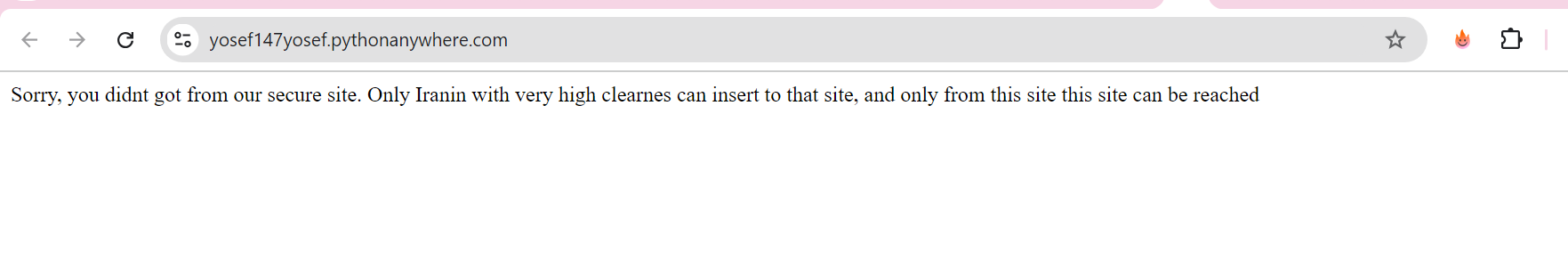
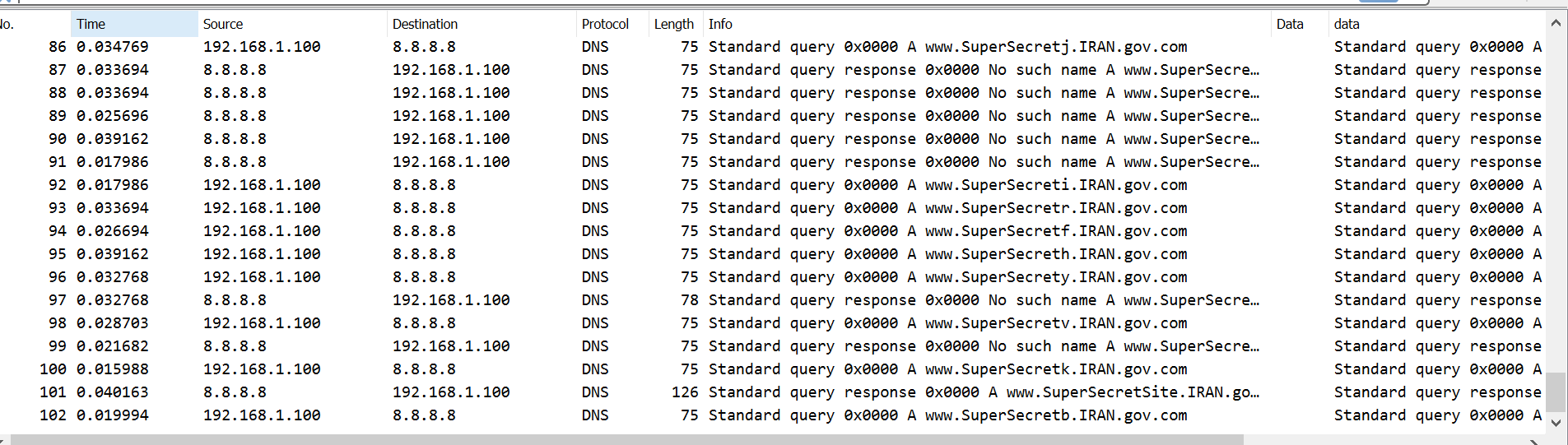
First, let's open the website given to us in the link.



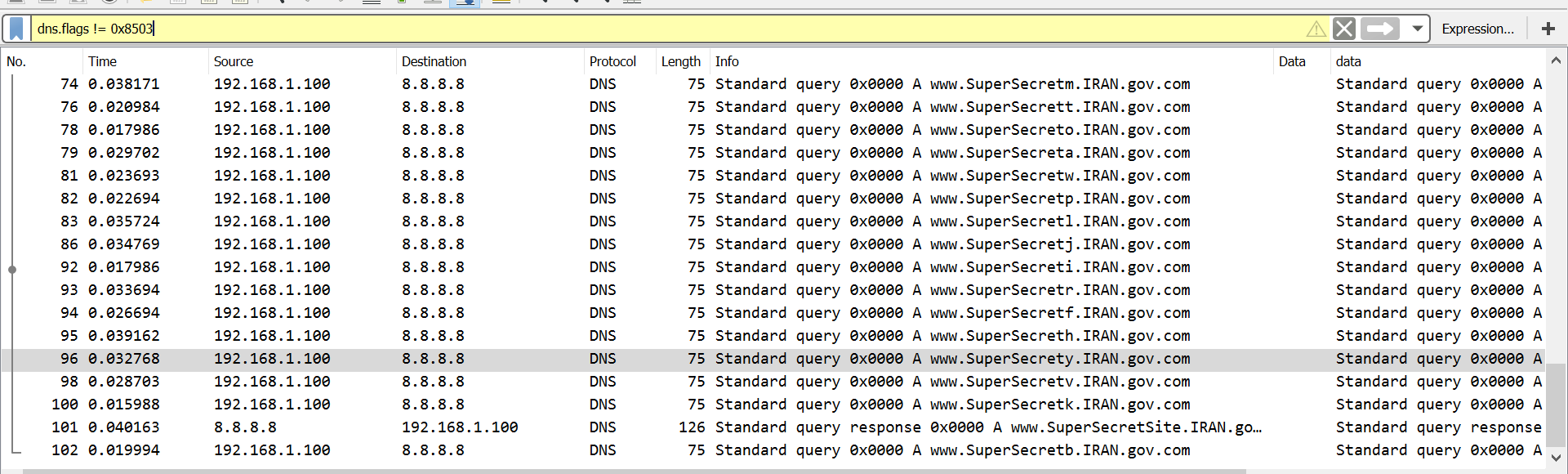
This website is secure. If we read the instructions, we understand that we cannot just access this site, but only from another site where a security check will be performed to ensure we have the appropriate clearance level to access the site.

