# Reverse the Program:

The attached binary file called `program` demonstrates an authentication process. The program creates a socket, waits for connections, and to receive a message from a client that connects to the listening port (8888). Depending on the sent message, the program will send different replies to the client informing him whether the message sent is the correct one, close to the correct one, or not even close.

**Objective**: Receive a **Successful** reply from the server.

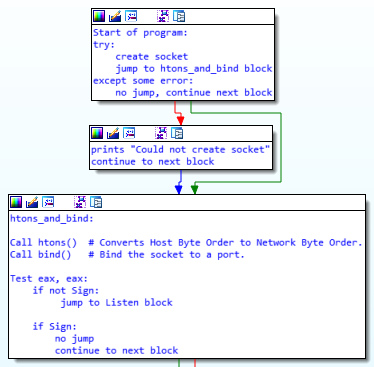
**Avoid**: Receive a **failed** reply from the server

# Reversing with IDA:

At the start of my research, I opened the program with **IDA** and learned about how the program takes the client’s message and parses it to compare with the expected correct message. **Here is the whole program workflow explained when a correct message is sent to the server**:

## Create Socket and Listen for Connections:

To receive messages from others, a socket connection must be made. The program starts by creating a socket and listen for connections. In addition, there is a call to `htons` before a socket is made. htons is a function that converts the **Host byte order** to **network byte order**.



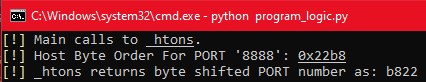


Figure I: call to \_htons

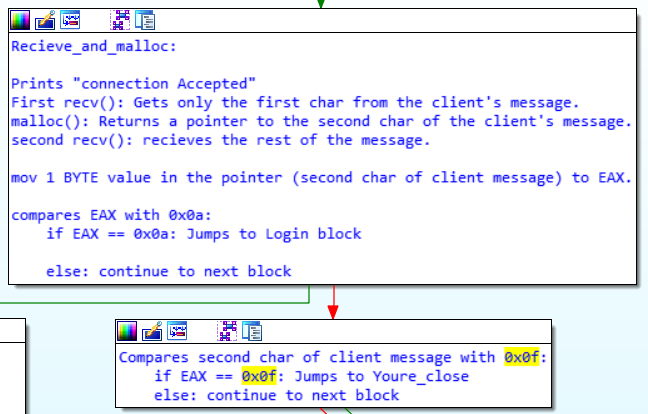
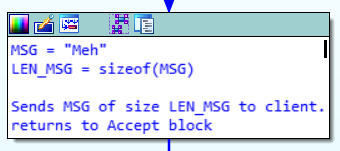
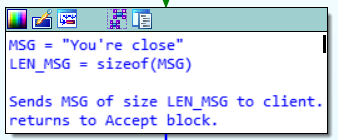
## Wait for Client Message:

After creating the socket successfully, the program will wait for a message from a client to be sent to it. It will receive only 1 character (1 Byte) from the message i.e. the letter ‘A’, move that letter with zero-extend (MOVZX), meaning pad the rest of the space with zeros and use that result to allocate memory on the heap using malloc.

## Validating “token” That Will Allow or Deny the Login Process to Begin:

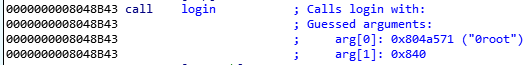
The program will receive the rest of the client’s message and starts by comparing the first character in the string. This comparison has 3 results:

1. Comparison is correct and the program jumps to the authentication process (function login).
2. The comparison is close and the program will not start the authentication process. Instead, it will send a “You’re close” message to the client and start waiting for connections again.
3. The comparison is neither of the two above – the program sends “Meh” to the client and waits for connections again.



## Login Block in main:

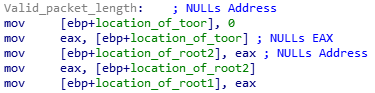
* If the character is 0x0f, the program will not start the authentication process. Instead, it will send “You’re close” and wait for another connection.
* If the first character (msg[0]) is 0x0a, the program will jump to somewhere in main where there is a call to login function that has all the necessary functions to authenticate the message received.
* If the first character is **neither** of those above, the program will send “Meh” back to the client and wait for another connection.



