CMR TECHNICAL CAMPUS

UGC AUTONOMOUS

Accredited by NBA & NAAC with 'A' Grade

Approved by AICTE, New Delhi and JNTUH. Hyderabad

Academic Year	:
Name of the Student	:
Roll No	:
Year	: B. Tech I/II/III/IV
Semester	: I/II
Section	:
Branch	:
Name of the Laboratory	:
Batch No.	:
Title of the Lab Report/Project	:
	:
Date	:
Signature of the Student	:

LABORATORY REPORT/PROJECT & PRESENTATION				
Problem Statement & Objectives	Design & Methodology	Implementation & Results	Total Marks	Final Marks
10	15	15	40	10

Remarks/Comments by the Faculty:

Name of the Faculty	:
Signature of the Faculty	:

ABSTRACT

The aim of this project is to design and implement a secure password manager that stores, manages, and synchronizes user credentials while preserving confidentiality and integrity. This system implements client-side cryptography: master passwords are transformed into strong encryption keys using PBKDF2, individual credentials are encrypted with AES-GCM, and each encrypted record is authenticated with Ed25519 digital signatures. A Tkinter-based GUI provides an intuitive interface for adding, viewing, and managing entries; synchronization to a remote server is performed over TLS-protected WebSockets so that the server only ever stores ciphertext and signatures (a zero-knowledge design). The implementation demonstrates practical key derivation, authenticated encryption, signature verification, and secure network transport. It is suitable for real-world demonstration and further extension (multi-user accounts, 2FA, cloud integration), while emphasizing secure coding practices and the ethical handling of sensitive data..

TABLE OF CONTENTS

1. INTRODUCTION

A password manager is a security application that securely stores and manages users' login credentials for various websites and services. Instead of remembering multiple complex passwords, users rely on a master password to access all their stored accounts. This approach enhances both convenience and security by encouraging the use of strong, unique passwords for each service.

This project involves developing a Secure Password Manager that focuses on protecting user data through robust cryptographic techniques. The system uses AES-GCM encryption to ensure confidentiality and authenticity of stored passwords, while PBKDF2 derives a strong encryption key from the user's master password to resist brute-force attacks. Each password entry is digitally signed using Ed25519 signatures to maintain data integrity and detect tampering.

The application includes a Tkinter-based graphical user interface (GUI) that allows users to add, view, and manage their credentials easily. Additionally, it features secure synchronization with a remote server via TLS-encrypted communication, ensuring that only encrypted data is transmitted and stored — the server never has access to the plaintext passwords.

This project demonstrates the real-world application of cryptography and network security concepts, offering an educational platform for understanding secure data storage, encryption standards, and secure client-server communication.

2. LITERATURE SURVEY

In Developing a secure password manager requires understanding existing cryptographic methods, storage techniques, and secure communication standards.

1.Fundamentals of Secure Storage:
Password managers protect user credentials using encryption and key derivation. Algorithms like AES-GCM ensure confidentiality and integrity, while PBKDF2 strengthens weak passwords by generating secure keys from a master password.

2. Existing Password Managers:
Tools such as KeePass, Bitwarden, and 1Password implement local encryption and server synchronization through TLS, following a zero-knowledge architecture, where servers never access plaintext passwords.

3. Cryptographic Techniques:
AES-GCM provides authenticated encryption to prevent tampering. Ed25519 digital signatures verify data integrity and authenticity, ensuring that stored or transmitted data remains unaltered.

4. Secure Communication:
TLS (Transport Layer Security) is widely used to protect data in transit between the client and server, preventing eavesdropping and man-in-the-middle attacks.

5. Ethical and Security Considerations:
Literature emphasizes responsible handling of sensitive data, enforcing strong key management, secure synchronization, and privacy preservation in password management systems.

3. ANALYSIS AND DESIGN

System Analysis

Functional Requirements:

• User Authentication: Secure login using a master password derived into an encryption key via PBKDF2.

• Password Encryption: Store credentials encrypted with AES-GCM for confidentiality and integrity.

• Digital Signature: Verify data authenticity using Ed25519 signatures.

• Data Synchronization: Sync encrypted data with the server using TLS-secured WebSockets.

• GUI Interface: Manage passwords through a user-friendly Tkinter interface.

Non-functional Requirements:

• Security: Ensure strong protection using AES-GCM, PBKDF2, Ed25519, and TLS.

• Performance: Fast encryption and smooth user experience.

• Usability: Simple and intuitive GUI for all users.

• Scalability: Easily expandable for multi-user or cloud-based systems.

• Reliability: Detect tampering and prevent data loss through digital signatures and encrypted backups.

System Design

Architectural Design

Encryption Layer:

- **Purpose:** Securely encrypt and decrypt user passwords to ensure confidentiality.
- **Implementation:** Uses **AES-GCM** for authenticated encryption, providing both data privacy and integrity. Each password is encrypted with a unique nonce and stored as ciphertext in a local JSON database.
- **Tools:** Python's cryptography library for AES-GCM operations.

• **Key Derivation Layer:**

- **Purpose:** Generate a strong symmetric key from the user's master password.
- **Implementation:** Utilizes **PBKDF2** with a random salt and multiple iterations to derive a secure 256-bit key, protecting against brute-force and dictionary attacks.

• **Signature & Verification Layer:**

- **Purpose:** Ensure the authenticity and integrity of encrypted data.
- **Implementation:** Uses **Ed25519** digital signatures to sign encrypted records. Before decryption, each record's signature is verified to detect any tampering.

• **Storage Layer:**

- **Purpose:** Store encrypted credentials locally and maintain database consistency.
- **Implementation:** Encrypted entries and their signatures are saved in a JSON file (db.json). Only ciphertext and metadata are stored—no plaintext is ever written to disk.

• **Network Synchronization Layer:**

- **Purpose:** Synchronize encrypted password data with a remote server securely.
- **Implementation:** Data is transmitted over **TLS-secured WebSockets (wss://)**, ensuring encrypted, authenticated communication. The server stores only ciphertext in storage.json.

• **User Interface Layer:**

- **Purpose:** Provide a simple and intuitive interface for password management.
- **Implementation:** Built using **Tkinter**, offering functions to add, view, and sync entries. The GUI interacts with cryptographic and networking modules to perform secure operations seamlessly.

Flow Diagram

1. Start Sniffing:
   The system initializes packet sniffing on the specified interface using Scapy's sniff() function.

2. Packet Capture:
   Packets are captured in real-time from the network interface and handed to the parsing layer.

3. Protocol Parsing:
   Each packet is parsed to extract:

   - IP Layer: Source/Destination IP and protocol.
   - Transport Layer: TCP/UDP headers.
   - Application Layer: Raw payload data.

4. Filtering:
   User-defined filters are applied to pass only relevant packets to the output stage.

5. Output and Analysis:
   The extracted packet information is displayed in a structured format or logged for further analysis.

4. IMPLEMENTATION

4.1 STEPS TO EXECUTE:

Step 1: Set Up the Environment

1. **Install Scapy Install Required Packages:**
2. Ensure Python 3.8 or above is installed.

   ```
   >>>pip install cryptography tkinter websocket-server
   ```

3. **Generate SSL Certificates (for Secure Communication):** Open PowerShell in the server folder and run:

   ```
   >>>openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout server.key -out server.crt -subj "/C=US/ST=State/L=City/O=SecurePM/OU=Dev/CN=localhost"
   ```

Step 2: Start the server

1. **Navigate to the server directory:** cd server.
2. Run the secure server: python server.py
3. The server listens on a **TLS-secured WebSocket** port and waits for client connections.

Step 3: Prepare the Client

1. **Move to the client directory:**.

   ```
   >>cd ../client
   ```

2. **Ensure the following files exist:**

   >>gui_client.py – GUI program for password management

   >>crypto_utils.py – Cryptographic operations (AES-GCM, PBKDF2, Ed25519)

   >>db.json – Local encrypted database (automatically created on first run)

Step 4: Run the GUI Client

1. **Execute the GUI application:**

   ```
   >>python gui_client.py
   ```

2. Enter a master password when prompted (used to derive the encryption key via PBKDF2).

Step 5: Use the Application

Add a Password Entry:

- Input website, username, and password → click "Add".
- The password is encrypted with AES-GCM, signed with Ed25519, and stored locally.

View Stored Passwords:

- The app decrypts and displays entries securely after verifying digital signatures.

Sync with Server:

- Click "Sync" to send encrypted data to the server via TLS.
- The server stores only ciphertext — no plaintext passwords.

Step 6: Validate Execution

1. Verify that the GUI is functional and server logs show a secure connection.
2. Check db.json and storage.json — both should contain encrypted ciphertext, not readable text.
3. Modify, add, or sync new entries to confirm successful encryption, signature validation, and synchronization.

4.2 CODE

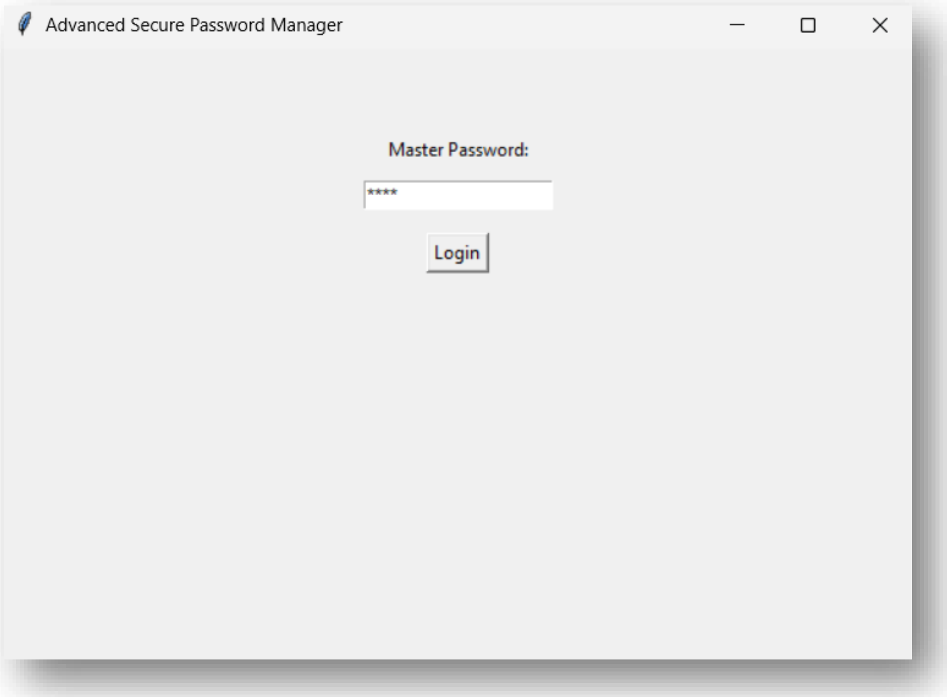
Server .py

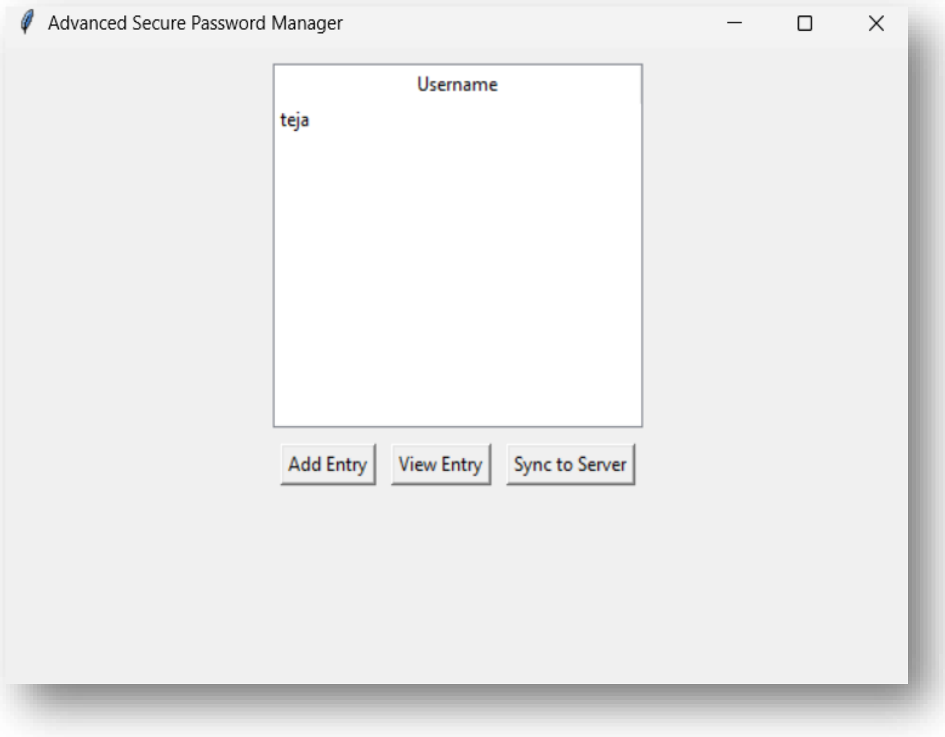
```
import import asyncio
import json
import ssl
import websockets

STORAGE_FILE = "storage.json"

async def handler(ws):
    async for msg in ws:
        db = json.loads(msg)
        # Store encrypted database (ciphertext + signatures)
        with open(STORAGE_FILE, "w") as f:
            json.dump(db, f)
        await ws.send(json.dumps({"status": "ok"}))
        print("[+] Received and stored encrypted data")

sslctx = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
sslctx.load_cert_chain(certfile="server.crt", keyfile="server.key")

start_server = websockets.serve(handler, "0.0.0.0", 8765, ssl=sslctx)
print("[Server] Listening on wss://localhost:8765")
asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

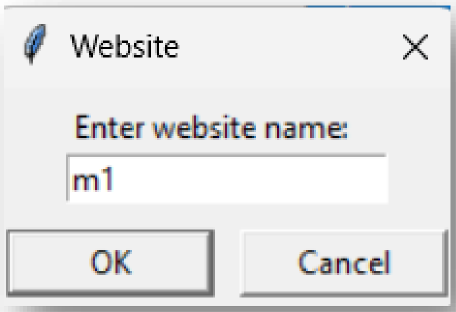
gui_client.py

```
class PasswordManagerApp(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("Advanced Secure Password Manager")
        self.geometry("600x400")
        self.db = load_db()
        self.aes_key = None
        self.priv = None
        self.pub = None

        self.create_login_frame()

    def create_login_frame(self):
        self.login_frame = tk.Frame(self)
        self.login_frame.pack(pady=50)

        tk.Label(self.login_frame, text="Master Password:").pack(pady=5)
        self.master_entry = tk.Entry(self.login_frame, show="*")
        self.master_entry.pack(pady=5)
        tk.Button(self.login_frame, text="Login", command=self.login).pack(pady=10)

    def login(self):
        master = self.master_entry.get()
        if not master:
            messagebox.showerror("Error", "Enter master password")
            return

        # Load or create salt
        salt_file = "salt.bin"
        if os.path.exists(salt_file):
            with open(salt_file, "rb") as f:
                salt = f.read()
        else:
            salt = os.urandom(16)
            with open(salt_file, "wb") as f:
                f.write(salt)
        self.aes_key = derive_aes_key(master, salt)
```

5. TESTING AND DEBUGGING

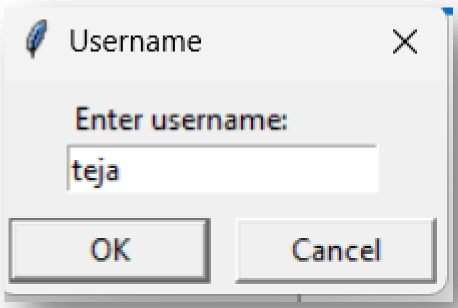





Website

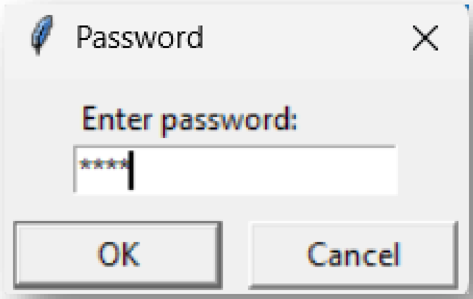
Enter website name:

m1


Username

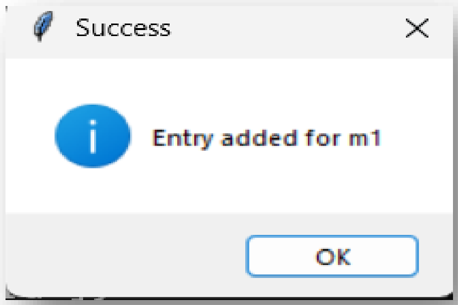
Enter username:

teja

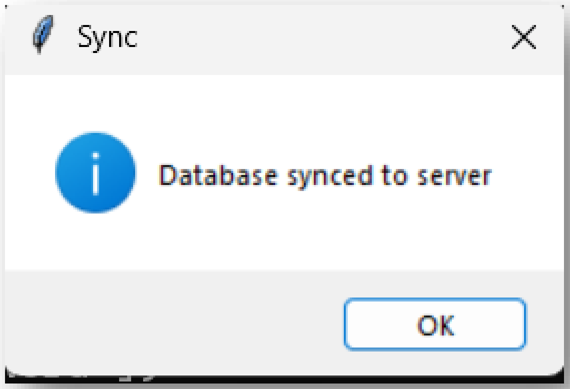

Password

Enter password:

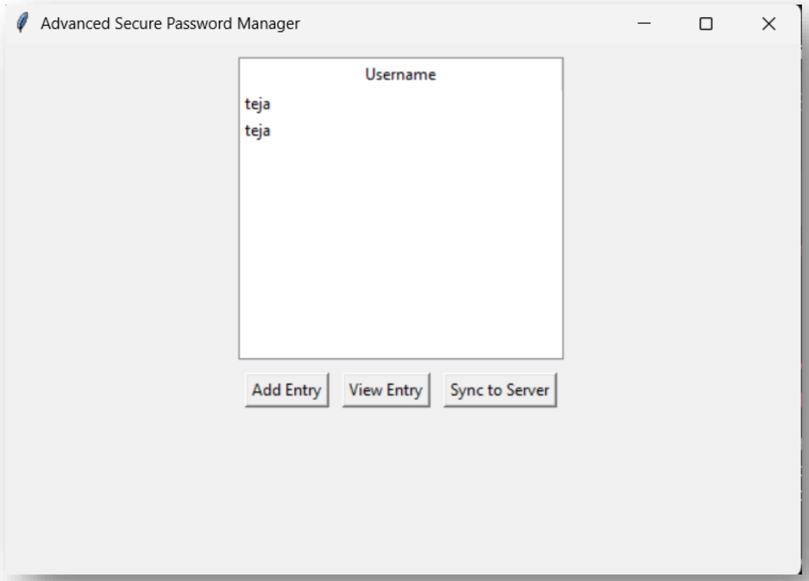
****

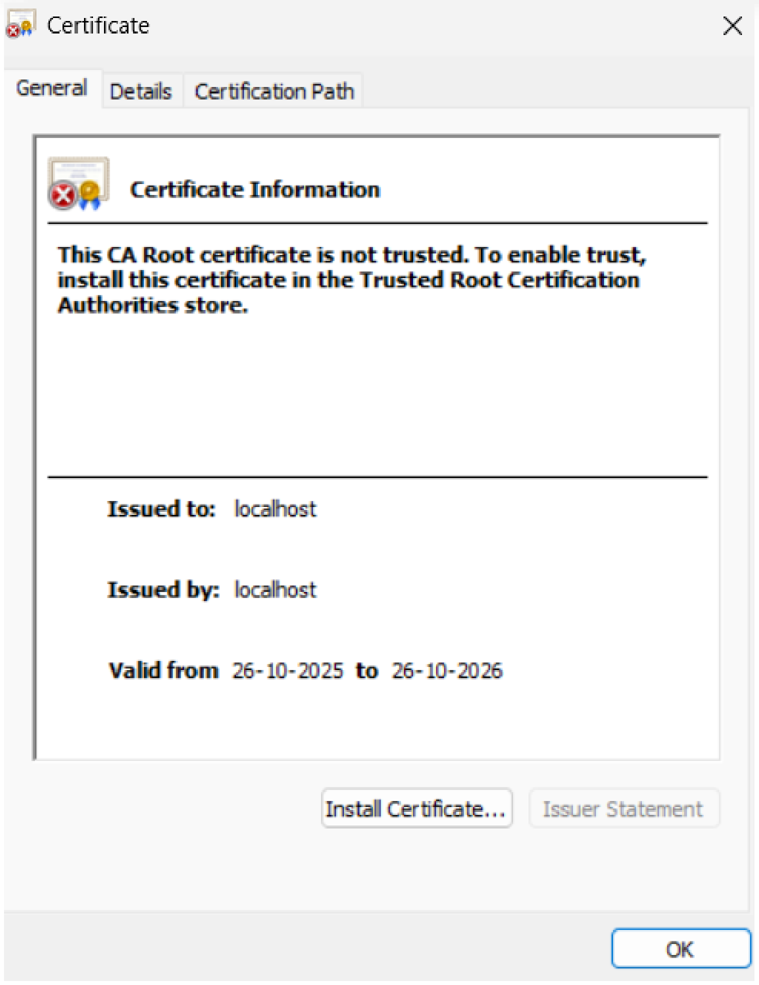

Success

Entry added for m1


Sync

Database synced to server





This CA Root certificate is not trusted. To enable trust, install this certificate in the Trusted Root Certification Authorities store.

Issued to: localhost

Issued by: localhost

Valid from 26-10-2025 **to** 26-10-2026

6. CONCLUSION

In conclusion, the Secure Password Manager developed in this project effectively protects user credentials by employing modern cryptographic techniques and secure communication protocols. It ensures confidentiality through AES-GCM encryption, integrity via Ed25519 digital signatures, and secure data transmission using TLS. Passwords are encrypted and signed locally before being synchronized to the server, which stores only encrypted data, demonstrating a zero-knowledge approach.

This project serves as both a practical tool and an educational example, illustrating how cryptography can be applied to real-world applications to safeguard sensitive information. While the system provides strong security guarantees, users must continue to follow best practices, such as using strong master passwords and keeping devices secure. The implementation highlights how modern cryptography and secure design principles can be combined to build reliable and trustworthy software, offering foundational insights for anyone interested in cybersecurity or secure software development.

7. REFERENCES

Books

 Seitz, J., & Arnold, T. (Year). Black Hat Python: Python Programming for Hackers and Pentesters. [Publisher].

- Explores Python-based security tools, including network scanning, packet analysis, and penetration testing scripts.

 O'Connor, T. J. (Year). Violent Python: A Cookbook for Hackers, Forensic Analysts, Penetration Testers, and Security Engineers. [Publisher].

- Provides practical Python scripts and methodologies for cybersecurity professionals, including encryption and network tasks.

 Ferguson, N., Schneier, B., & Kohno, T. (2010). Cryptography Engineering: Design Principles and Practical Applications. Wiley.

- Covers the design and implementation of secure cryptographic systems, including encryption algorithms, digital signatures, and key management.

 Paar, C., & Pelzl, J. (2010). Understanding Cryptography: A Textbook for Students and Practitioners. Springer.

- Provides a clear explanation of modern cryptographic techniques such as AES, RSA, and digital signatures.

 Stallings, W. (2021). Cryptography and Network Security: Principles and Practice. Pearson.

- Comprehensive guide on cryptographic algorithms, protocols, and secure system design.

Research Papers & Journals

1. Dierks, T., & Rescorla, E. (2008). *The Transport Layer Security (TLS) Protocol Version 1.2*. IETF RFC 5246.
   - Standard specification for TLS, detailing secure communication protocols used in client-server encryption.
2. Bernstein, D. J., Duif, N., Lange, T., Schwabe, P., & Yang, B.-Y. (2012). *High-speed high-security signatures*. Journal of Cryptographic Engineering, 2(2), 77–89.
   - Discusses the Ed25519 signature algorithm, used for ensuring data integrity in secure systems.

Websites and Online Resources

1. **Scapy Documentation** – Scapy's official documentation.
    - Essential for understanding packet structures, network traffic analysis, and building Python-based network tools.
    - https://scapy.readthedocs.io
2. **Real Python – Network Programming Tutorials**
    - Offers tutorials on Python networking, socket programming, and packet manipulation.
    - https://realpython.com
3. **GeeksforGeeks – Networking with Python**
    - Provides step-by-step examples of network programming and packet inspection using Python libraries.
    - https://www.geeksforgeeks.org
4. **Wireshark.org – Wireshark Documentation and Labs**
    - Useful for understanding packet analysis concepts, network protocols, and traffic inspection techniques.
    - https://www.wireshark.org
5. **Python Cryptography Toolkit (PyCryptodome) Documentation**
    - Reference for implementing AES, digital signatures, and other cryptographic operations in Python.
    - https://www.pycryptodome.org
6. **OWASP Cryptographic Storage Cheat Sheet**
    - Provides best practices for secure password storage, encryption, and key management.
    - https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html
7. **Mozilla Developer Network (MDN) – Web Security Guidelines**
    - Offers guidance on TLS, HTTPS, and secure client-server communications.
    - https://developer.mozilla.org/en-US/docs/Web/Security
8. **NIST Special Publication 800-63B: Digital Identity Guidelines**
    - Provides recommendations for secure password management, storage, and authentication practices.
    - https://pages.nist.gov/800-63-3/sp800-63b.html