SPIDER TASK-1

Karthikeya Naidu (108124135)

1 Basic Tasks

- Cybersecurity
- Computer Networking
- Blockchain

2 Domain Specific Task

- Cybersecurity

—

3 Basic Task-1: Computer Networking

3.1 Problem Statement

This analysis aims to provide insights into the traffic patterns and behaviors within a basic PCAP file. The capture likely contains multiple protocols and file transfers, which may highlight potential signs of interest or suspicious activity.

Aim: To analyze network traffic captured in a basic PCAP file using Wireshark in order to identify communication patterns, protocol usage, and potential indicators of suspicious or anomalous activity, such as unauthorized file transfers or abnormal connections.

3.2 Procedure

1. First upload the PCAP (aka packet capture file) in Wireshark (STEPS: wireshark_file_open)
2. Which will result in the traffic network data of some activity done in that device or application

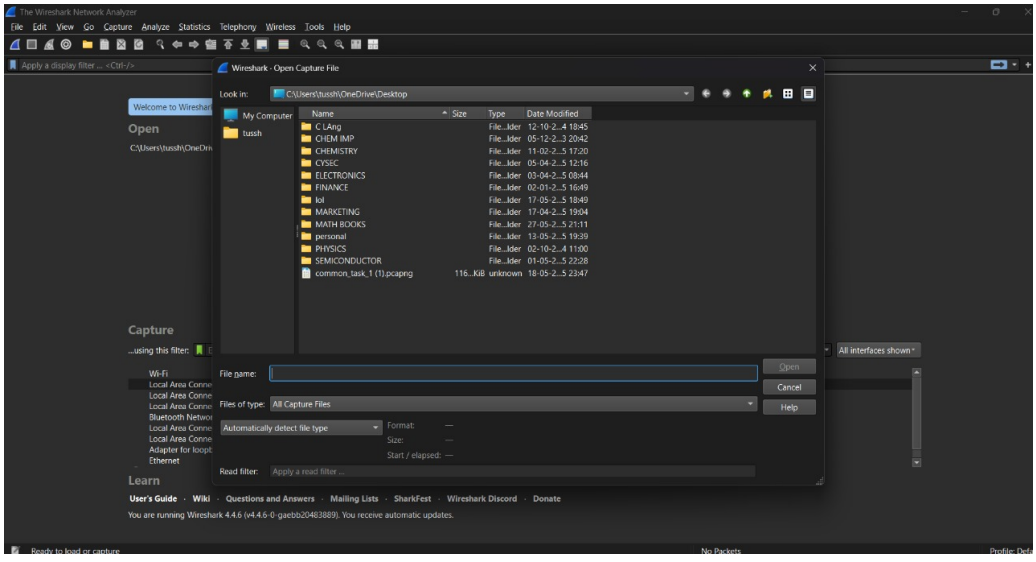

Figure 1: Wireshark interface with loaded PCAP file

3.3 Questions and Answers

1. **What types of traffic (HTTP, DNS, FTP, etc.) are present?**
   To do this we are gonna use the **protocol hierarchy status present in the statistics dropdown**, because this will list all the protocol used and in what hierarchy it is present.
   STEPS: statistics_protocol hierarchy

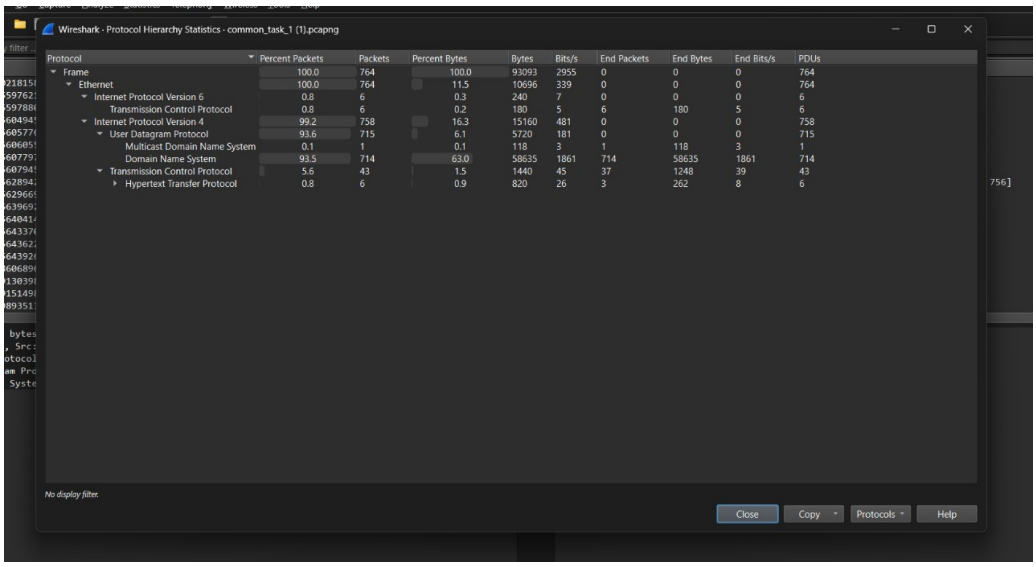

Figure 2: Protocol hierarchy in Wireshark

2. **How many DNS queries were made in total?**
   A **DNS query** is a request sent by a client to a DNS server to resolve a domain name into its corresponding IP address. It is usually sent over UDP port 53 (we can use other ports for a little bit of security) and includes the type of record requested (e.g 'A' for IPv4) and the domain name. The DNS server processes the query and returns a response with the relevant IP address or forwards the query to other servers if needed.
   Here the DNS queries is asked, so we will use our display filter toolbar present on the top to filter out the DNS queries.
   STEPS: command: dns.flags.response==0