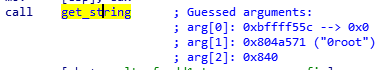
### Login Function:

Inside the login function, there is a check for a valid packet length. If the packet is not valid it will exit the program, not allowing for the authentication to begin.

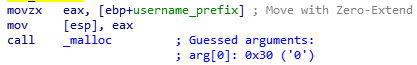
#### NULLs Memory Addresses for Later Use:



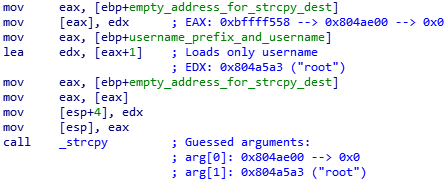
#### Call to get\_string:



##### Malloc is called with the second character of the message as an argument:

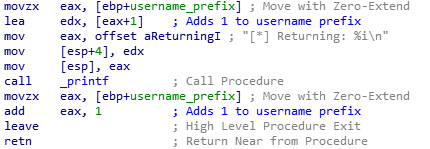


##### Preparations for strcpy are made:



Here, strcpy copies everything until a NULL byte or EOF. To pass the program and get success, a NULL byte must be inserted after the username as a part of the message.

##### Return the decimal value of the username prefix number:



After strcpy, the program will print to stdout the decimal value + 1 of the character it used in the previous malloc. It will store the result (username\_prefix + 1) in memory for later use.

Example: “0root” was in the message. Malloc takes the first character and allocates memory using that argument. ‘0’ is 0x30 with a decimal value of 48. After strcpy, the program will return 48 + 1 and print the returned value in this format:

“[\*] Returning: %i\n” ; %i = Decimal value of the returned result.

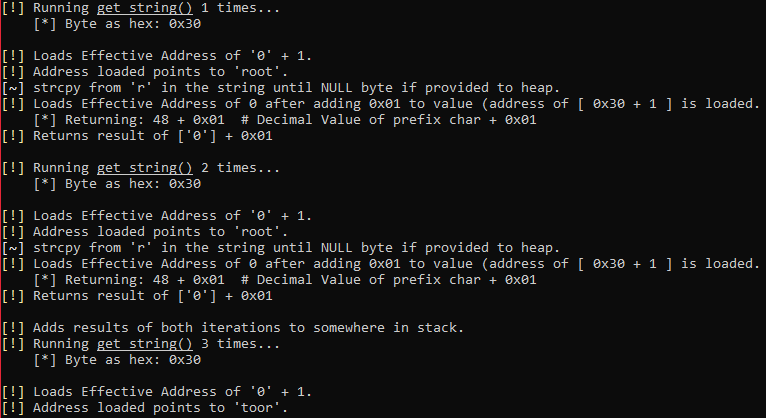
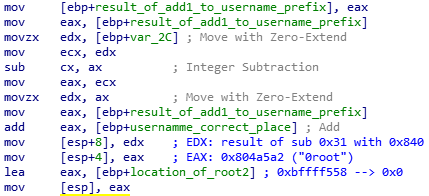
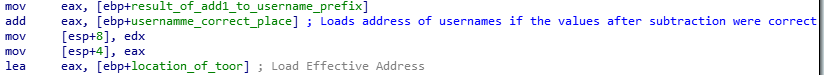


Figure II: get\_string functions explained

##### Preparations for the next get\_string calls:



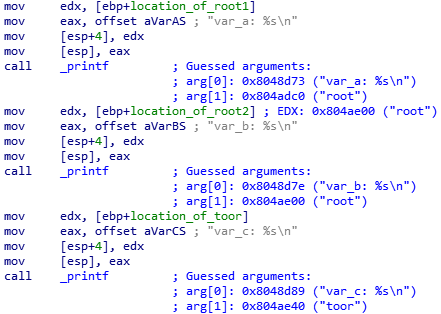
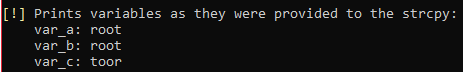
The returned result from the previous get\_string is moved to the stack. The result is then subtracted, from the result of MOVZX with the first character received (‘A’ in our example). (0x840-0x31). The correct address (an empty address) is loaded for the next get\_string call. The same process as above is happening exactly 3 times overall.



**The result of the subtraction each time after** get\_string **is crucial** because the result is used to load the correct address for the next location of the username in the whole string. If the username prefix is not correct, wrong addresses will be loaded to strcpy, making the program return failed on the final comparison.