Which site? It's unclear. We need to open the documentation file and see.



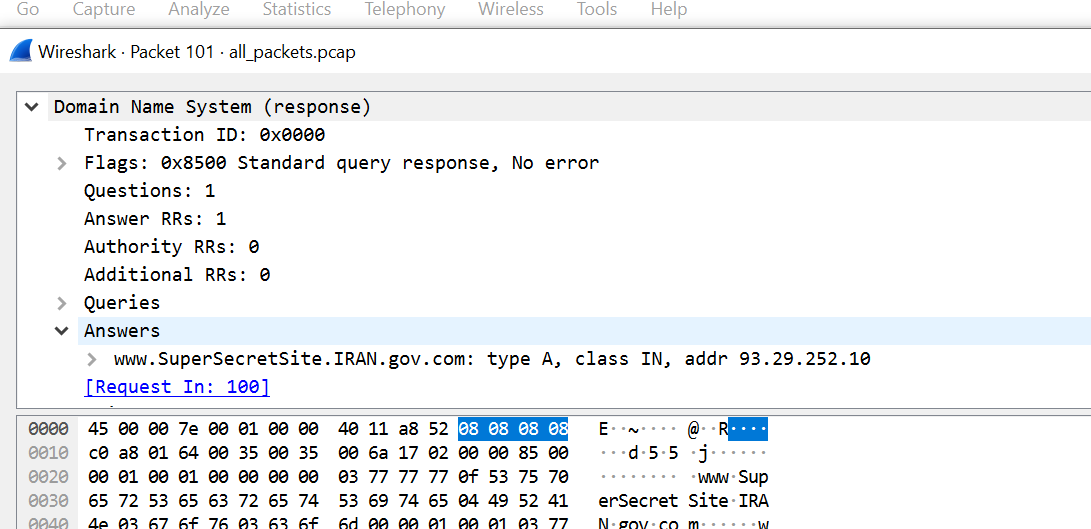
The file contains many packets with unclear protocols. But we can see that there are DNS queries.

There's a lot of mess here. The answers aren't organized properly, and there are many requests to a site with a similar name. We need to filter only DNS requests that received a positive response.



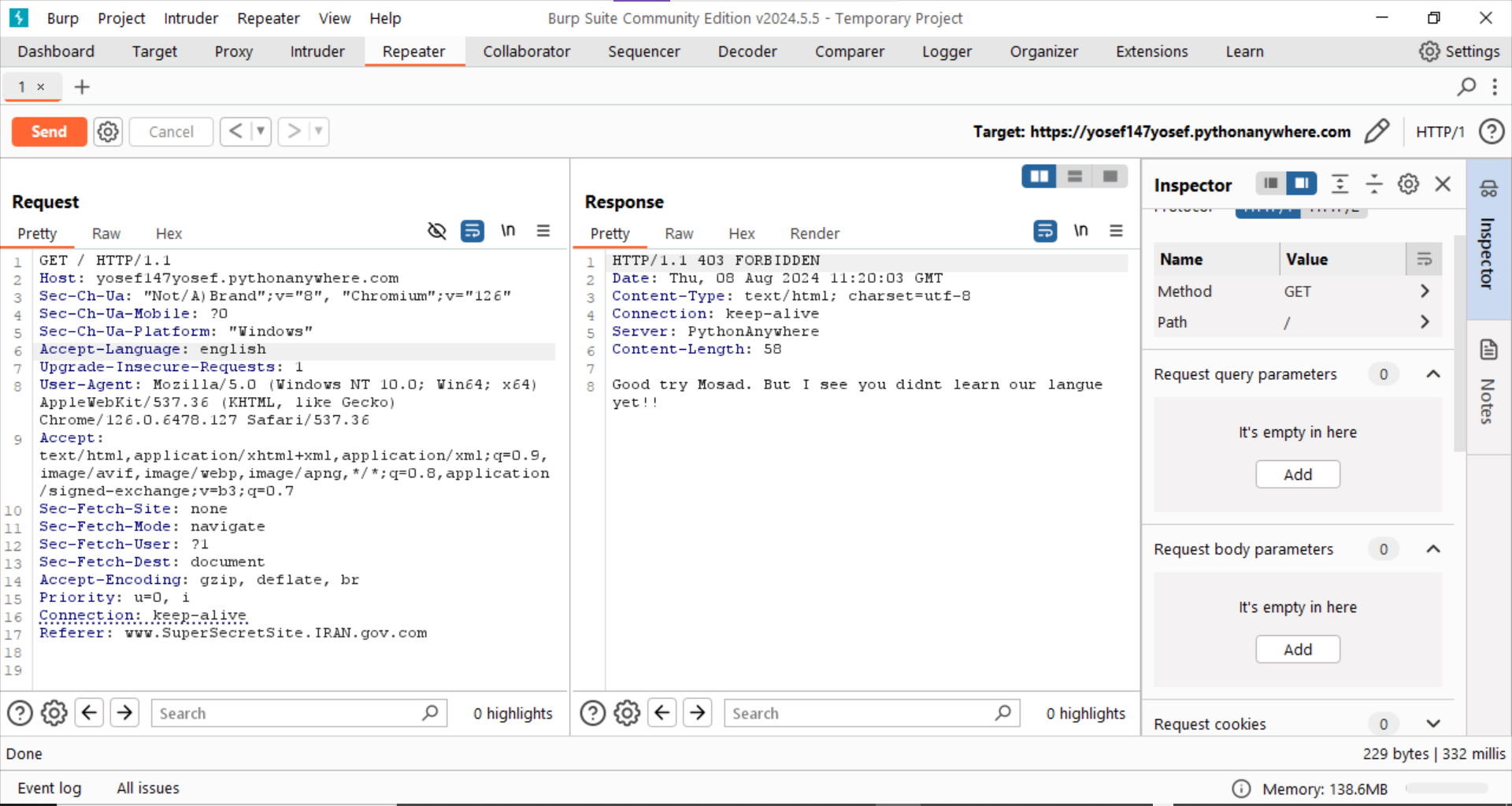
I filtered out packets that didn’t receive a “domain not found” response, and there is one packet where the server returns a positive response.