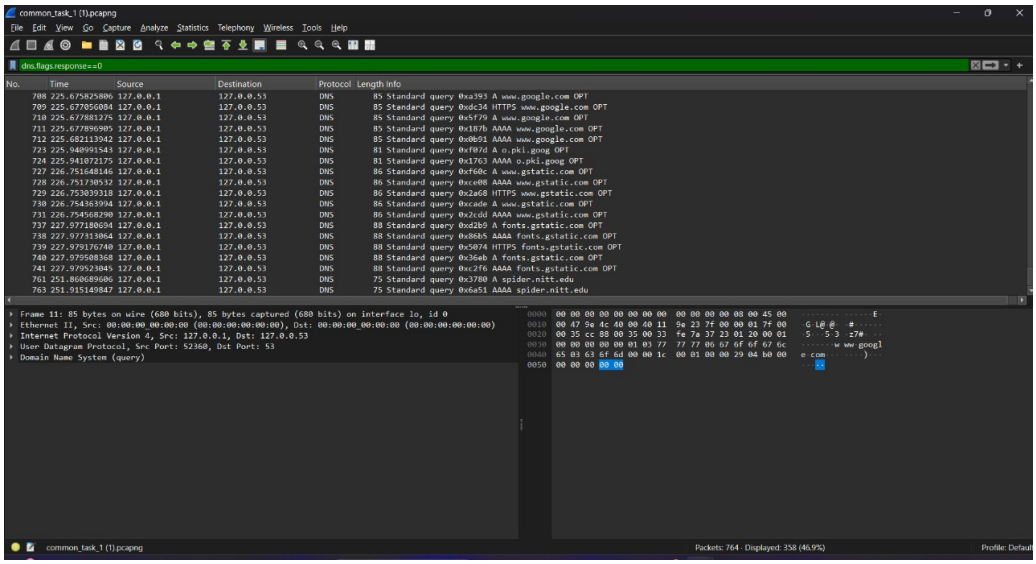

Figure 3: DNS queries filtered in Wireshark

3. **What types of DNS queries were made?**
   STEPS: statistics_DNS_query-response

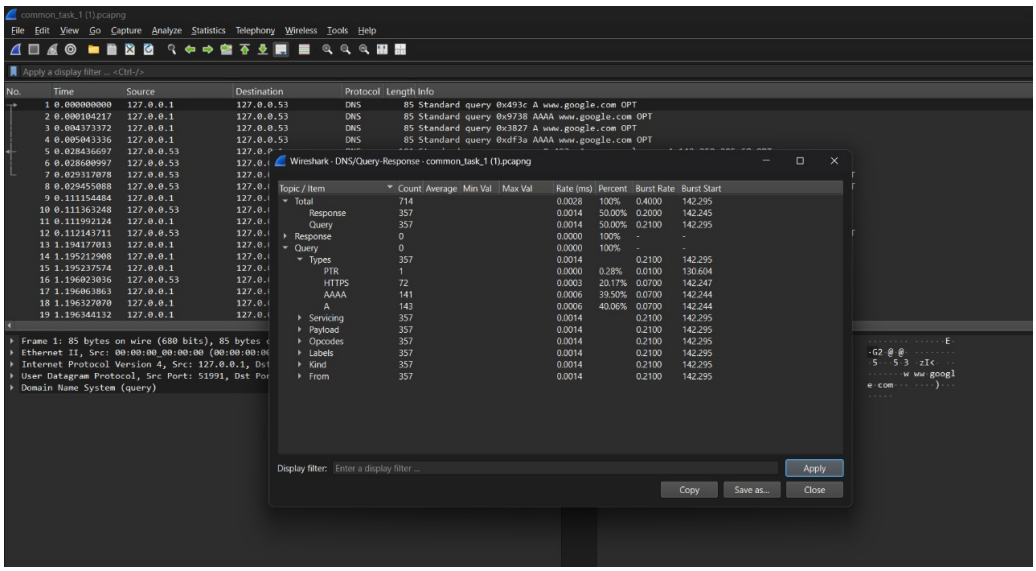

Figure 4: DNS query types

4. **What is a Loopback Interface?**
   A **loopback interface** is a virtual network interface that allows a device to communicate with itself. It is not tied to any physical hardware and always remains active. It typically uses 127.0.0.1 (IPv4) or ::1 (IPv6).
   Why? It is basically used to test if the TCP/UDP protocols in our computer are working properly or not and used for security exercises or a client is fetching files from the server locally.
   HOW TO CHECK? In order to check if the traffic or some part of traffic is running loopback interface is by looking at the destination and source IP address if source and destination IPs are same and 127.0.0.1 then the network interface is loopback interface.
   STEPS: statistics_capture file properties_interfaces

