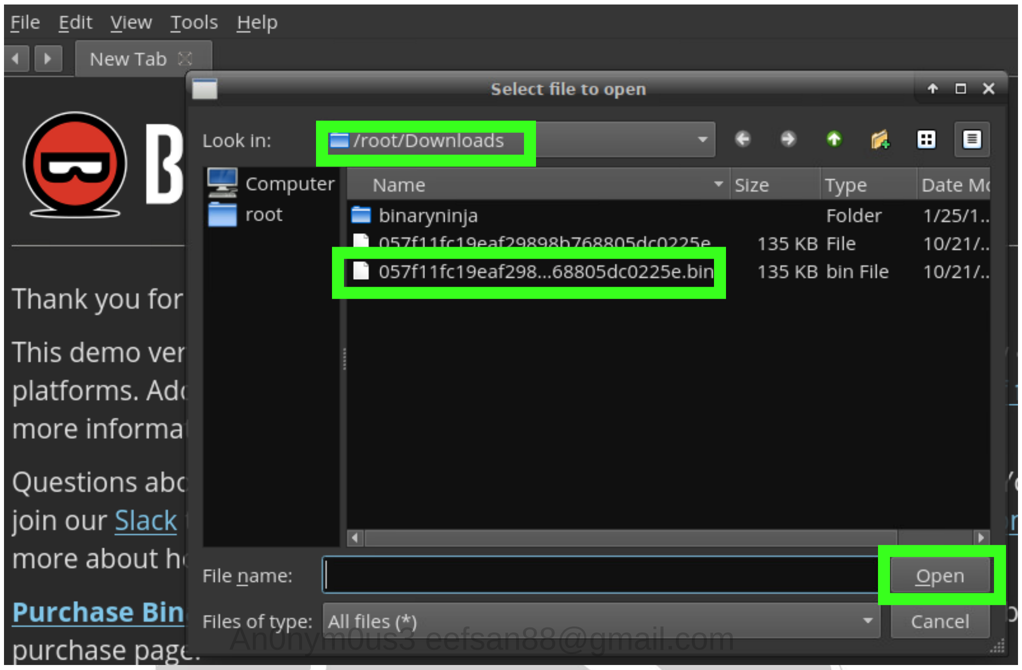
for this challenge. You can find the shortcut to open it at the top of your ESCALATE Kali desktop VM – it is the red circle with a sideways 'B' in it.



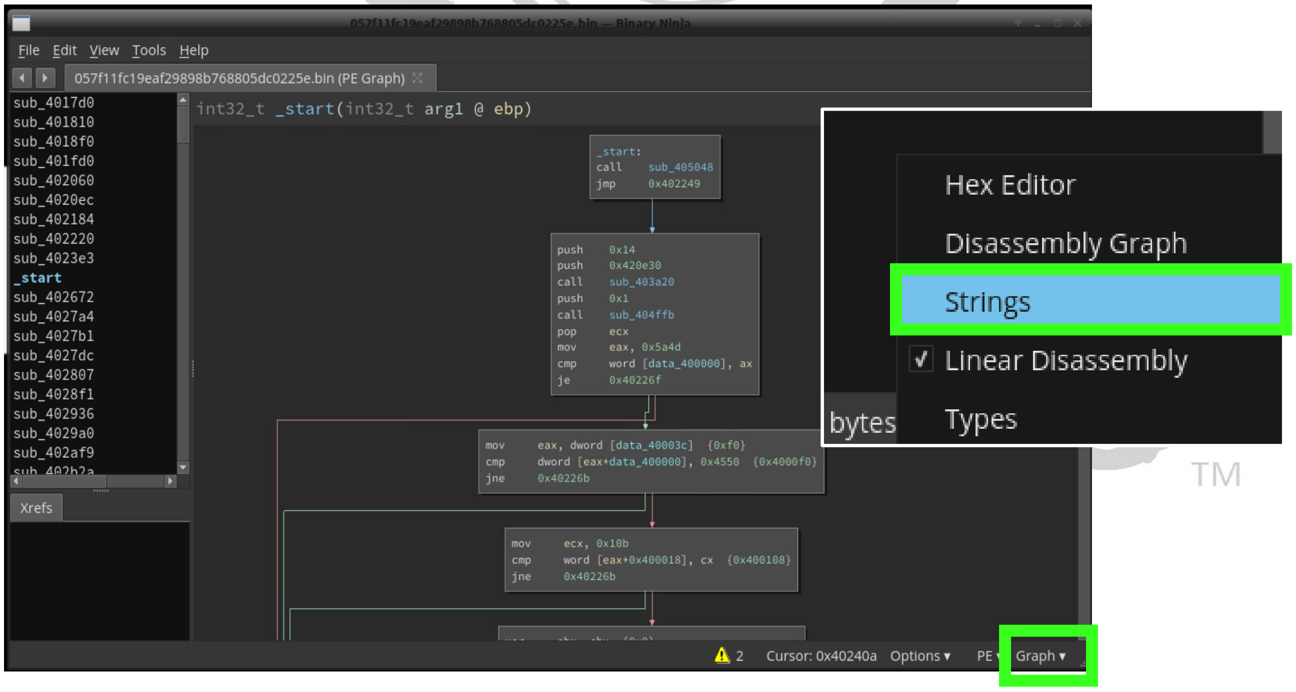
Now that we are in Binary Ninja let's open the binary via File > Open or Ctrl+O.



Navigate to Downloads...



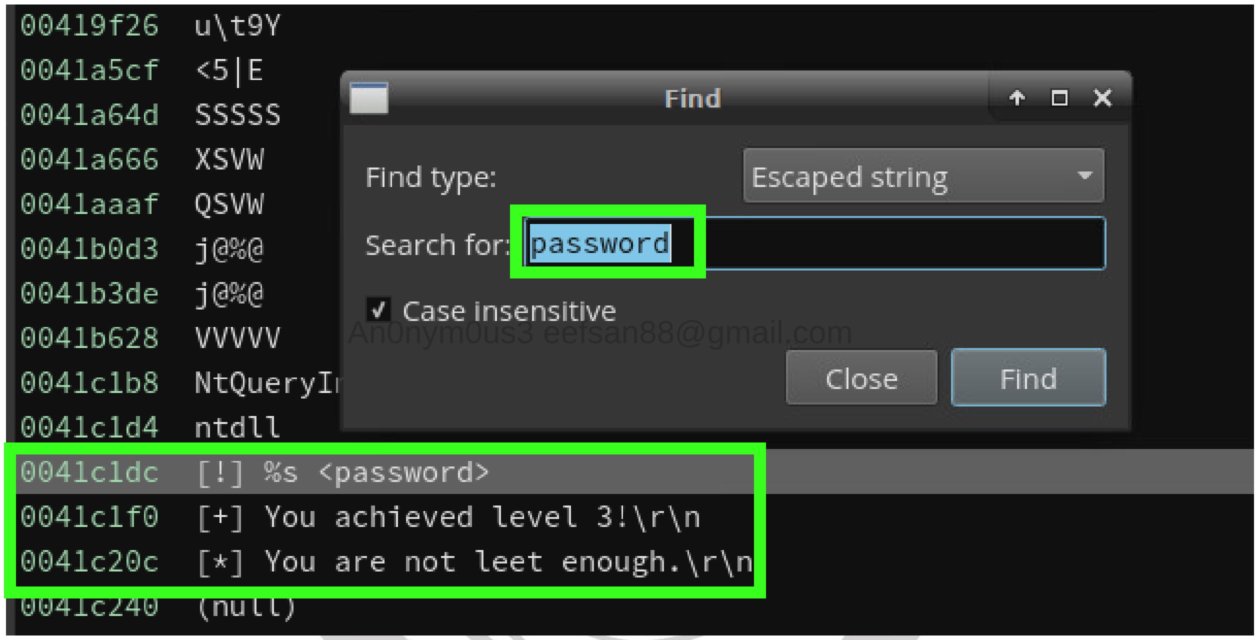
Open the previously saved 057f11fc19eaf29898b768805dc0225e.bin file in Binary Ninja.
Once the file is opened, Binary Ninja displays a Graphical representation of the assembly code (commonly referred to as disassembly). The binary file is divided into multiple parts, logically organizing the executed code. Let's review some of the interesting text or 'strings' that were identified in the previous step. To view, select the 'Strings' view, from the menu in the bottom right corner.



```
int32_t _start(int32_t arg1 @ ebp)

_start:
call    sub_405048
jmp     0x402249

push    0x14
push    0x420e30
call    sub_403a20
push    0x1
call    sub_404ffb
pop     ecx
mov     eax, 0x5a4d
cmp     word [data_400000], ax
je      0x40226f

mov     eax, dword [data_40003c]  {0xf0}
cmp     dword [eax+data_400000], 0x4550  {0x4000f0}
jne     0x40226b

mov     ecx, 0x10b
cmp     word [eax+0x400018], cx  {0x400108}
jne     0x40226b
```

In order for the binary to display messages to the user when ran, it is logical that the messages exist within the program. In the Strings view we can view all of the strings that Binary Ninja identified with in the program.