We found a domain. This is probably the site from which we need to access the site we are trying to hack. Now we need to make the site think that we indeed came from this secure site.



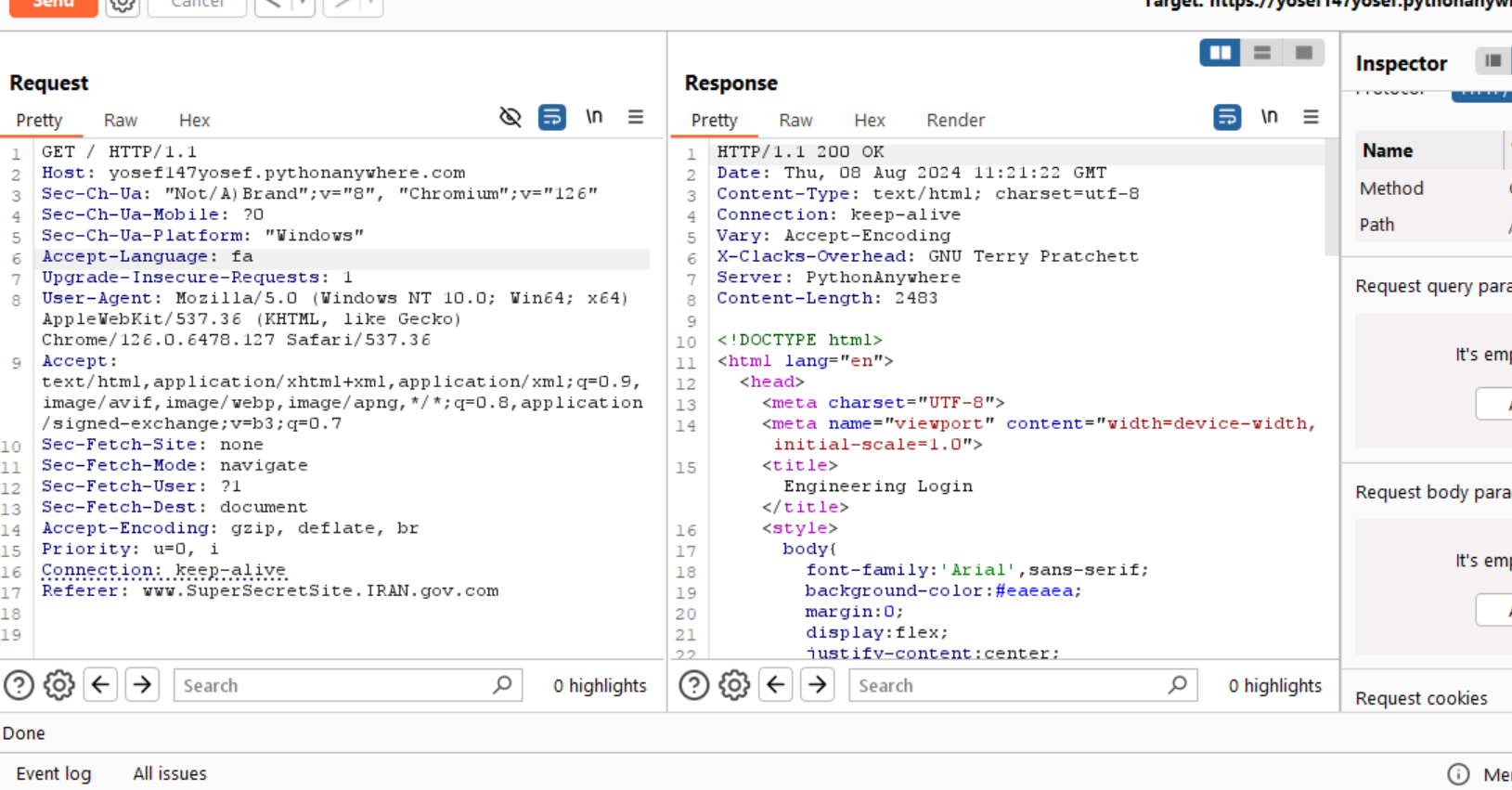
There is a field in the HTTP protocol called the referrer, which ensures that we came from the secure site. Let's try to change it to match.

We sent an HTTP request using a tool called Burp Suite. We received a strange response.



The site identifies that we are not Persian speakers, so it doesn’t let us in. We will change the language field to Persian.

We got access to the site!