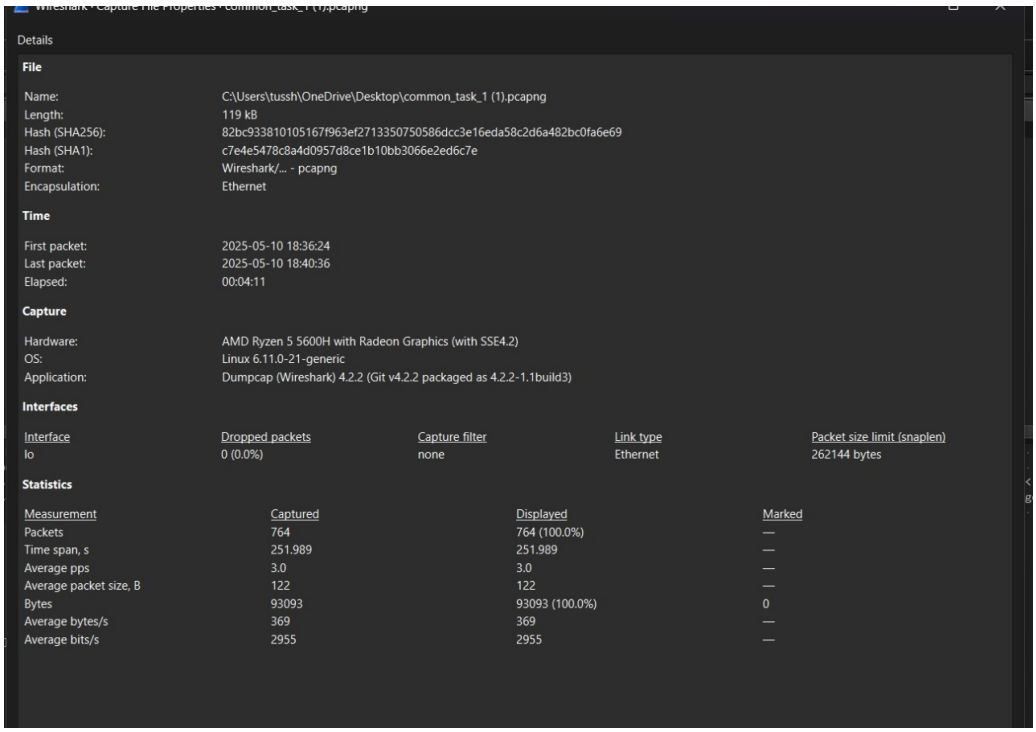

Figure 5: Loopback interface in Wireshark

   As you can see here, under the interface tab it is showing **lo** (aka loopback, since the OS used is Linux, for Windows it shows loopback pseudo interface).
5. **How many .txt files were requested? List their names.**
   Display filter: http.request

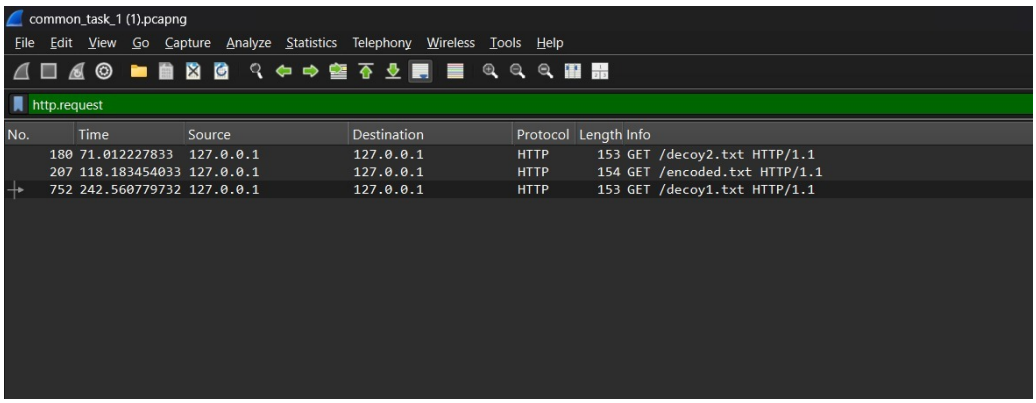

Figure 6: HTTP requests for .txt files

6. **One .txt file contains base64-encoded content. Identify and decode it. What does it contain?**
First of all...what is base64? Base64 is a binary-to-text encoding scheme that converts binary data (like files or raw bytes) into a printable ASCII string format. It is widely used for transmitting data over text-based protocols (e.g., email, HTTP, JSON) where raw binary is not supported.
How do identify a base64 content? In layman terms it looks like a total gibberish with a == or = at the end (also called a padding, it ensures that while encoding, it ends with a multiple of 4).

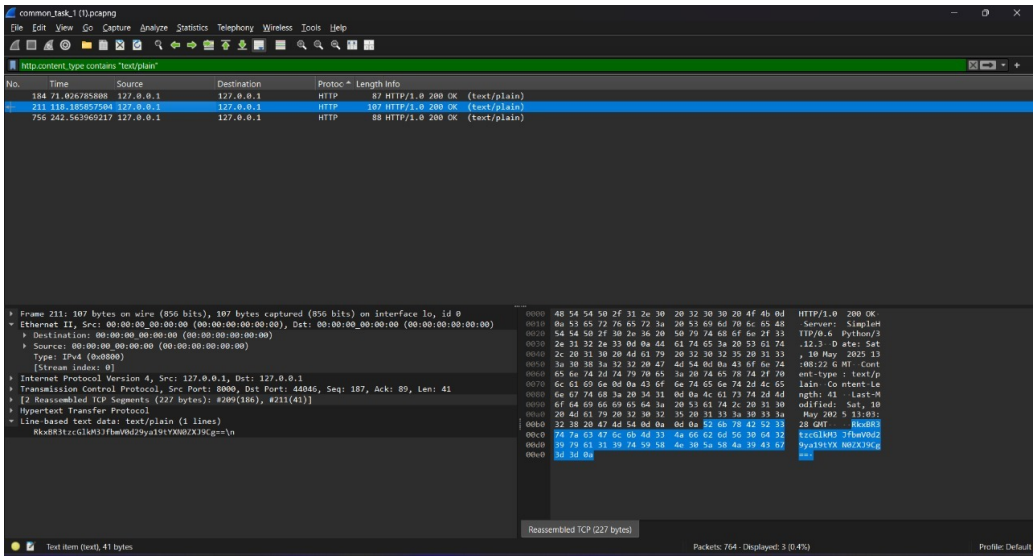

Figure 7: Base64 content in Wireshark

IN OUR PCAP file: It is clear that here, that the second packet transfer (No:211) is a base64 encoded, by looking at the contents in the line based text data.
DECODING IT: We can use software like CyberChef, Base64 encoder etc, to decode it.

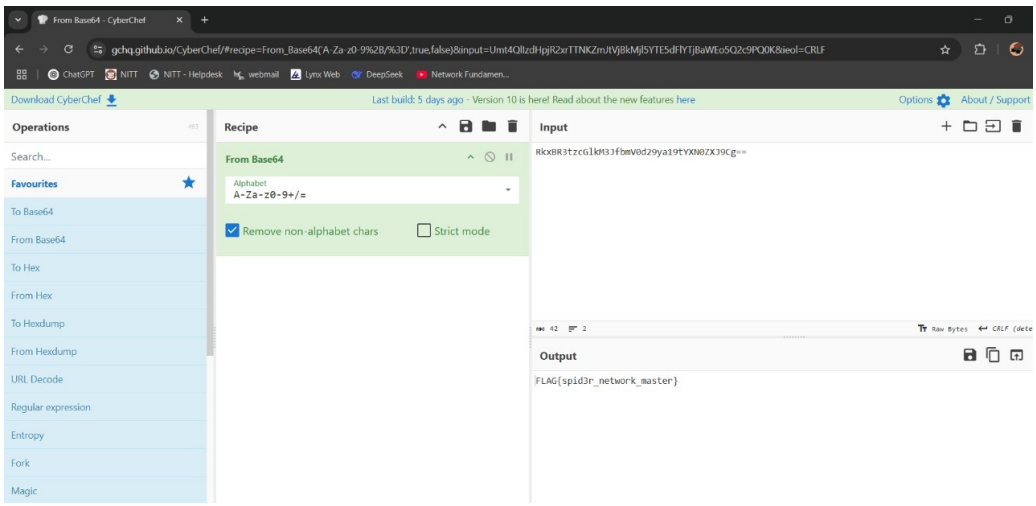

Figure 8: Decoding base64 content

7. **Was any attempt made to distract the analyst using decoy files? Explain.**
   Yes, there were some .txt files that were present to distract the analyst from other files.

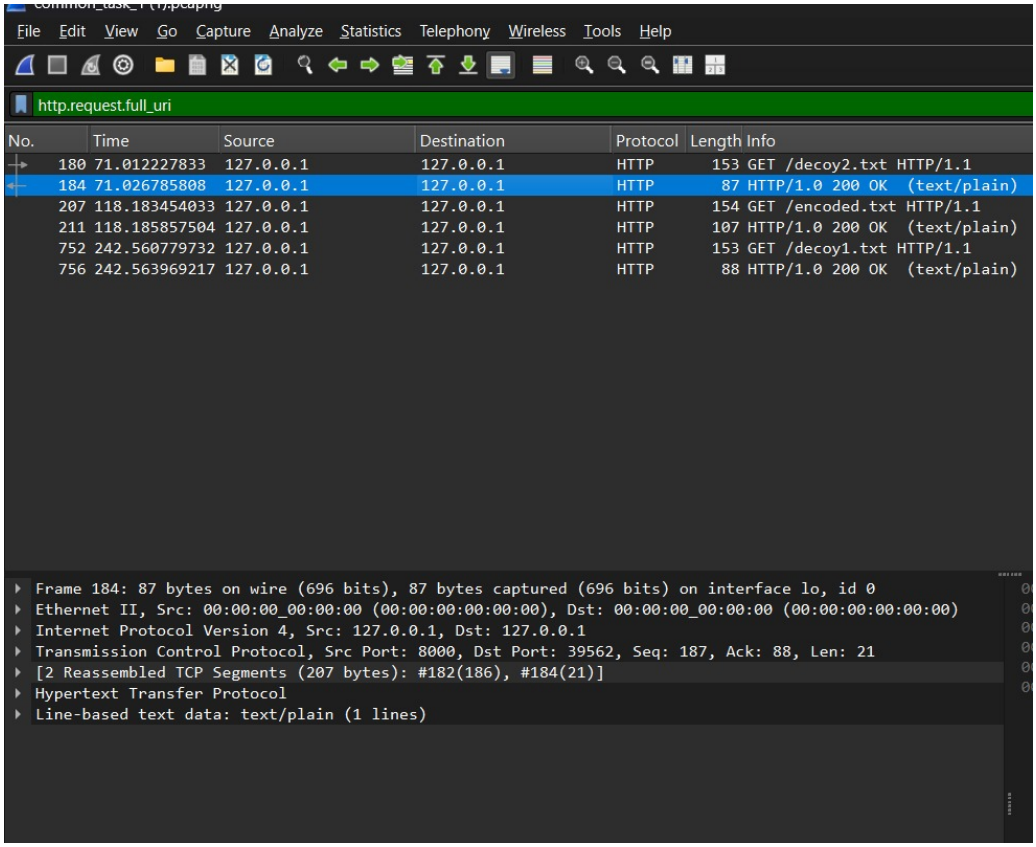

Figure 9: Decoy files in Wireshark

As we can see here, there were two decoy files (decoy1.txt, decoy2.txt), to distract the user analyst.

8. Are there any known ports being used for uncommon services? **No, I don't think there were any known ports used for uncommon services, we can check this by using the display filter and the common ports and its protocols table.**
   DNS/UDP port:53
   TCP/HTTP port:8000
   Although it is fine to use port 8000 for TCP (typically used for dev servers), it is sure rare to see port 8000 is use for plain text file transfer (HTTP).

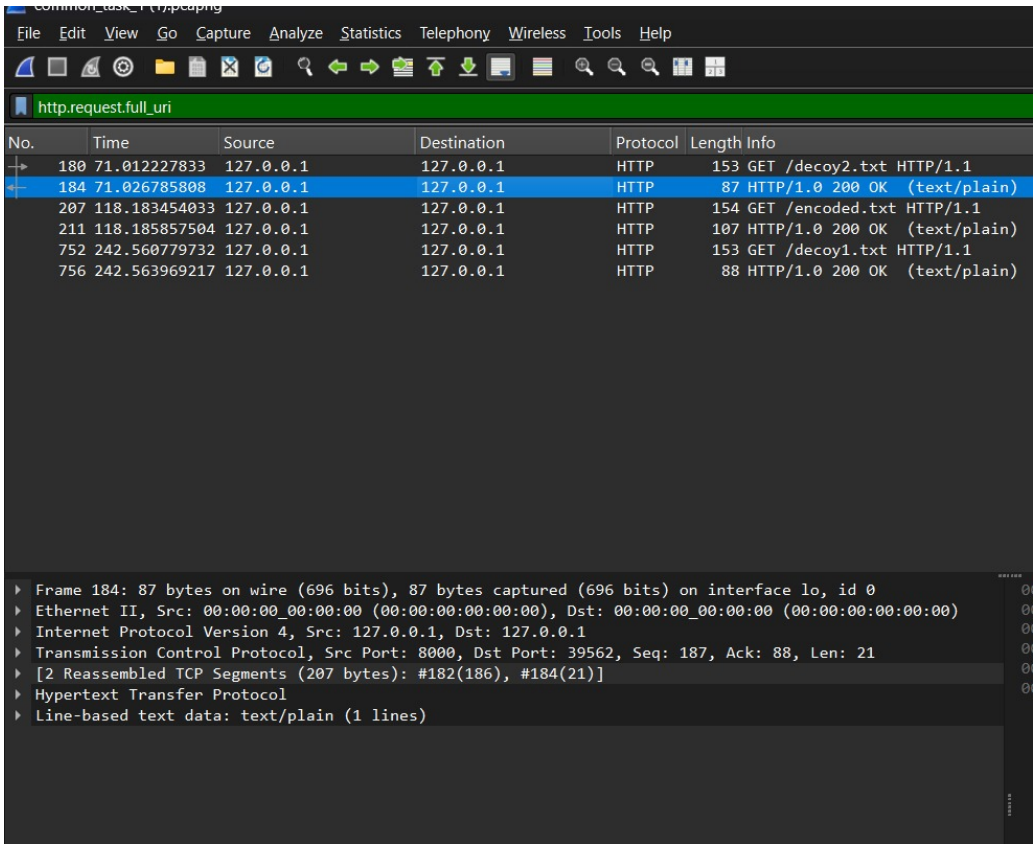

Figure 10: Port usage in Wireshark

9. **How many HTTP GET requests are visible in the capture?**
   Totally 3 HTTP GET requests were made out.
   Display filter command: http.request

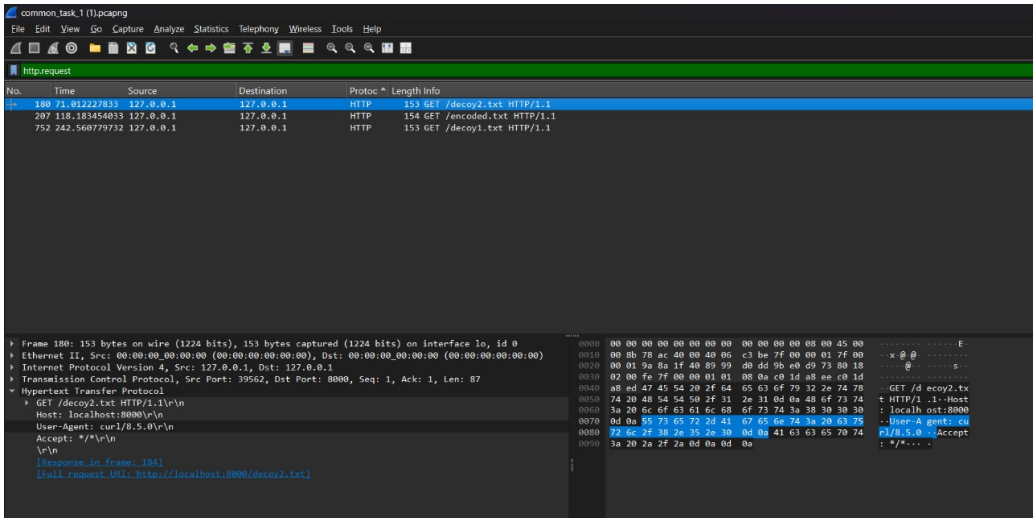

Figure 11: HTTP GET requests in Wireshark

10. **What User-Agent was used to make the HTTP requests?**
    First of all... what is a user-agent? In an HTTP request, the **User-Agent** is a header that identifies the **client software** making the request to the server. This typically includes information about the browser, operating system, device type, and sometimes even the rendering engine. **Think of the User-Agent in an HTTP request like a digital nametag.**

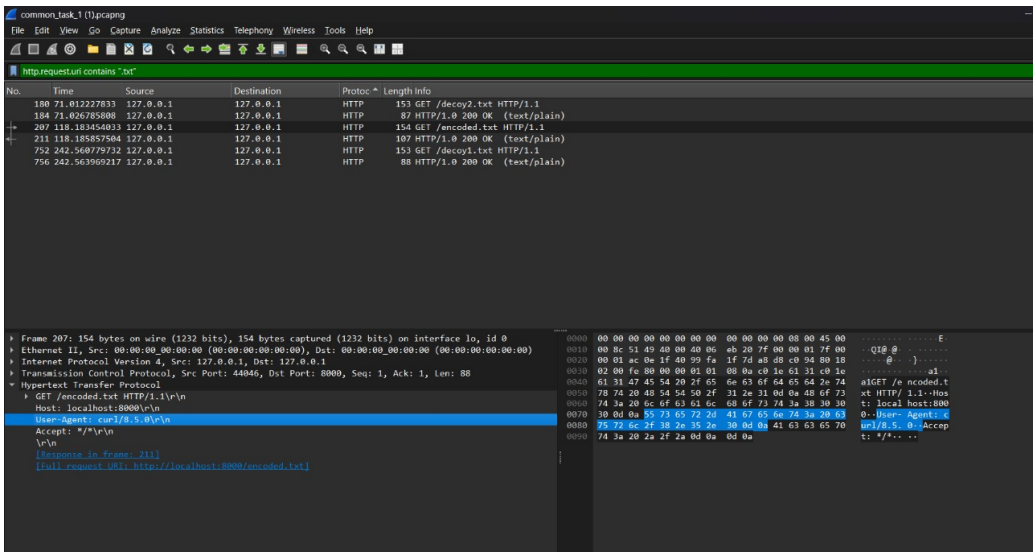

Figure 12: User-Agent in Wireshark

—

4 Basic Task-2: Cybersecurity (Google Dorks)

4.1 Problem Statement

Using Google Dorks, enumerate digital artifacts (documents, credentials, login panels, backups, and configurations) exposed on the internet for a given target organization (perform for at least 3-4 organizations) using publicly available Google search techniques.

4.2 Step 1: Target Setup

You are given a domain: tesla.com

4.3 Step 2: Use Google Dorks to Find and Record the Following:

- Public Documents (PDF, XLSX, DOCX, etc)
- Login Pages / Admin Panels
- Public Backup / Config Files
- Exposed Logs / Errors
- Emails & Contact Info
- Git Folders and Env Files

4.4 Idea

Tesla is a high-profile company with strong security, so traditional recon methods often hit dead ends—especially on public-facing pages like customer service. **But sometimes, digging deeper with the right keywords can uncover some stuff:)**

Since Tesla's main site is locked down, I focused on **uncommon terms** like **"confidential"**, **"internal"**, or **"shareholder meeting"** in Google dorks. Why? Because even secure companies sometimes misplace sensitive files, like **investor reports, internal documents, or archived data, on subdomains or poorly indexed back-end servers**.

4.5 Dorks Used

1. site:tesla.com intext:confidential filetype:pdf
2. site:tesla.com intext:admin filetype:pdf
3. site:tesla.com intext:confidential "login"

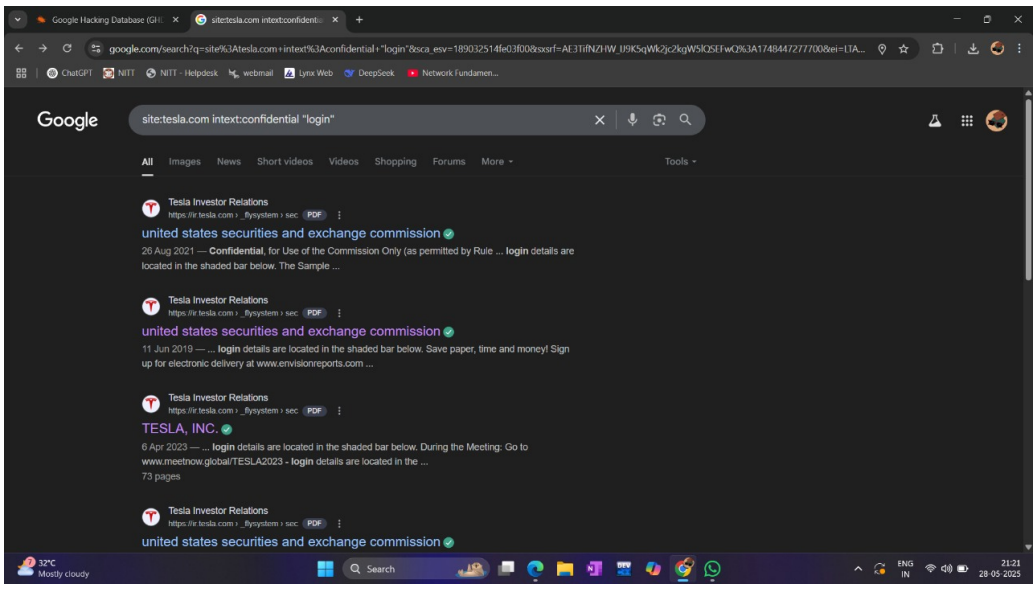

Figure 13: Google Dork results for Tesla

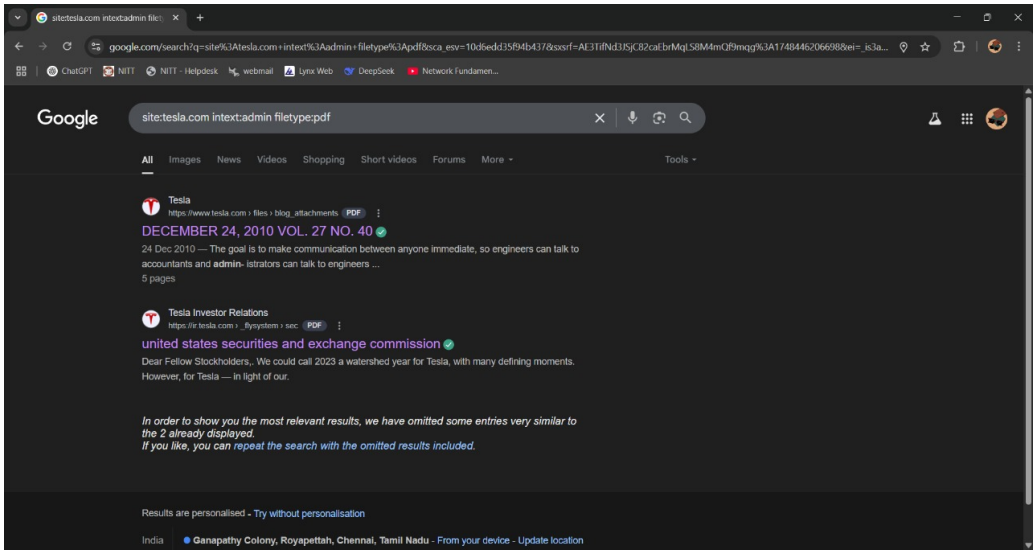

Figure 14: Additional Google Dork results

4.6 Some Links Found

- `https://ir.tesla.com/_flysystem/s3/sec/9999999997200000038/filename1.pdf`

---

5 Basic Task-3: Blockchain (Shamir Secret Scheme)

5.1 Problem Statement

Implement the core logic of Shamir's Secret Sharing Scheme, a fundamental cryptographic method used to split a secret (like a private key) into multiple parts, such that only a subset (threshold) of them is needed to reconstruct the original secret. Implement SSS in your preferred programming language (Python/JavaScript recommended). Use polynomial-based splitting to divide a number (secret) into n shares. Require only k out of n shares to reconstruct the secret. Demonstrate the reconstruction using Lagrange Interpolation.

5.2 Idea

Shamir's Secret Sharing works by hiding the secret inside a randomly generated polynomial where the secret is the constant term. To split the secret, we create a polynomial of degree k-1 (where k is the threshold number of shares needed to reconstruct the secret) and evaluate it at n different x-values to get n shares. Each share is simply a point (x,f(x)) on that polynomial.

The beautiful part is that any k of these points can be used to uniquely reconstruct the original polynomial using **Lagrange interpolation!!**, and evaluating this polynomial at x=0 gives us back the secret.

5.3 How to Implement This Logic in Python?

In Python, we implement this by generating random coefficients for the polynomial, evaluating it to produce shares, and using modular arithmetic and Lagrange interpolation to recover the secret from any k of them.

Note: In order for the SSS to hold we are using a special library called sympy, this generates a random prime number that is greater than a secret.

5.4 The Code

```
import␣random
from␣sympy␣import␣nextprime

#␣Evaluates␣the␣polynomial␣at␣a␣given␣x␣(mod␣prime)
def␣evaluate_polynomial(coeffs,␣x,␣prime):
␣␣␣␣result␣=␣0
␣␣␣␣for␣i,␣coeff␣in␣enumerate(coeffs):
␣␣␣␣␣␣␣␣result␣=␣(result␣+␣coeff␣*␣pow(x,␣i,␣prime))␣%␣prime
␣␣␣␣return␣result

#␣Generates␣n␣shares␣using␣a␣random␣polynomial␣of␣degree␣k-1
def␣generate_shares(secret,␣n,␣k,␣prime):
␣␣␣␣coeffs␣=␣[secret]␣+␣[random.randint(1,␣prime␣-␣1)␣for␣_␣in␣range(k␣-␣1)]
␣␣␣␣shares␣=␣[(x,␣evaluate_polynomial(coeffs,␣x,␣prime))␣for␣x␣in␣range(1,␣n␣+␣1)]
␣␣␣␣return␣shares

#␣Computes␣modular␣inverse␣(used␣for␣division␣in␣finite␣fields)
def␣mod_inverse(a,␣prime):
␣␣␣␣return␣pow(a,␣-1,␣prime)

#␣Reconstructs␣the␣secret␣using␣Lagrange␣interpolation
def␣reconstruct_secret(shares,␣prime):
␣␣␣␣secret␣=␣0
␣␣␣␣k␣=␣len(shares)
␣␣␣␣for␣i␣in␣range(k):
␣␣␣␣␣␣␣␣xi,␣yi␣=␣shares[i]
␣␣␣␣␣␣␣␣li␣=␣1
␣␣␣␣␣␣␣␣for␣j␣in␣range(k):
␣␣␣␣␣␣␣␣␣␣␣␣if␣i␣!=␣j:
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣xj,␣_␣=␣shares[j]
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣numerator␣=␣(0␣-␣xj)␣%␣prime
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣denominator␣=␣(xi␣-␣xj)␣%␣prime
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣li␣*=␣numerator␣*␣mod_inverse(denominator,␣prime)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣li␣%=␣prime
␣␣␣␣␣␣␣␣secret␣=␣(secret␣+␣yi␣*␣li)␣%␣prime
␣␣␣␣return␣secret

def␣main():
␣␣␣␣secret␣=␣int(input("Enter␣the␣secret␣(a␣number):␣"))
␣␣␣␣n␣=␣int(input("How␣many␣total␣shares␣to␣create?␣"))
␣␣␣␣k␣=␣int(input("Minimum␣shares␣needed␣to␣recover␣the␣secret?␣"))

␣␣␣␣if␣k␣>␣n:
␣␣␣␣␣␣␣␣print("Error:␣Threshold␣can't␣be␣greater␣than␣number␣of␣shares.")
␣␣␣␣␣␣␣␣return

␣␣␣␣prime␣=␣nextprime(secret␣+␣100)
␣␣␣␣print(f"Chosen␣prime␣for␣field␣arithmetic:␣{prime}")

␣␣␣␣shares␣=␣generate_shares(secret,␣n,␣k,␣prime)
␣␣␣␣print("\nGenerated␣Shares:")
␣␣␣␣for␣i,␣share␣in␣enumerate(shares,␣start=1):
␣␣␣␣␣␣␣␣print(f"␣Share␣{i}:␣x␣=␣{share[0]},␣y␣=␣{share[1]}")

␣␣␣␣print(f"\nEnter␣any␣{k}␣shares␣to␣reconstruct␣the␣secret:")
```

```
    selected_shares = []
    for i in range(k):
        x = int(input(f"␣Share␣{i+1}␣-␣x:␣"))
        y = int(input(f"␣Share␣{i+1}␣-␣y:␣"))
        selected_shares.append((x, y))

    recovered = reconstruct_secret(selected_shares, prime)
    print(f"\nRecovered␣Secret:␣{recovered}")

if __name__ == "__main__":
    main()
```

5.5 Result

```
Enter the secret (a number): 1234
How many total shares to create? 5
Minimum shares needed to recover the secret? 3
Chosen prime for field arithmetic: 1361

Generated Shares:
 Share 1: x = 1, y = 217
 Share 2: x = 2, y = 858
 Share 3: x = 3, y = 435
 Share 4: x = 4, y = 309
 Share 5: x = 5, y = 480
```