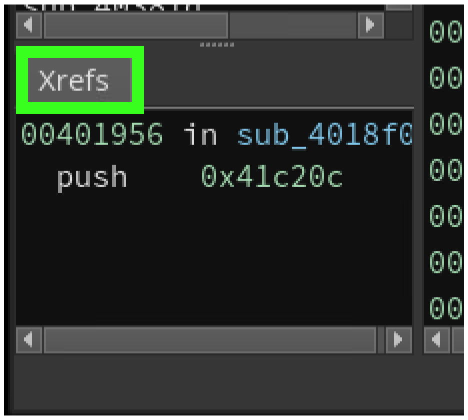
When the 'strings' command was run on the program in the previous step, we identified the string "<password>". We can scroll through the strings to find that message or press Ctrl+F and search for it directly. Since we know this is the message given after an incorrect password perhaps we can find what message is printed for a correct password nearby.


```
00419f26  u\t9Y
0041a5cf  <5|E
0041a64d  SSSSS
0041a666  XSVW
0041aaaf  QSVW
0041b0d3  j@%@
0041b3de  j@%@
0041b628  VVVVV
0041c1b8  NtQueryI
0041c1d4  ntdll
0041c1dc  [!] %s <password>
0041c1f0  [+] You achieved level 3!\r\n
0041c20c  [*] You are not leet enough.\r\n
0041c240  (null)
```

When we hit the 'Find' button, it locates the string in Strings view. There you see the address in memory where the String is stored as it waits for the program to reference it. When you highlight the string, you notice the Xrefs container in the lower left corner becomes populated with information. Let's chase down the string "You are not leet enough" to see if it has a checksum operation prior to denying access.

Reverse Engineering Basics

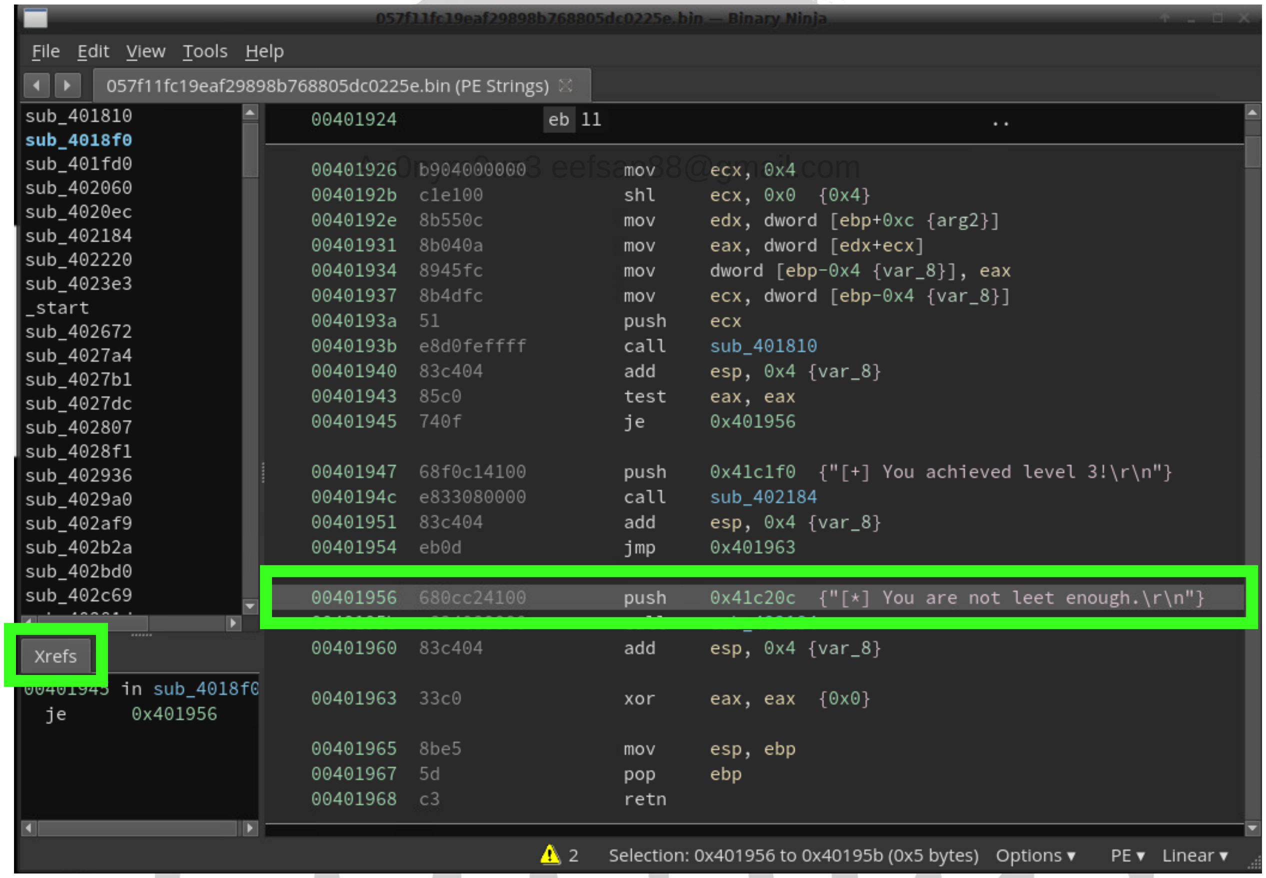
This next step requires the use of a function of the Binary Ninja tool called Xref, which Stands for cross reference. The Xrefs view in the lower-left shows all cross-references to a selected location, string, or variable. Note that the cross-references pane will change depending on whether an entire line is selected (all cross-references to that address are shown), or whether a specific token within the line is selected. In this challenge, the Xref will display all of the functions and locations within the program that reference that string and address location. Its like following breadcrumbs through a program.



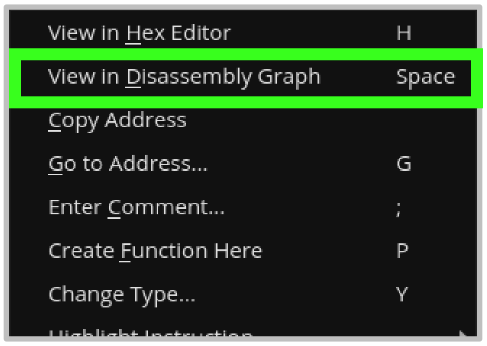
```
Xrefs
00401956 in sub_4018f0
  push     0x41c20c
```