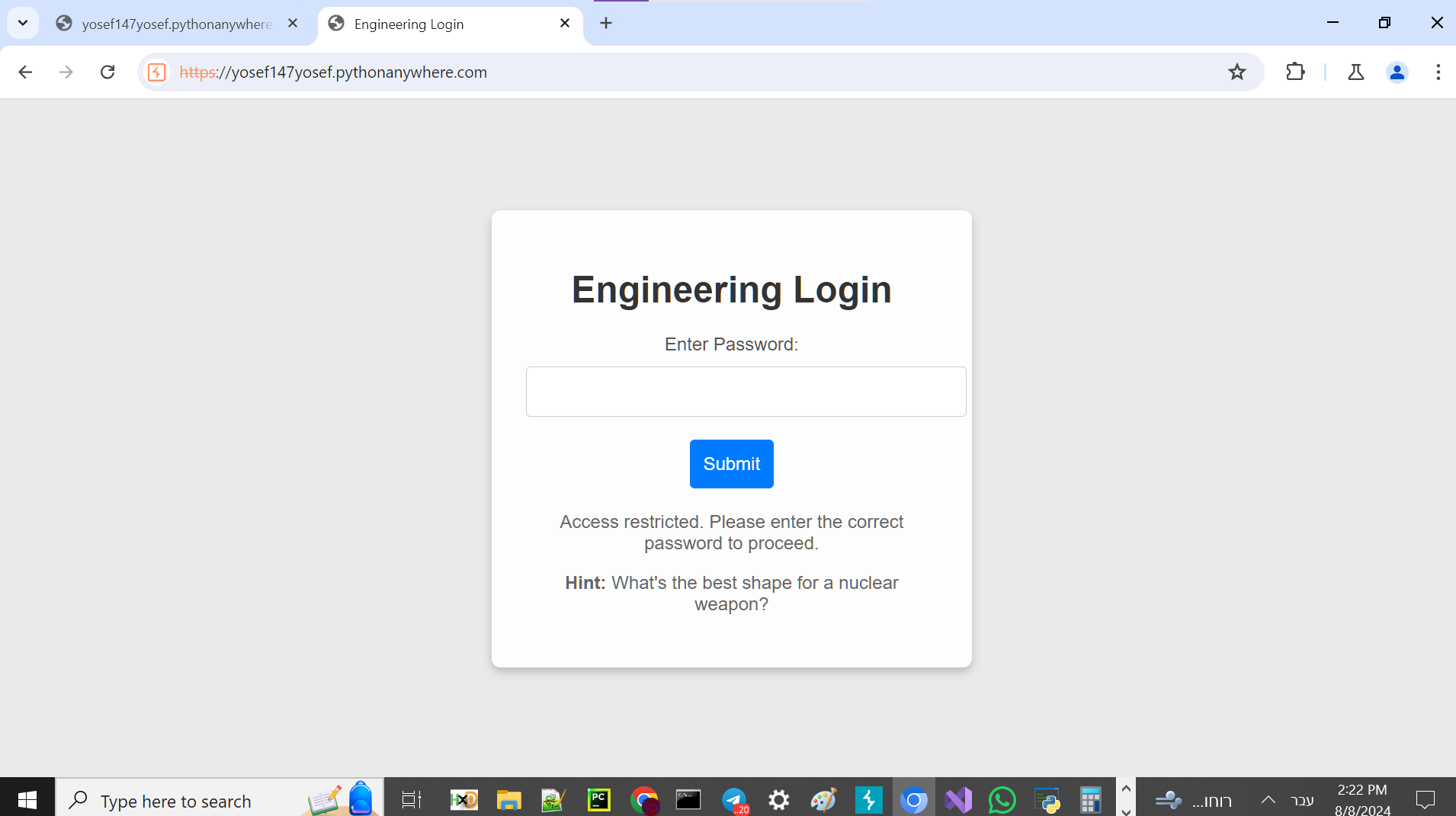


Let's see its layout in the browser.

We entered the Iranian engineering site. But now we need to enter a password.

After trying to find the password on the site itself, which, of course, failed, let's go back to the documentation file and see if the documentation of the reactor conversations might reveal the password.

There are a lot of packets in Wireshark, and there’s no way to filter packets with the correct checksum.



For this task, we won’t be able to use Wireshark. The instructions explicitly stated that the server was flooded by cyber kids, and we were even given the checksum algorithm used. So we need to write a Python code to filter only the texts of packets with the correct checksum.

The Python code:

from scapy.all import \*  
def custom\_checksum(packet):  
 if not packet.haslayer(UDP):  
 raise ValueError("Packet must contain a UDP layer")  
  
 # Create a copy of the packet with the checksum field set to 0  
 packet\_no\_checksum = packet.copy()  
 packet\_no\_checksum[UDP].chksum = 0  
  
 # Extract the raw bytes without the checksum field  
 packet\_bytes = bytes(packet\_no\_checksum)  
  
 # Calculate checksum  
 checksum = 0  
 for i in range(0, len(packet\_bytes)):  
 word = packet\_bytes[i]<<(i%4)  
 checksum += word  
 # One's complement  
 checksum = ~checksum & 0xffff  
  
 return checksum  
result = []  
def print\_packets\_with\_checksum(pcap\_file):  
 global result  
 # Read the pcap file  
 packets = rdpcap(pcap\_file)  
 for packet in packets:  
 # Check if the packet has a UDP layer  
 if UDP in packet:  
 # Check if the UDP checksum is 0xffff  
 chcksum = custom\_checksum(packet)  
 if packet[UDP].chksum == chcksum and packet[IP].src!='44.99.1.1':  
 result += [packet[Raw].load.decode()]  
  
# Replace 'your\_file.pcap' with the path to your pcap file  
pcap\_file = 'all\_packets.pcap'  
print\_packets\_with\_checksum(pcap\_file)  
print(result)

The code uses Scapy and filters only packets with the correct checksum.

['Bux pdqsrm gh bwg zmdr', 'Ivqf bq pb qzimghiam ktganzc ucz nej ivm rgexbmrkq ufwz hsg gcckcbs trtbtf', 'Rwwa Zxqhoor pyh svpkwehmq ucriar pc ptznbb ufwz mft waetcam Ublys', 'Qg bq icw ehscr wa mft hwc Mfxg evej eib n lkxzm bg rws nnvch cn gac tbmzr']

We got an array of strings, but they seem to be encrypted.

It's easy to see that this is a substitution cipher of some kind. We will try to break it using online substitution cipher tools, but the results are not logical. It seems to be a Vigenère cipher.



The correct string can be seen as:

"The rocket is not good."

Let's continue with the other strings:

"This is an important message for all the engineers from our supreme leader."

"This Message was encrypted because we are afraid of the Israeli Mossad."

"It is too round on the top. This will put a smile on the faces of the enemy."

It’s also slightly garbled, but it’s clear they’re talking about the rocket not being good because of its rounded shape.