#### Print the Variables It Received:

After all strcpy were made inside the 3 get\_string function calls, the program will print out the variables it got. It takes the memory address it received from strcpy and prints the variables 1 at a time. **Again the** NULL **byte is crucial**.

#### Comparison of Variables With Expected Variables:

The comparison is made using strcmp. The program takes the above variables as they were provided and compares them using strcmp with **hard-coded variables** stored in memory.

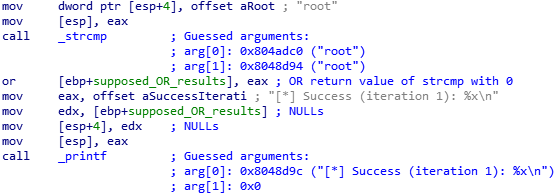
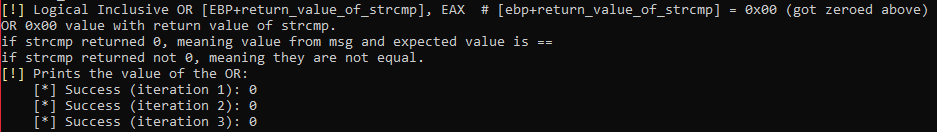


Figure III: Example of 1 strcmp

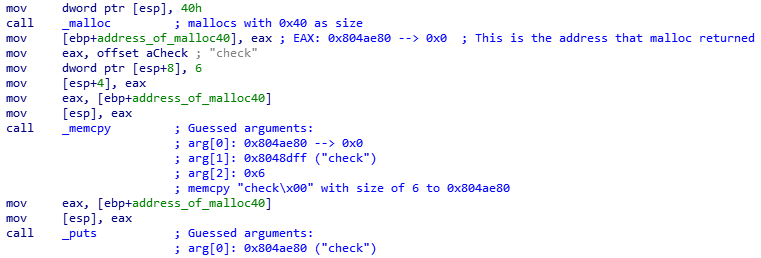
Strcmp returns a value to EAX stating if the comparison is equal or not. If the comparison is equal, it will return 0 to EAX, else - non-zero value. **Logical inclusive OR operation** is made on the returned value with **0**. After that, the program will print the value of the **OR** result.



Each iteration is every result of **strcmp return value** **OR** with **0**. To make things easier the program prints out the result as shown above. To succeed in the objective the result should always evaluate to **0** (as shown above).

#### Malloc(40) & memcpy(void \*dest, const void \* src, size\_t n)

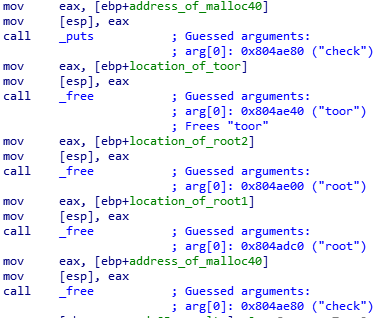
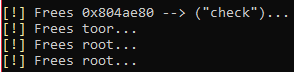
A call to malloc with the size of 0x40 is made. Memcpy is called to copy “check\x00” that ends with NULL byte to the pointer returned from malloc. Puts is called to print out that string.





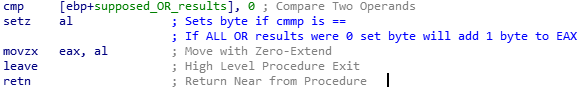
#### 4 Calls to Free:

**4 frees are called on each of the variables in reverse order**: “check” 🡪 “toor” 🡪 “root” (the second one) 🡪 “root” (the first one)



#### Compare Result of the OR Operation With ‘0’ and Return to main:

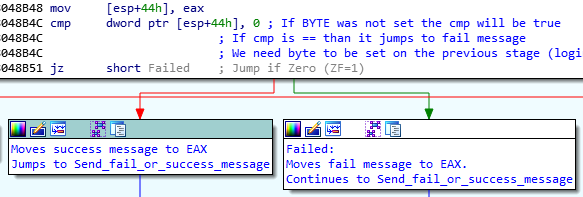
After all the comparisons and frees, there is a comparison with the results of the previous OR operation to value of ‘0’. If the comparison is equal (meaning if all strcmp returned 0 and OR 0 with 0 were executed) The results are saved in a location on the stack named “supposed OR results” in the picture. The comparison is made on all the results and if the result is still **0** the comparison will turn the **ZERO flag** on. After that, there is a set byte instruction (Set byte if ZF=1). Set byte will set byte to EAX if zero flag is on.