Step 9: Cross-reference to Find the Function

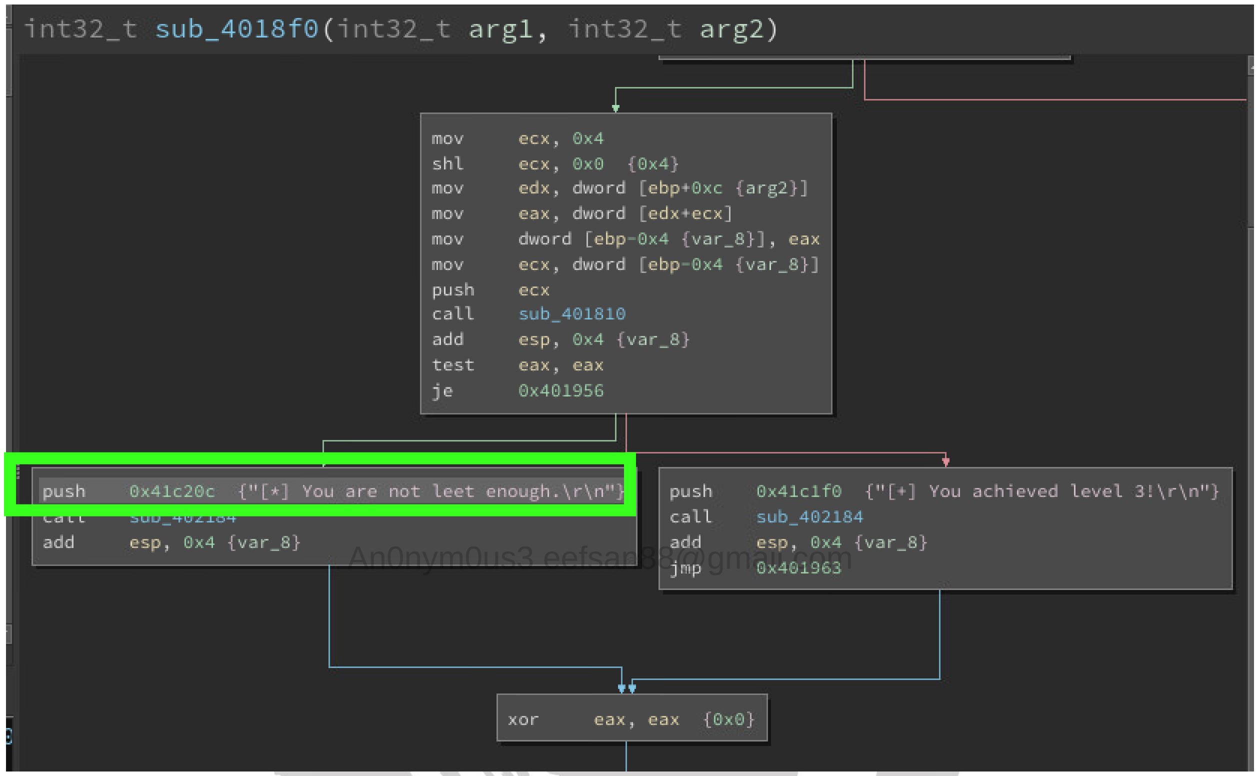
Double click on the data loaded into the Xrefs container. You should see the main container switch to the Linear view and the 'You are not leet enough' string should be highlighted. The binary file contains different sections-- some that contain the instructions to be executed and others that contain data such as the string 'You are not leet enough'. By finding the string 'You are not leet enough', in the data section, and then following the cross reference we are taken to the instructions that are executed.



```
057f11fc19eaf29898b768805dc0225e.bin — Binary Ninja
File  Edit  View  Tools  Help
057f11fc19eaf29898b768805dc0225e.bin (PE Strings)

sub_401810      00401924              eb 11                      ..
sub_4018f0
sub_401fd0      00401926  b904000000     mov     ecx, 0x4
sub_402060      0040192b  c1e100         shl     ecx, 0x0  {0x4}
sub_4020ec      0040192e  8b550c         mov     edx, dword [ebp+0xc {arg2}]
sub_402184      00401931  8b040a         mov     eax, dword [edx+ecx]
sub_402220      00401934  8945fc         mov     dword [ebp-0x4 {var_8}], eax
sub_4023e3      00401937  8b4dfc         mov     ecx, dword [ebp-0x4 {var_8}]
_start          0040193a  51             push    ecx
sub_402672      0040193b  e8d0feffff     call    sub_401810
sub_4027a4      00401940  83c404         add     esp, 0x4 {var_8}
sub_4027b1      00401943  85c0           test    eax, eax
sub_4027dc      00401945  740f           je      0x401956
sub_402807
sub_4028f1      00401947  68f0c14100     push    0x41c1f0  {"[+] You achieved level 3!\r\n"}
sub_402936      0040194c  e833080000     call    sub_402184
sub_4029a0      00401951  83c404         add     esp, 0x4 {var_8}
sub_402af9      00401954  eb0d           jmp     0x401963
sub_402b2a
sub_402bd0      00401956  680cc24100     push    0x41c20c  {"[*] You are not leet enough.\r\n"}
sub_402c69
                00401960  83c404         add     esp, 0x4 {var_8}
Xrefs
00401945 in sub_4018f0  00401963  33c0   xor     eax, eax  {0x0}
  je      0x401956
                00401965  8be5           mov     esp, ebp
                00401967  5d             pop     ebp
                00401968  c3             retn

⚠ 2   Selection: 0x401956 to 0x40195b (0x5 bytes)   Options ▾   PE ▾   Linear ▾
```



Let's try viewing this in Graph view to see if it helps us understand the string and function's purpose in the program. To do so, right click on the highlighted String we are investigating and select 'View in Disassembly Graph' or press the Spacebar key. The graph view groups the basic organizes disassembly into visually distinct blocks with edges showing control flow between them. By using this view, we see a graph representation on the flow of the program pictured below.



```
int32_t sub_4018f0(int32_t arg1, int32_t arg2)

mov     ecx, 0x4
shl     ecx, 0x0  {0x4}
mov     edx, dword [ebp+0xc {arg2}]
mov     eax, dword [edx+ecx]
mov     dword [ebp-0x4 {var_8}], eax
mov     ecx, dword [ebp-0x4 {var_8}]
push    ecx
call    sub_401810
add     esp, 0x4 {var_8}
test    eax, eax
je      0x401956

push    0x41c20c  {"[*] You are not leet enough.\r\n"}
call    sub_402184
add     esp, 0x4 {var_8}

push    0x41c1f0  {"[+] You achieved level 3!\r\n"}
call    sub_402184
add     esp, 0x4 {var_8}
jmp     0x401963

xor     eax, eax  {0x0}
```

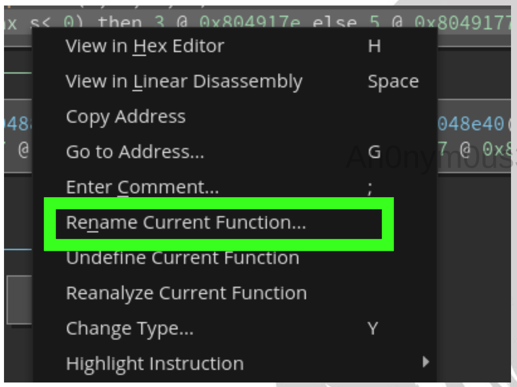
We see the string 'You are not leet enough' in a box on the lower left, and to the right we see 'You achieved level 3!'. This view displays there are two outcomes based on the functions occurring in the box above them. In a program, there are numerous logic constructs that allow for different execution paths. In this case, an if-statement would have been used which gives us two different branches the executable can follow. Now look at the block of code directly above the two outcomes. We see the following sequence at the end of the block call -> add -> test ->je. This series of events would come from code similar to the following:

```
if(sub_401810(arg1))
{
    print “[+]You achieved level 3!\r\n”
}
else
{
    print “[*]You are not leet enough.\r\n”
}
```

ⓘ An " if " statement is a conditional programming statement that, if true, performs a function or displays information. Above is the if statement used in this challenge, not specific to any particular programming language.

Reverse Engineering Basics

Like most disassemblers, Binary Ninja assigns a name to functions based on its location in memory. It is helpful to rename a function once you understand the purpose of a function (or even have a best guess). It's easier to understand the logic in a program if we change the name from sub_401810 to CheckPassword.

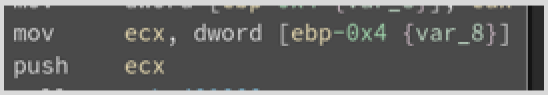

View in Hex Editor H
View in Linear Disassembly Space
Copy Address
Go to Address... G
Enter Comment... ;
Rename Current Function...
Undefine Current Function
Reanalyze Current Function
Change Type... Y
Highlight Instruction

To better organize your analysis, rename the function that we think contains an if statement to check the password. This will help us better track this function during ongoing analysis. To do so, right click the sub_401810 function, select 'Rename Current Function…', and type: `CheckPassword`

Let's take a second to familiarize ourselves with some programming basics.

Computer programming Basics

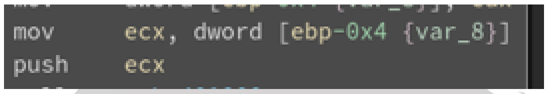
A program runs by executing numerous simple instructions. The instructions consist of bits of data like binary values or could also contain something to present to the user like the address in memory for a string of characters. Both the data and the instructions of what to do with that data are loaded into the stack to be executed in a specific order. This data and instruction are transported in registers. Once the data and instruction are executed, the register empties and is again available to carry out a concurrent operation in the program. This is what we see in assembly as we walk down the line of execution.



```
mov     ecx, dword [ebp-0x4 {var_8}]
push    ecx
```

Pictured here, you see that the instruction is to MOV (move) DWORD (which is 32 bits of data represented by the series in the brackets) into the register ECX. The next instruction then says to PUSH the ECX register and its newly obtained contents onto the stack to be executed. If you don't understand it completely it right now, that's alright. This program is designed to allow us to learn as we go.

Let’s continue our analysis on this function that we think is used to verify/check a password. Notice that immediately preceding the CALL for our newly named CheckPassword function (sub_401810), there is an instruction to PUSH ECX. On this operating system’s architecture (x86), arguments are often passed to functions by first PUSHing them to the stack prior to the function call. Since we just discussed how a register is loaded and pushed to the stack, we could figure out where in the process ECX got its value by tracing the last time that register was loaded with information. Logically, it would have to be the most recent time that register was filled with information since they are only loaded, used, and emptied in preparation for a following operation. If we review our operations in Binary Ninja, above that line, the mov ecx, dword [ebp-0x4 {var_8}] instruction sets ECX with whatever value is at EBP-0x4. This may seem like a lot of instructions, but PUSH, MOV, and CALL are the primary ones that we will get comfortable working with for a while.



```
mov     ecx, dword [ebp-0x4 {var_8}]
push    ecx
```

ⓘ x86 Registers

On x86, there are eight (8) general purpose registers that you will encounter. By convention, many of them have a purpose that is *sometimes* respected.

EAX - Stores return values from functions.

EBX

ECX - Is often used as a loop counter, or as the object in C++.

EDX

ESI - The "source" register for string instructions.

EDI - The "destination" register for string instructions.

ESP - Points to the top of the stack.

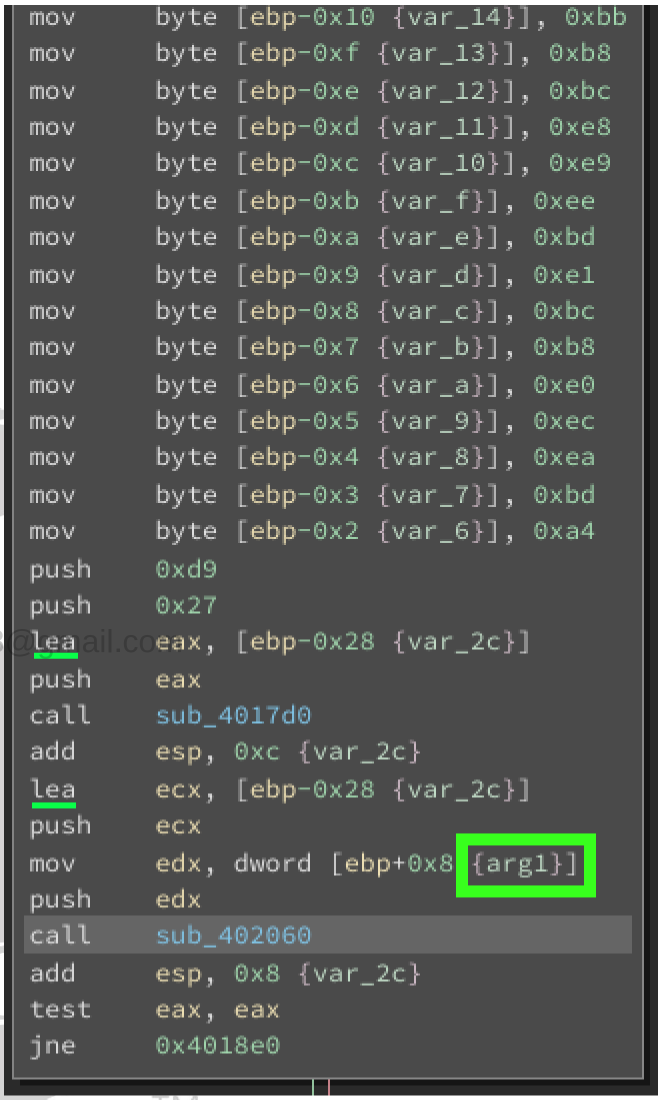
EBP - Often points to the base of the stack frame.

Additionally, we will sometimes speak about EIP. EIP is the program counter. It points to the next instruction the processor should execute. On x86, EIP is not a general-purpose register

Reverse Engineering Basics

Learning assembly language is like learning any other language: immersion is key! Luckily, instead of having to learn hundreds of verbs, there are only a few mnemonics to learn to be effective here. Try to focus purely on the CALLs to begin with. Whenever you see a CALL, try to figure out if there are arguments being passed to it by use of the PUSH instruction. If the argument being PUSHed is a register, look above to see what value was most recently MOVed into it.

Now, let's double click on our newly named function CheckPassword (sub_401810) which will take us into the internal function. We will begin our triage of this function by looking for the CALL instructions to determine flow of the program. By scrolling down in the Binary Ninja disassembler, you should see the following code pictured to the right:



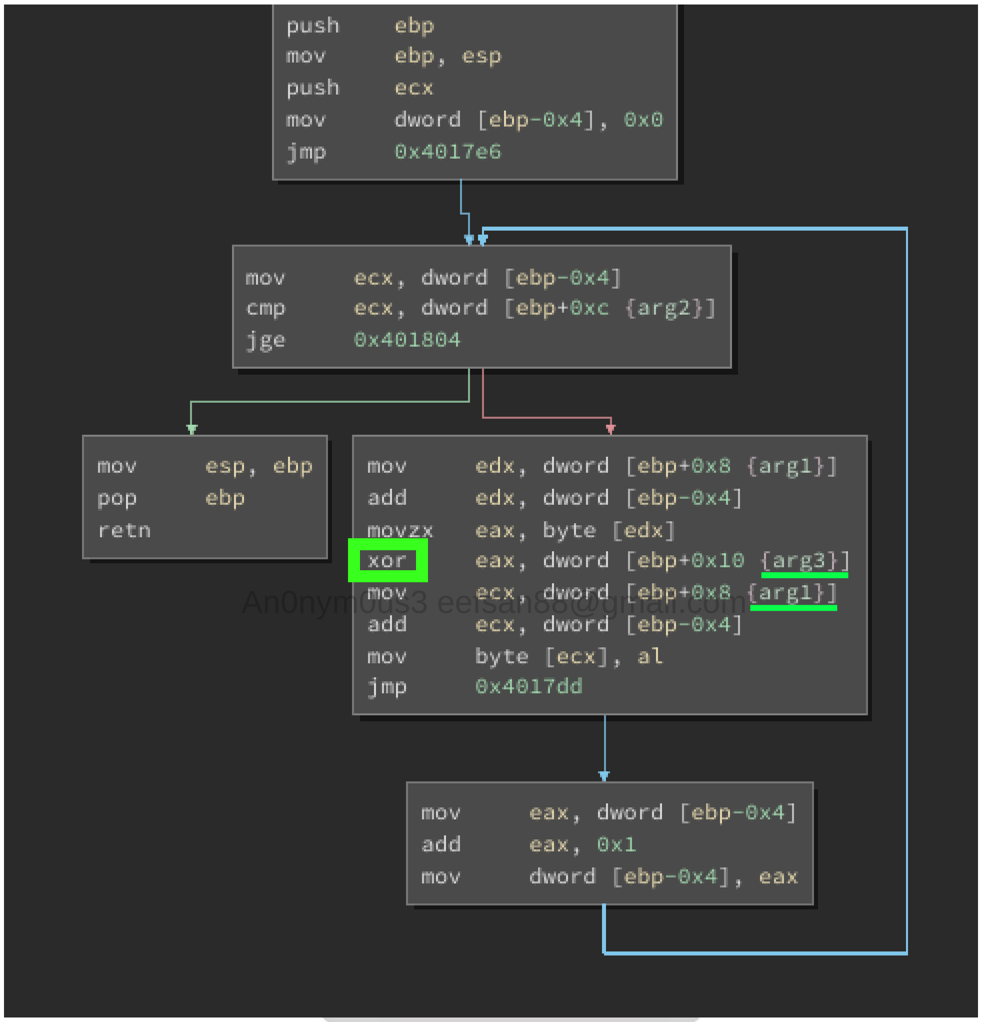
```
mov     byte [ebp-0x10 {var_14}], 0xbb
mov     byte [ebp-0xf {var_13}], 0xb8
mov     byte [ebp-0xe {var_12}], 0xbc
mov     byte [ebp-0xd {var_11}], 0xe8
mov     byte [ebp-0xc {var_10}], 0xe9
mov     byte [ebp-0xb {var_f}], 0xee
mov     byte [ebp-0xa {var_e}], 0xbd
mov     byte [ebp-0x9 {var_d}], 0xe1
mov     byte [ebp-0x8 {var_c}], 0xbc
mov     byte [ebp-0x7 {var_b}], 0xb8
mov     byte [ebp-0x6 {var_a}], 0xe0
mov     byte [ebp-0x5 {var_9}], 0xec
mov     byte [ebp-0x4 {var_8}], 0xea
mov     byte [ebp-0x3 {var_7}], 0xbd
mov     byte [ebp-0x2 {var_6}], 0xa4
push    0xd9
push    0x27
lea     eax, [ebp-0x28 {var_2c}]
push    eax
call    sub_4017d0
add     esp, 0xc {var_2c}
lea     ecx, [ebp-0x28 {var_2c}]
push    ecx
mov     edx, dword [ebp+0x8 {arg1}]
push    edx
call    sub_402060
add     esp, 0x8 {var_2c}
test    eax, eax
jne     0x4018e0
```

Let's convert the code around the two CALLed functions highlighted in blue back to a high-level representation like we have previously:

```
sub_4017d0(eax, 0x27, 0xd9)
if(sub_402060(arg1, ecx))
{
...
}
else
{
...
}
```

We see in the first call the first argument passed to the function sub_4017d0 is the value from the EAX register, this points to an address by the LEA command. There are two other arguments passed to this function as well, these are the values "0x27" and "0xd9". Then we see that Binary Ninja named arg1 for us representing the value passed in to the function sub_402060. Because the surrounding context tells us that a register was loaded to be compared with an arg1, which we haven't seen referenced previously, we can logically deduce that this is the value (argument) the user typed in on the keyboard. However, the second argument consists of the values from the ECX register, and this was set to an address by the LEA command. If we examine the values set to the EAX and ECX registers we see they contain the same address from the LEA commands. This is an interesting part of the code because the second function (i.e. sub_402060) is taking two parameters: our inputted password and some unknown value located at a memory address. The first function (i.e. sub_4017d0) is taking one parameter: an unknown value located at a memory address. Since the second function uses the password we supplied and the unknown value it is logical to assume this function must determine if the user has typed the correct password or not. One could make a logical leap here and assume that unknown value at the calculated memory address somehow contains the correct password (i.e. the flag), but that would be little more than an educated guess at this point. Next let's take a closer look at the first function and see what it is doing with the unknown value.

ⓘ Load Effective Address (LEA dest, src): Determines the effective address of the second operand (the source operand) and stores it in the first operand (destination operand). The source operand is a memory address (offset part); the destination operand is a general-purpose register.



```
push    ebp
mov     ebp, esp
push    ecx
mov     dword [ebp-0x4], 0x0
jmp     0x4017e6
```

```
mov     ecx, dword [ebp-0x4]
cmp     ecx, dword [ebp+0xc {arg2}]
jge     0x401804
```

```
mov     esp, ebp
pop     ebp
retn
```

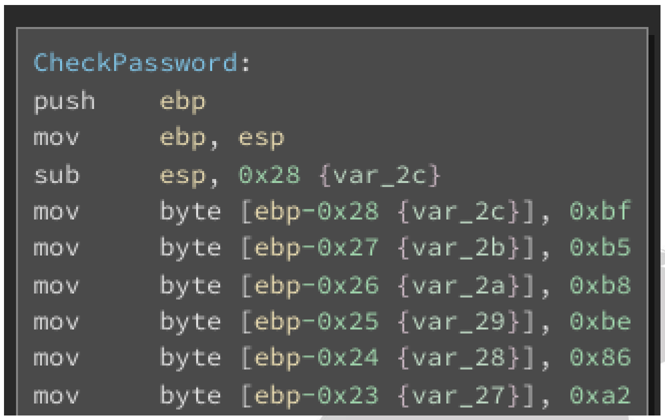
```
mov     edx, dword [ebp+0x8 {arg1}]
add     edx, dword [ebp-0x4]
movzx   eax, byte [edx]
xor     eax, dword [ebp+0x10 {arg3}]
mov     ecx, dword [ebp+0x8 {arg1}]
add     ecx, dword [ebp-0x4]
mov     byte [ecx], al
jmp     0x4017dd
```

```
mov     eax, dword [ebp-0x4]
add     eax, 0x1
mov     dword [ebp-0x4], eax
```

Let's convert the code from the first function pictured above into a high-level representation.

```
while(count < arg2){
    XOR byte of arg1 with arg3
}
```

After analyzing this function, we see that it loops over the values starting at the address passed in arg1, and continues until the counter reaches the value passed in arg2. The loop uses MOVZX to move one byte from the address location into the EAX register. The binary value in EAX is then XOR'd with the binary value from arg3. It may not be clear, however what is happening here is the password has been obfuscated before storing it in the program. Before the second function compares our password entered with the valid password, our flag, it needs to be de-obfuscated. This function is getting each byte of the password and converting it to the correct value that matches the ascii character.


```
CheckPassword:
push    ebp
mov     ebp, esp
sub     esp, 0x28 {var_2c}
mov     byte [ebp-0x28 {var_2c}], 0xbf
mov     byte [ebp-0x27 {var_2b}], 0xb5
mov     byte [ebp-0x26 {var_2a}], 0xb8
mov     byte [ebp-0x25 {var_29}], 0xbe
mov     byte [ebp-0x24 {var_28}], 0x86
mov     byte [ebp-0x23 {var_27}], 0xa2
```

Let's look at how we would get the first byte 0xbf to the correct ascii value. The first byte in hexadecimal is 0xbf, the value of arg3 which was determined above is 0xd9 as it was passed into the function sub_4017d0. The XOR function is described in the table below. It requires the operator to break any ASCII or HEX value down to its binary value in order to conduct the next step outlined in the table and example below.

ⓘ XOR

XOR, exclusive or, will set a bit to 1 if and only if the two input values are not the same.

Input		Output
A	**B**	
0	0	0
0	1	1
1	0	1
1	1	0

Work the first character below by taking the binary value of 0xbf (first value passed to build the password) and XOR it with 0xd9 (passed to the sub_4017d0 as XOR).

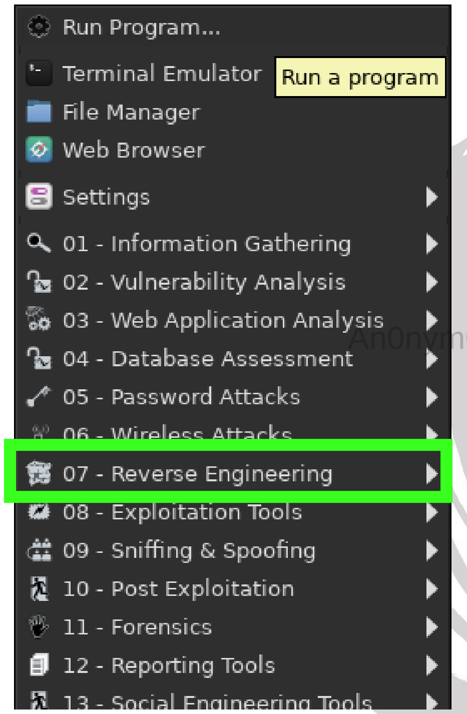
0xbf	**1**	**0**	**1**	**1**	**1**	**1**	**1**	**1**
0xd9	1	1	0	1	1	0	0	1
XOR Result	0	1	1	0	0	1	1	0

The XOR result creates a new binary of 01100110; which is 0x66 in hexadecimal. If we look this value up in the ASCII table we see that the ascii value is f.