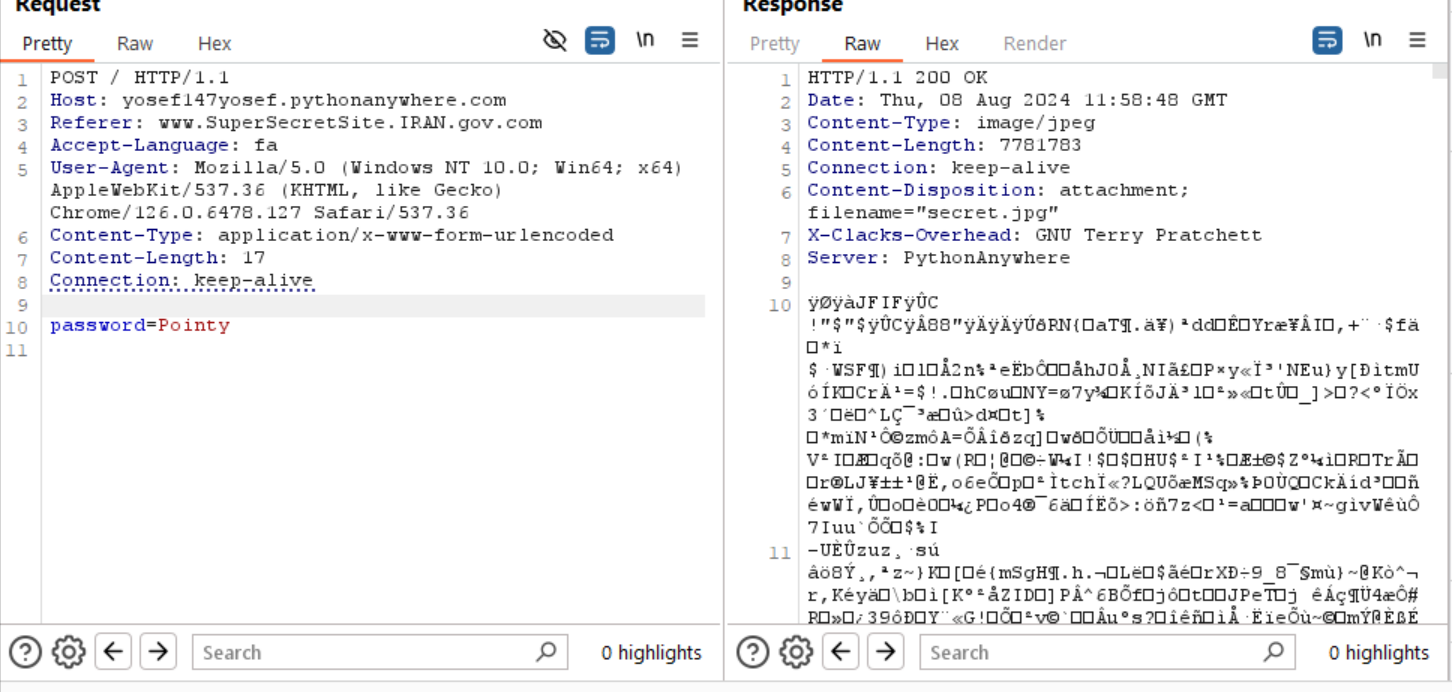
This matches the hint for the password: the ideal shape of a nuclear missile.

Of course, this is the famous line from the movie "The Dictator" by Sacha Baron Cohen: A nuclear missile should be pointy.

“It needs to be pointy!”

Of course, here too, we need to maintain the structure of the Iranian HTTP packet. Let's try the password.

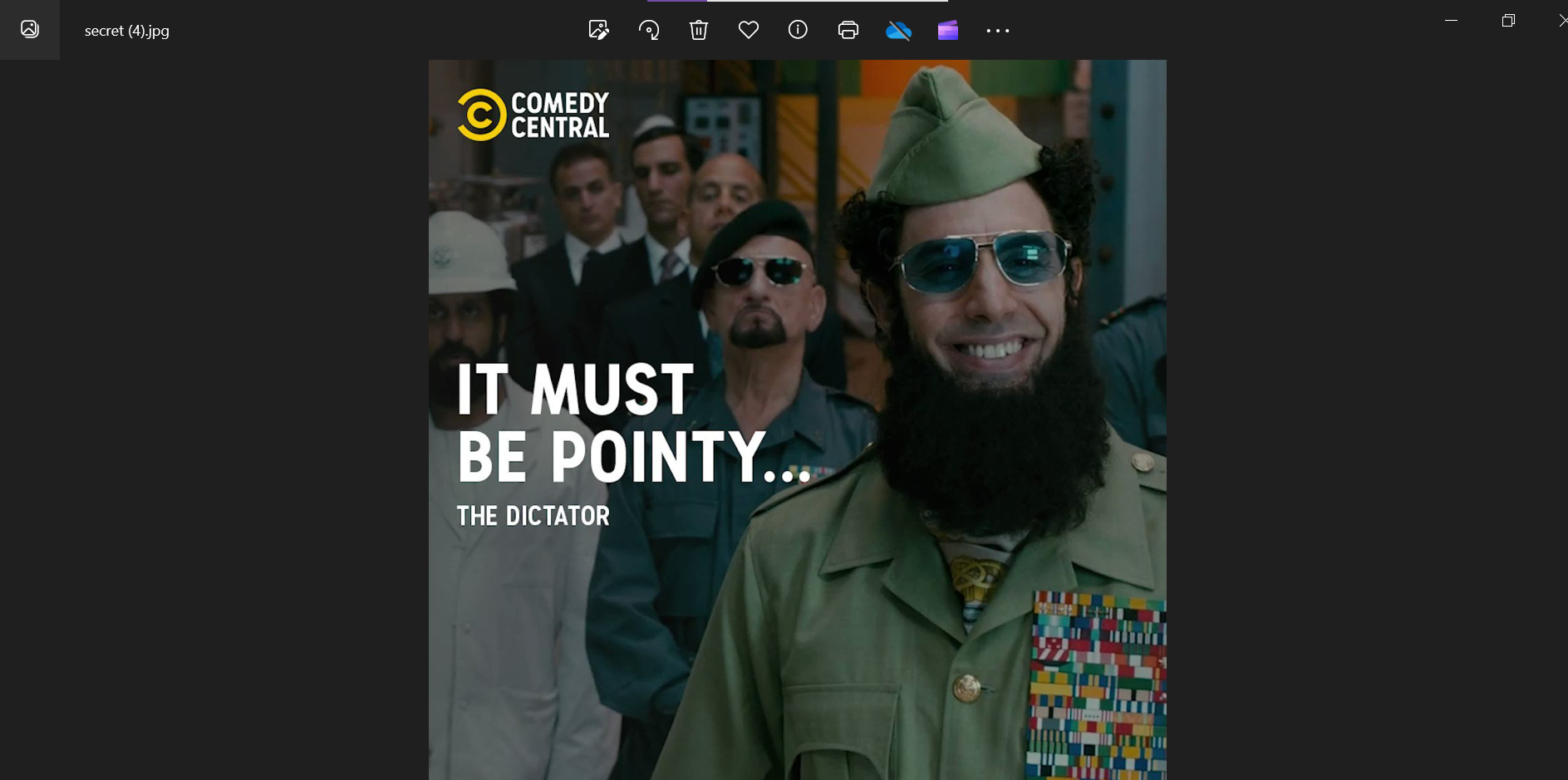
We received a picture from the server. Hooray!