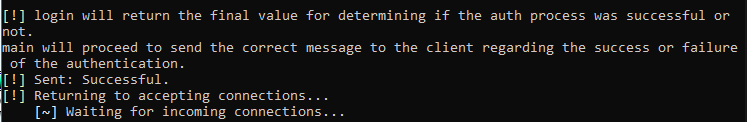


**The program returns to main with the value of 0 or 1 depending on whether the set byte was executed.**

### Final Comparison:

These next steps will determine if the server will reply with “successful” or “failed”. The return value from login function is moved to somewhere in the stack. The comparison is made on that location with ‘0’. **If the comparison is equal** (Meaning, if byte was **NOT** **set** in the previous stage), the program jumps to the location where it sends a “failed” message. **But if the comparison is not equal** (Meaning, byte **WAS** **set**) the program doesn’t jump and continues to sending a “successful” message to the client.





# Conclusions:

A program that authenticates the client input with various stages and returns the result as strings of “successful” or “failed”. The message must have the following attributes to pass the program and get **SUCCESS**  and all are crucial:

1. Providing a character that the first \_recv function will get. Only after that character, the second \_recv will get the whole string that starts with the **authentication** **token** (0x0a).
2. The token at the start of the string that will **allow** or **deny** access to the login function.
3. Usernames prefix numbers - Each username must start with a prefix number **(the correct one is ‘0’)** This number will be used for getting the correct index locations of the 3 different usernames for the different functions.
4. Each username **must end with a NULL byte** for strcpy to copy usernames properly to the heap.
5. **A Padding** of **43 bytes** is **required** between each username.

# Connect to The Program:

In order to connect to the program and receive a “successful” message, the correct message will be:

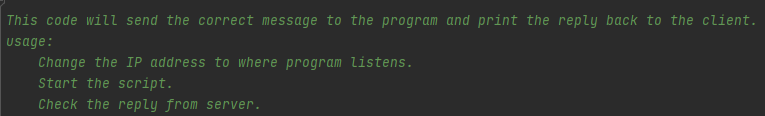
‘A # First character that will get recived in \_recv(1)

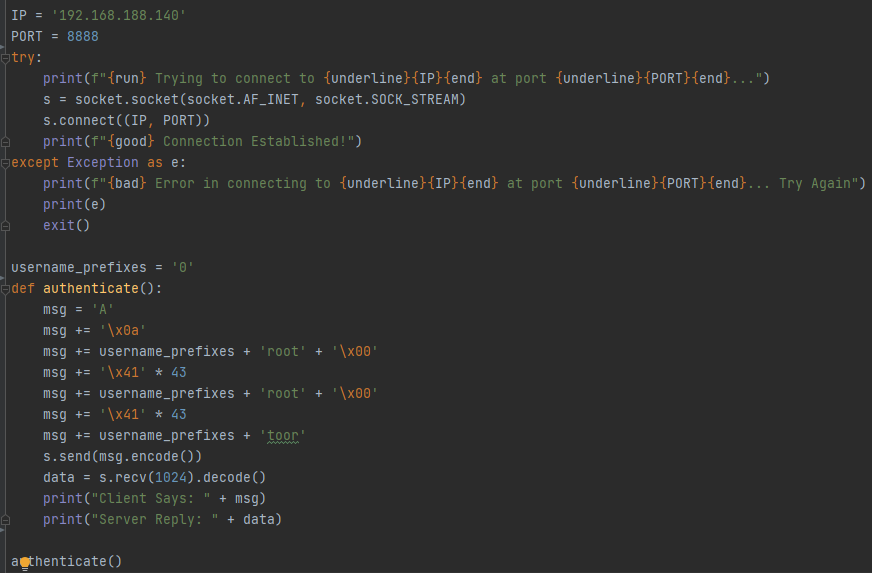
\x0a # New line character in hex (0x0a)

0root\x00AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA0root\x00AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA0toor’ # Username prefixes, usernames and padding of 43 bytes. Each username ends with NULL byte)

## Connect.py

This python script will connect to the server at port 8888, and send the message that will return “successful”. In addition, it prints the message as **ASCII characters** to stdout as well as the reply from the server.





* **Output**: