Network Security

Lab3 - *packetz.pcap*

When presented with a .pcapng file (Packet Capture Next Generation), the first thing to understand is **what it is**. A .pcapng file is a format used to store network traffic captured over a period of time. It records each packet's metadata and payload, allowing us to "replay" the network communication for analysis.

These files are often created using packet capture tools like **tcpdump** or **Wireshark**, and they store all the traffic that passed through a specific network interface.

To analyze the .pcapng file, I used **Wireshark**, one of the most powerful and widely used **network protocol analyzers**.

After launching Wireshark and opening the packetz.pcapng file, the first step is usually to get an overview of the types of protocols and conversations that took place.

First thing I did was to see which protocols are present (**Statistics → Protocol Hierarchy)**.

* Transmission Control Protocol - 95.5%, of which only 30% contain data
* Address Resolution Protocol - 4.5%

I applied **Data** as a filter.

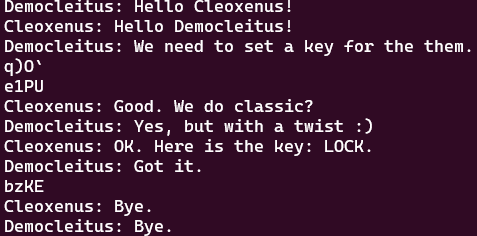
Then, I tried to **follow TCP Stream** to reconstruct readable conversations, but without success.

After filtering for Data packets and attempting to **Follow TCP Stream**, I noticed that many streams either contained no meaningful data or had repeated content across multiple sessions — likely just protocol handshakes or background noise.

Instead of relying solely on stream reconstruction, I started **manually checking the contents of each TCP packet that contained data**. This deeper inspection revealed something more interesting.

At first glance, many packets contained what looked like random characters or protocol metadata. But then I noticed a pattern in some of them: **a sequence of exchanges that resembled a text-based conversation**.

Here’s the reconstructed conversation from the raw TCP payloads:



The key moment in the conversation is: „Here is the key: LOCK.”.

After the conversation provided the key LOCK, I initially attempted to decrypt the encrypted chunks using XOR encryption with the key "LOCK." However, the result didn't yield any meaningful plaintext.

After the XOR decryption attempt, I realized that the data might not be directly encrypted using a traditional cipher. Given the flag format ECSC{SHA256}, which suggested that the encrypted data could be a **hash** or **checksum**, I decided to dig deeper into the packet details, particularly focusing on the **Internet Protocol (IP) header**.

At this point, I turned my attention to the **IP header**, which contains critical information for routing and error checking. What I found in the header caught my attention.

**Header Checksum Validation Disabled**

One important aspect of the IP header that stood out was the **checksum validation**. Typically, the **header checksum** is used for error detection. However, upon examining the IP header of the packets, I noticed that **header checksum validation was disabled**. This detail was significant because it meant that certain values in the IP header could be **modified** without causing the packet to be discarded due to corruption.

This opened up an interesting possibility: the packet's header values could be altered in such a way that the data remains valid, potentially revealing hidden information or clues, without breaking the integrity of the packet.

After discovering that the **header checksum validation** was disabled and that certain fields in the **IP header** (like **Identification**, **Differentiated Services Field**, and **Header Checksum**) could potentially be manipulated, I decided to test an approach where I tried to **decrypt** the values of these fields using the XOR operation with the **key "LOCK"**.

My reasoning behind this was that the values of these fields (like the checksum, identification, and DS Field) could contain hidden data or information that might be useful for revealing the flag, particularly if XOR with the key "LOCK" could expose readable text or a hash-like structure, which might match the format ECSC{\*}.

After multiple attempts to decrypt various fields using the key **LOCK**, I focused on the **Differentiated Services Field** across the captured packets. Specifically, I checked this field on the **last four TCP packets containing data**, and what stood out was that—when decrypted using XOR with the key "LOCK"—the resulting characters were:

**C**, **E**, **S**, **C**

Interestingly, the very **first packet** in the capture, when processed the same way, yielded the character ‘**}**’.

This immediately caught my attention. The values seemed meaningful but were in the **wrong order**. Rearranging them from **end to start**, I obtained:

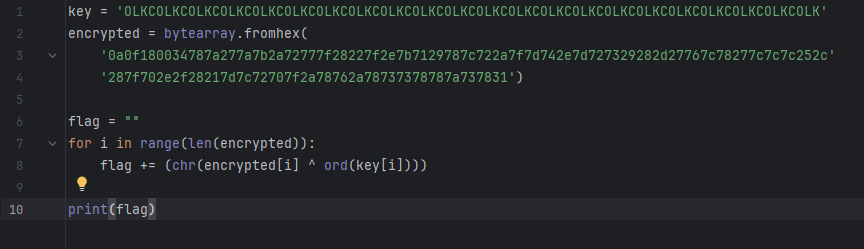
ECSC{...}

This revealed the correct start of the flag in the expected format. It became clear that the **flag was encoded in reverse**, spread across specific IP header fields (in this case, the Differentiated Services Field) and XOR-encrypted using the shared key "LOCK" mentioned in the in-packet conversation.

This insight guided the remaining part of the analysis: applying the same decryption strategy in reverse order, packet by packet, across the same header field, until the entire flag was recovered.

Once I identified that the encrypted message was most likely hidden in a series of values and XORed with the repeated key **LOCK**, I extracted the full encrypted string from the packet data and wrote a simple Python script to decrypt it.

Since the encryption was done using a repeating key pattern (LOCK repeated), I expanded the key to match the length of the encrypted message and applied XOR decryption. Here is the script I used:



Running this script returned the final **CTF flag**, successfully decrypted from the payload:

ECSC{41d57a183ca0b02f471e367a190fdfd903d307fcd43accb20930f35e48074107}