But we haven't reached the flag yet.

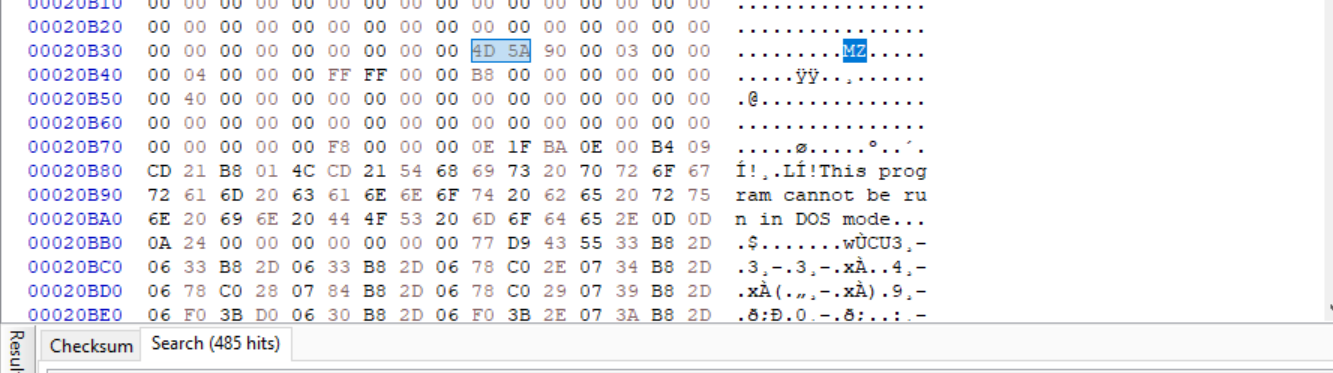
Let's open the image in a hex editor. The image is very large compared to a typical image.

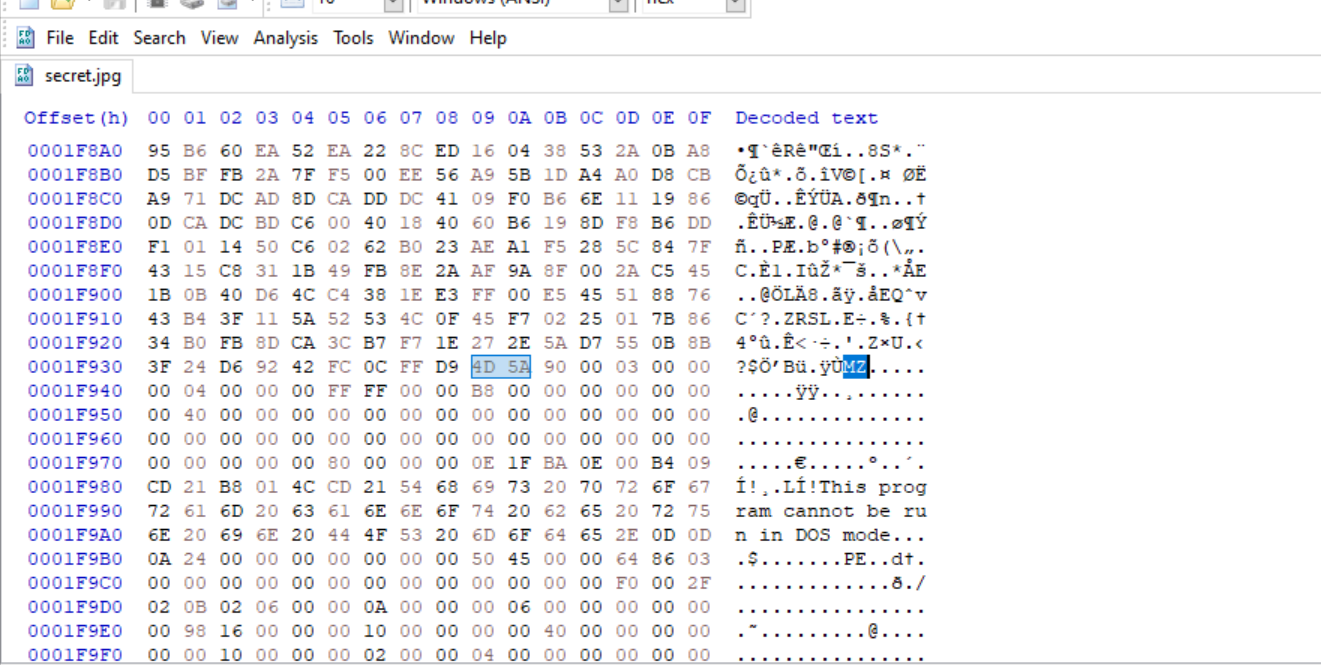
Let's see if there’s something hidden in it. We found two MZ prefixes of a PE format.



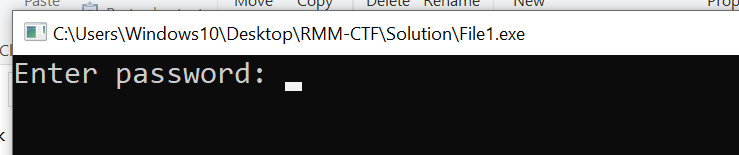
We extracted both to two executable files.

Let’s open them and see what we get.

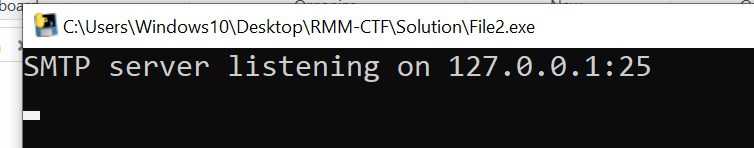




The first file asks for a password.



The second file is an SMTP server.