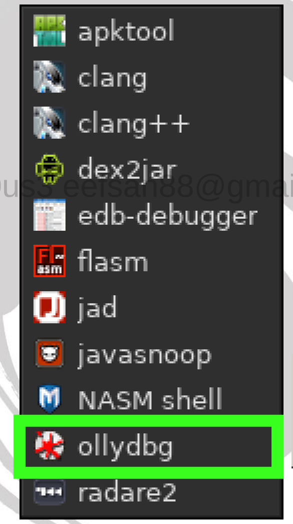
We could continue converting the byte values from the check password function and at the end we would have our flag. However, lets switch gears and finish this in a debugger. Make note that the second function is sub_402060.

Step 10: Further Debugging of the Identified Function

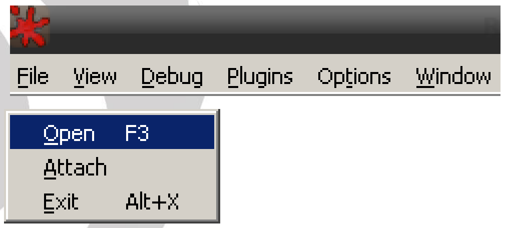
In order to confirm our hypothesis that the flag is being passed into sub_402060 and XOR'd to create the password, we need to analyze what is entered into the EAX register at the point when it is compared to the user's input while the program is running. To do this we need to conduct live digital forensics, also known as dynamic analysis. In this challenge we will use Ollydbg available in the ESCALATE browser VM and free to download online. Other debuggers listed in the materials section of this tutorial will also work. Open Ollydbg, or another Windows debugger by clicking the applications menu and then select "Reverse Engineering"



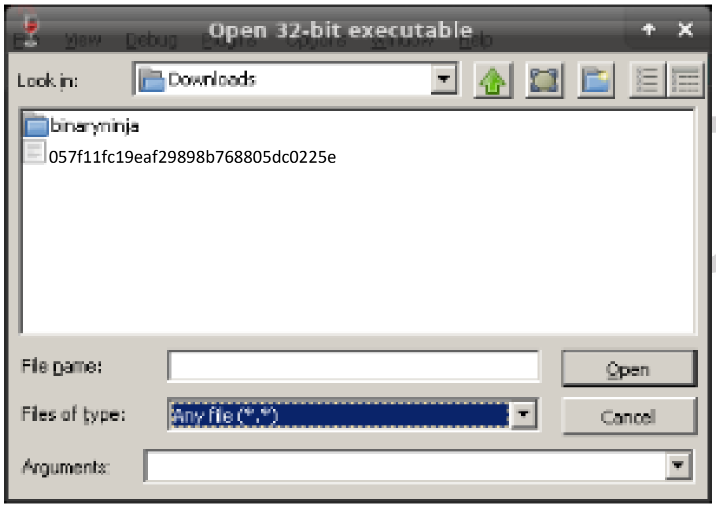
This will open the submenu that contains Ollydbg, select the program to open it.

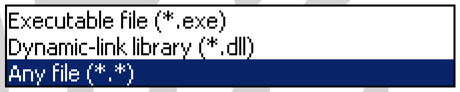


Then select File > Open or hit F3

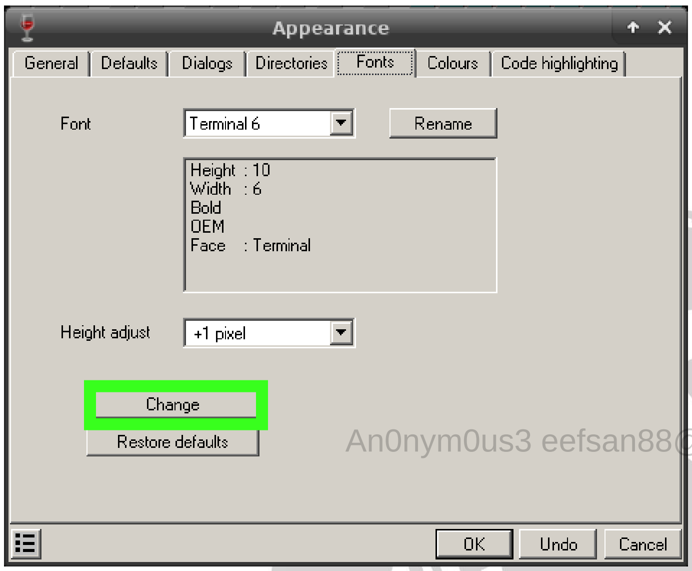


Change Files of type to Any file (*.*) and choose the file '057f11fc19eaf29898b768805dc0225e.bin'





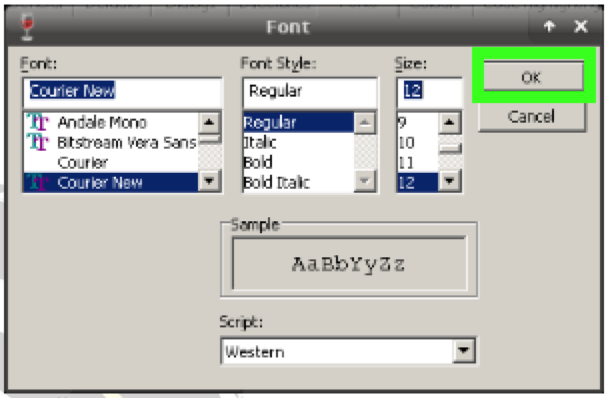
Olly debugger is another powerful tool with many capabilities. Although learning a new new may be intimidating, simliar to earlier challenges, we are learning new skills one step at a time. Let's start getting a handle on this tool by adjusting the display of the program so it presents what we see in a readable way. Once the Olly debugger is open, select Options > Appearance > Fonts > Change. Then select a font and font size that works for you.



Font: Terminal 6

```
Height  : 10
Width   : 6
Bold
OEM
Face    : Terminal
```

Height adjust: +1 pixel



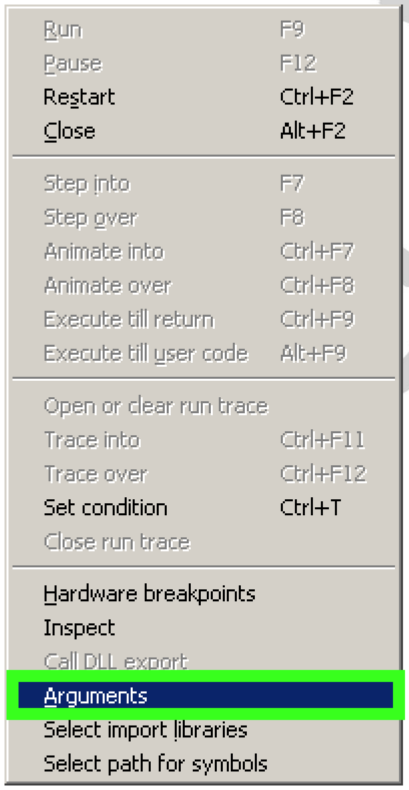
Font: Courier New (Andale Mono, Bitstream Vera Sans, Courier, Courier New)

Font Style: Regular (Regular, Italic, Bold, Bold Italic)

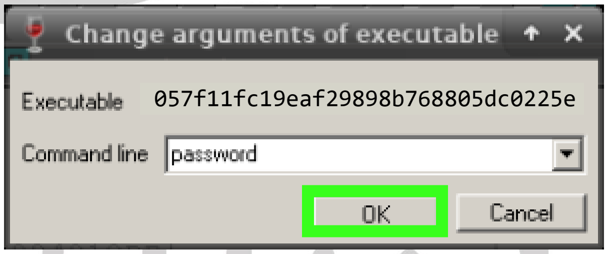
Size: 12 (9, 10, 11, 12)

Script: Western

Debuggers have the ability to give a program inputs or arguments so the operator can watch the program operate as if it were engaging with a user. To do so there are inputs or arguments that our program requires like a password. Let's give Olly debugger the argument (password) required to interact with our binary. Select Debug > Arguments, enter the password into the command line box and select ok. You will receive a warning about restarting. To restart select the debug menu and click Restart.



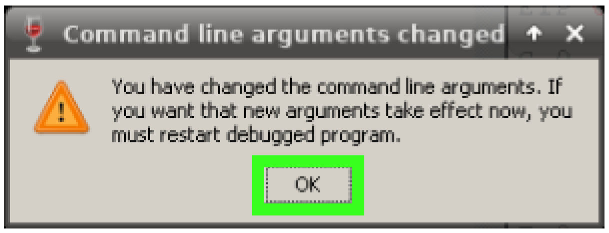
Enter anything into this box and hit ok.



Executable 057f11fc19eaf29898b768805dc0225e

Command line password

You will get a warning that you will must restart the debug process fo the change to take effect.



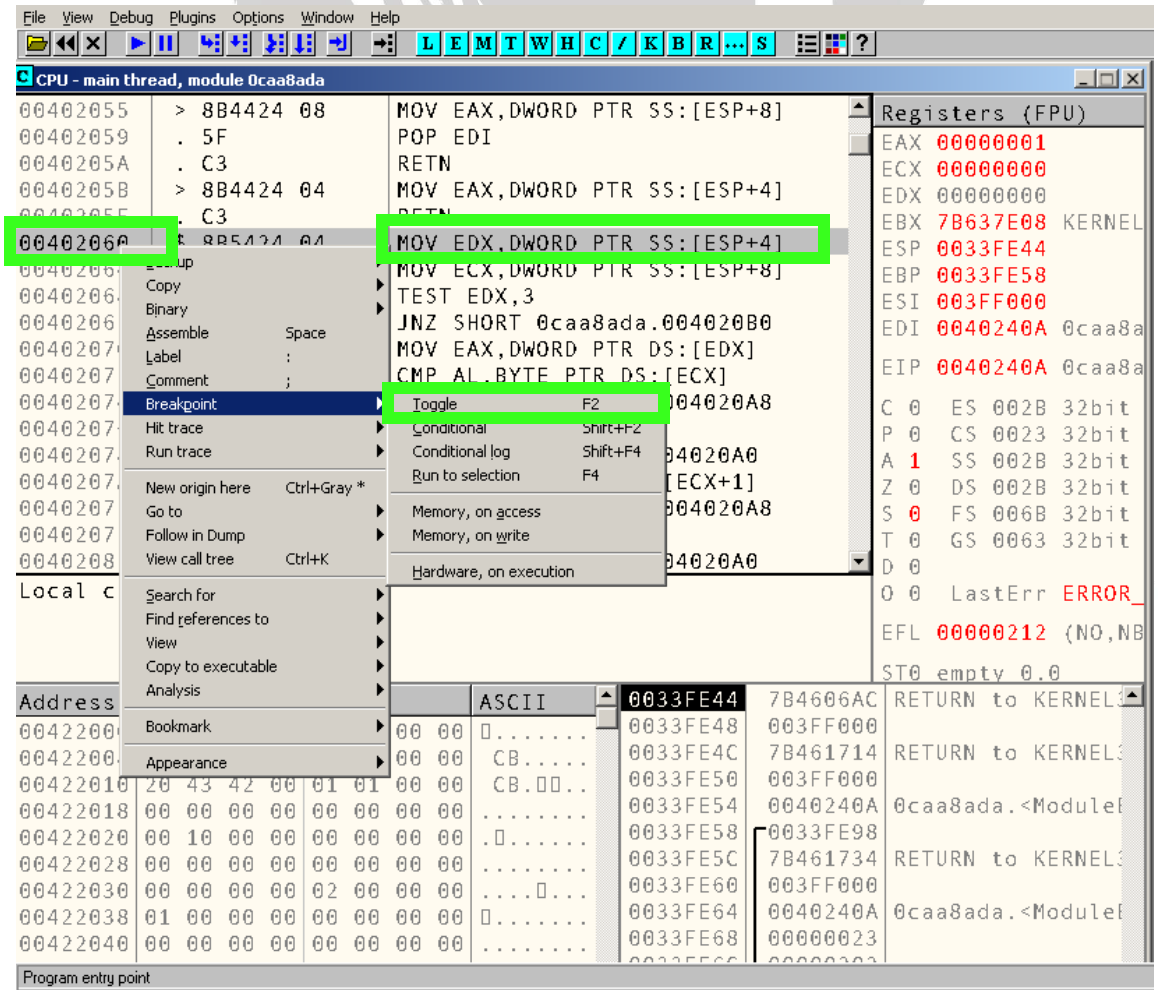
You have changed the command line arguments. If you want that new arguments take effect now, you must restart debugged program.

Reverse Engineering Basics

As previously mentioned in the tutorial, the operator has the need to dump the contents of a register, address, or variable while the binary is running. To accomplish this, we will need to dynamically set a *breakpoint* in the program. A breakpoint is an intentional stop marked in the code of an application where execution pauses for debugging. This allows the operator to inspect the internal state of the application at that point.

A breakpoint helps to speed up the debugging process in a large program, allowing the program to run up to a desired point where the operator wants to dump registers, adjust values of variables, or determine when a memory address is accessed. It is also possible to "step through" a program's code on a line-by-line basis executing one operation at a time, but breakpoints are more efficient.

On the left hand side of the screen you see numbers in the 00402000 range as simliar to our Binary Ninja analysis. Let's set a breakpoint on the address of the CALL sub_402060 where we hypothesize the flag is built and XOR'd. Remember, our goal is to see the value located at the memory address being passed into the function. From looking at Binary Ninja, we can see the address is: 00402060. To set the breakpoint, press Ctrl + G and enter the address 00402060 and hit "Go". Verify the disassembly matches what you see in Binary Ninja, then right-click on the address, select Breakpoint → Toggle.



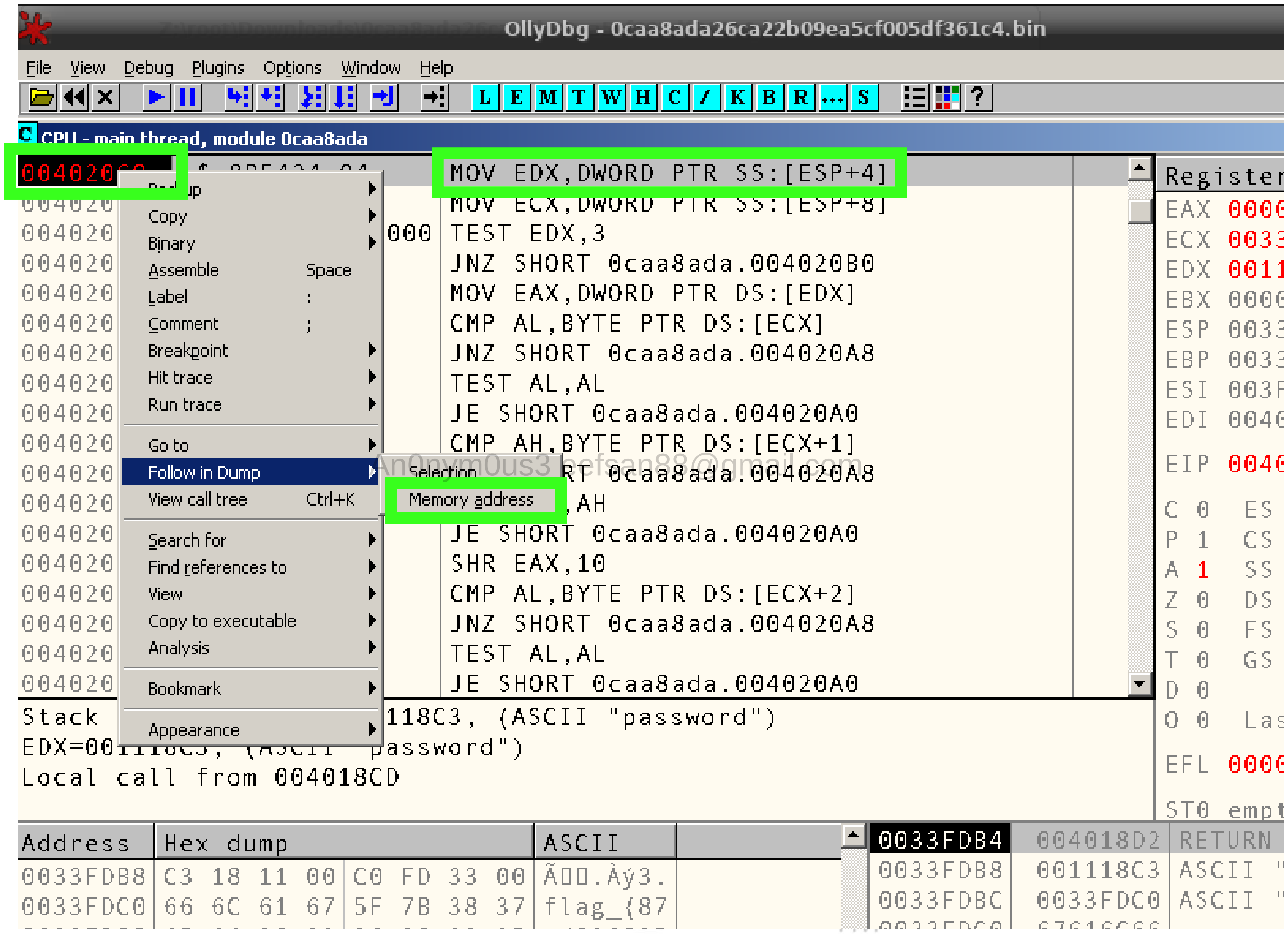
```
00402055   > 8B4424 08   MOV EAX,DWORD PTR SS:[ESP+8]
00402059   . 5F          POP EDI
0040205A   . C3          RETN
0040205B   > 8B4424 04   MOV EAX,DWORD PTR SS:[ESP+4]
0040205F   . C3          RETN
00402060   $ 8B5424 04   MOV EDX,DWORD PTR SS:[ESP+4]
                         MOV ECX,DWORD PTR SS:[ESP+8]
                         TEST EDX,3
                         JNZ SHORT 0caa8ada.004020B0
                         MOV EAX,DWORD PTR DS:[EDX]
                         CMP AL,BYTE PTR DS:[ECX]
                         004020A8
                         004020A0
                         [ECX+1]
                         004020A8
                         004020A0
```

```
Registers (FPU)
EAX 00000001
ECX 00000000
EDX 00000000
EBX 7B637E08 KERNEL
ESP 0033FE44
EBP 0033FE58
ESI 003FF000
EDI 0040240A 0caa8a

EIP 0040240A 0caa8a

C 0  ES 002B 32bit
P 0  CS 0023 32bit
A 1  SS 002B 32bit
Z 0  DS 002B 32bit
S 0  FS 006B 32bit
T 0  GS 0063 32bit
D 0
O 0  LastErr ERROR_

EFL 00000212 (NO,NB

ST0 empty 0.0
```

```
Address                            ASCII
00422000          00 00            ........
00422008          00 00             CB.....
00422010 20 43 42 00 01 01 00 00    CB.□□..
00422018 00 00 00 00 00 00 00 00   ........
00422020 00 10 00 00 00 00 00 00   .□......
00422028 00 00 00 00 00 00 00 00   ........
00422030 00 00 00 00 02 00 00 00   ....□...
00422038 01 00 00 00 00 00 00 00   □.......
00422040 00 00 00 00 00 00 00 00   ........
```

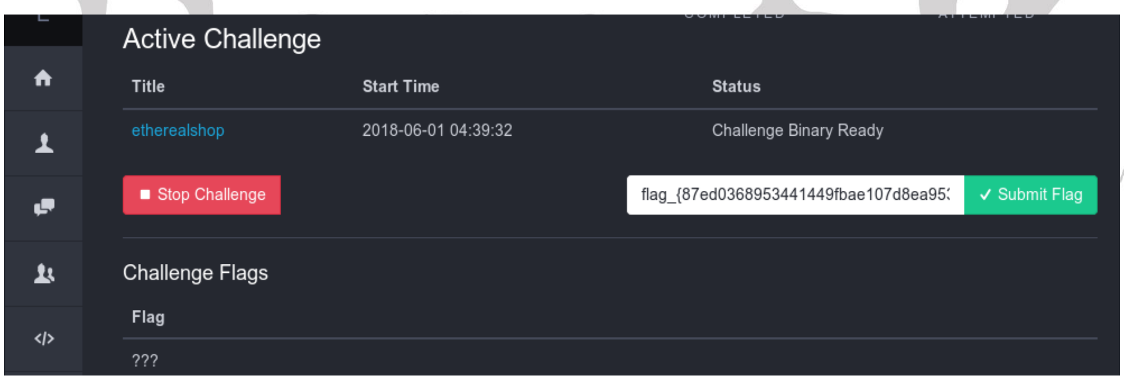
```
0033FE44  7B4606AC  RETURN to KERNEL3
0033FE48  003FF000
0033FE4C  7B461714  RETURN to KERNEL3
0033FE50  003FF000
0033FE54  0040240A  0caa8ada.<ModuleE
0033FE58  0033FE98
0033FE5C  7B461734  RETURN to KERNEL3
0033FE60  003FF000
0033FE64  0040240A  0caa8ada.<ModuleE
0033FE68  00000023
```

Now click on the “Play”, or Debug > Run, or press F9. This will continue executing the program to the line we set the breakpoint on. Now let's examine the the registers and find the flag. We know that the value is on the stack (displayed in the bottom right). Let's right-click and select "Follow in Dump" so that shows the memory at that address (in the bottom left).



```
00402000   MOV EDX,DWORD PTR SS:[ESP+4]
004020     MOV ECX,DWORD PTR SS:[ESP+8]
004020     TEST EDX,3
004020     JNZ SHORT 0caa8ada.004020B0
004020     MOV EAX,DWORD PTR DS:[EDX]
004020     CMP AL,BYTE PTR DS:[ECX]
004020     JNZ SHORT 0caa8ada.004020A8
004020     TEST AL,AL
004020     JE SHORT 0caa8ada.004020A0
004020     CMP AH,BYTE PTR DS:[ECX+1]
004020     JNZ SHORT 0caa8ada.004020A8
004020     ,AH
004020     JE SHORT 0caa8ada.004020A0
004020     SHR EAX,10
004020     CMP AL,BYTE PTR DS:[ECX+2]
004020     JNZ SHORT 0caa8ada.004020A8
004020     TEST AL,AL
004020     JE SHORT 0caa8ada.004020A0

Stack      118C3, (ASCII "password")
EDX=00     password")
Local call from 004018CD

Registers
EAX 0000
ECX 0033
EDX 0011
EBX 0000
ESP 0033
EBP 0033
ESI 003F
EDI 0040
EIP 0040
C 0  ES
P 1  CS
A 1  SS
Z 0  DS
S 0  FS
T 0  GS
D 0
O 0  Las
EFL 0000
ST0 empt

Address   Hex dump                  ASCII
0033FDB8  C3 18 11 00 C0 FD 33 00  Ã...Àý3.
0033FDC0  66 6C 61 67 5F 7B 38 37  flag_{87

0033FDB4  004018D2  RETURN
0033FDB8  001118C3  ASCII "
0033FDBC  0033FDC0  ASCII "
```

We now have our flag displayed in the Hex dump located in the the bottom left square. Now let’s go back to ESCALATE, submit our flag, and complete the challenge. Enter the flag and click submit.



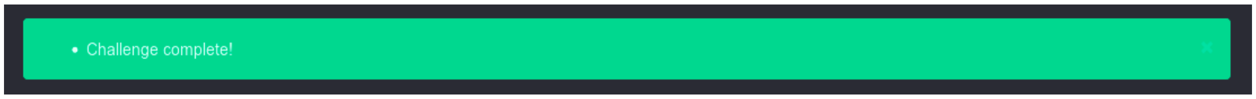
Active Challenge

Title	Start Time	Status
etherealshop	2018-06-01 04:39:32	Challenge Binary Ready

Challenge Flags

Flag
???

Challenge complete! Great work—You have taken a couple fundamental steps in improving your reverse engineering skills.



- Challenge complete!