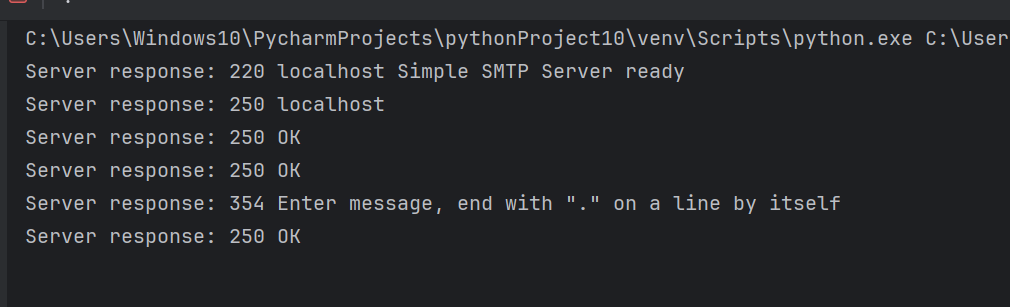


Let’s try to write a client to communicate with the server.

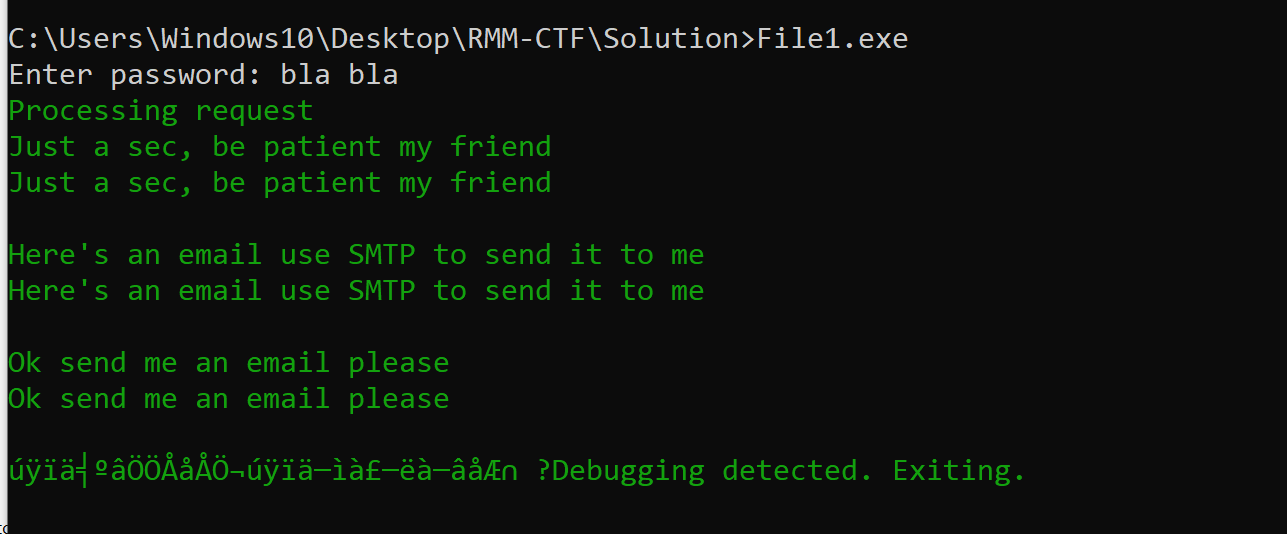
import socket  
import email  
from email.parser import Parser  
from email.mime.text import MIMEText  
  
HOST = '127.0.0.1' # Standard loopback interface address (localhost)  
PORT = 25 # SMTP port  
  
def send\_email(sender\_email, recipient\_email, subject, body):  
 # Create the message  
 msg = MIMEText(body)  
 msg['Subject'] = subject  
 msg['From'] = sender\_email  
 msg['To'] = recipient\_email  
  
 # Connect to the SMTP server  
 with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s:  
 s.connect((HOST, PORT))  
  
 # Receive the server's greeting  
 data = s.recv(1024)  
 print(f"Server response: {data.decode().strip()}")  
  
 # Send the HELO command  
 s.sendall(b'HELO localhost\r\n')  
 data = s.recv(1024)  
 print(f"Server response: {data.decode().strip()}")  
  
 # Send the MAIL FROM command  
 s.sendall(f"MAIL FROM:<{sender\_email}>\r\n".encode())  
 data = s.recv(1024)  
 print(f"Server response: {data.decode().strip()}")  
  
 # Send the RCPT TO command  
 s.sendall(f"RCPT TO:<{recipient\_email}>\r\n".encode())  
 data = s.recv(1024)  
 print(f"Server response: {data.decode().strip()}")  
  
 # Send the DATA command  
 s.sendall(b'DATA\r\n')  
 data = s.recv(1024)  
 print(f"Server response: {data.decode().strip()}")  
  
 # Send the email message  
 s.sendall(msg.as\_string().encode())  
 s.sendall(b'\r\n.\r\n')  
 data = s.recv(1024)  
 print(f"Server response: {data.decode().strip()}")  
  
 # Receive the response email from the server  
 response\_email = b''  
 while True:  
 data = s.recv(1024)  
 if not data:  
 break  
 response\_email += data  
 if data.endswith(b'\r\n.\r\n'):  
 break  
  
 # Parse and print the response email  
 try:  
 msg = Parser().parsestr(response\_email.decode())  
 print('Received response email:')  
 print(msg)  
 except (UnicodeDecodeError, email.errors.MessageParseError):  
 print('Error parsing response email')  
  
 # Send the QUIT command  
 s.sendall(b'QUIT\r\n')  
 data = s.recv(1024)  
 print(f"Server response: {data.decode().strip()}")  
  
# Example usage  
send\_email("your\_email@example.com", "test@test.com", "Test Email", "This is a test email.")

```

The server does not return anything.



Let's try entering a random password to see how the second file responds.

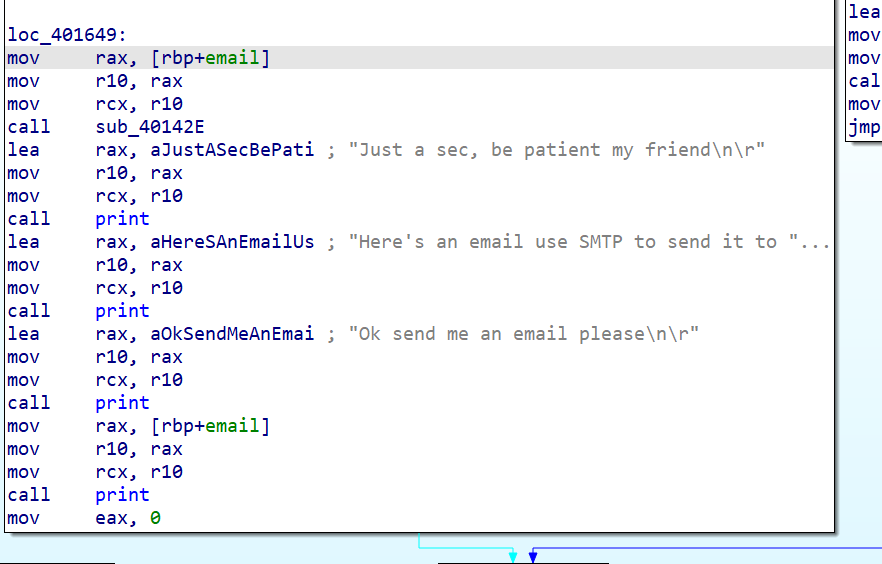


It currently doesn't provide anything. But where will we get a password?? If we give it the correct password, we will get the email.

Let's try reversing to crack it.

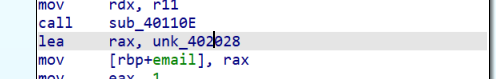
Let’s perform a static analysis. The code contains anti-debuggers, so I won't try to debug it but will analyze it statically.

Here are all the print statements. But the last print of the email is not from a fixed place. Let’s try to understand what's happening here.



The email variable receives a string from memory, but the string is encrypted.

Let’s return to the prints; there is one function that isn’t clear what it does:



\*\*This function does XOR with a fixed memory location.\*\*

It probably decrypts the encrypted string. Now we just need to see what the key is.

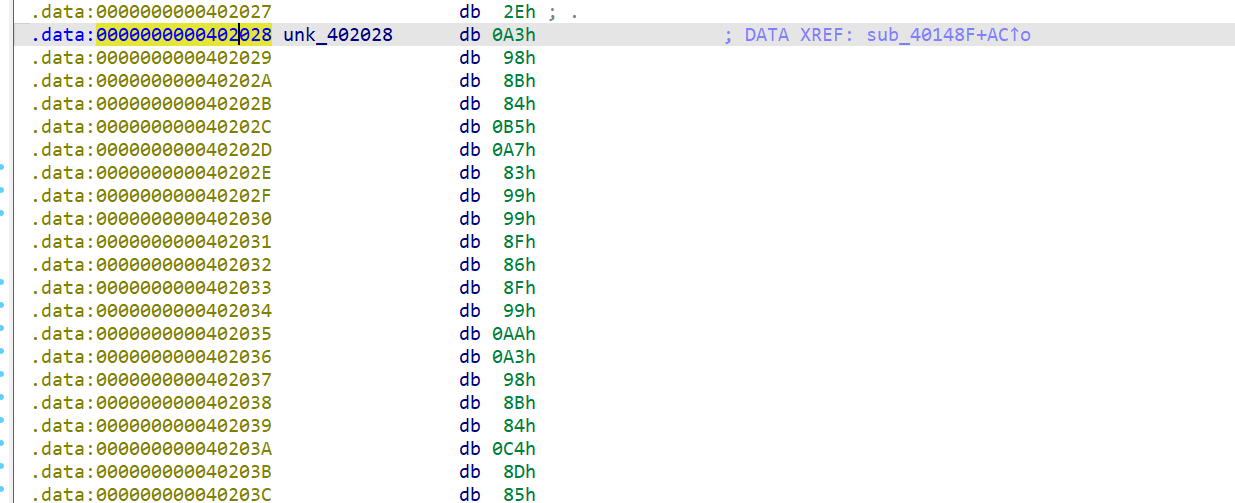
We do a lot of manipulations on the key. Its initial value is `0xEA`.

There’s a place here where it checks the password using a written hash function, and it changes the key and the pointer to the email string according to comparisons.

The hash function is terribly complicated.

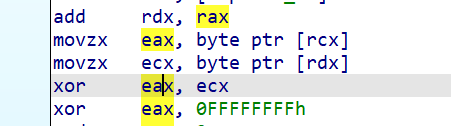
There’s no chance of finding the right password within a reasonable time. Let's patch it. The simplest idea is to just make the hash result the one we want. This will save us from understanding the process behind key creation and reaching the exact index in the email string.

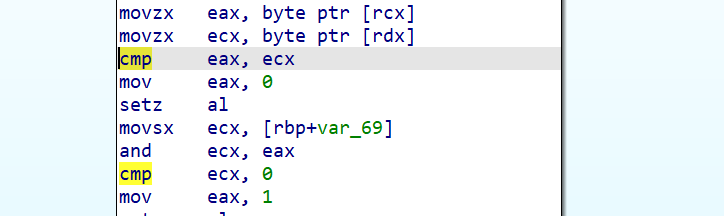
Here’s access to the house from the hash and the hash from the input hash.



There are further operations between them, so it's obviously intended

to confuse us.instead of just checking if there are the same, there are bytes operation between each one that supposed to change the pointer to the encrypt email string, and the key to decrypt it. The hard way will be to track this process, and see whats the encrypted string and key. This will take some time, consider the fact it seem complicated and there are anti debbuger all around. So I tried to just replace all the cmp and xor between the bytes being read to be with itself. If I will do that to any reading of the two arrays the result should be always like the input\_hash=password\_hash.

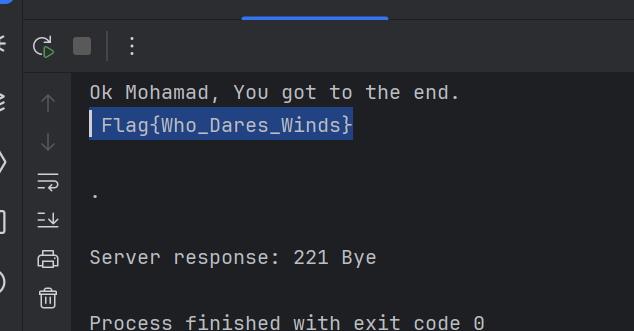




There are two xors and one cmp instruction between them.

After patching lets run the program again.



We got the Email!. Now, lets send to this email a message. 

Flag{Who\_Dares\_Wins}