Authentication and Access Control

This lab focuses on building a full authentication system as it concerns with the integration of how these components such as password strength analyses, secure hashing, implemneting hashing with salt & paper, adding TOTP-based 2FA, simulation of brute force attack inorder to have robust authentication.

Password Strength Analysis

Purpose

This step concerns with password quality evalution using metrics like length, character variety, entropy estimation all of this while reflecting the workshop's outlined requirements for understanding the fundamental password strength principles.

Practical Implementation

- A password meter has been built that checks for character variety in a password e.g. if it includes lowercase, uppercase, digits and symbols.
- It also checks for the length of the passwords and assigns points for 8 and 12 character long passwords.
- Entropy score also get calculated using formula entropy = length * log(pool_size)
- This passwords meter will reject the common weak passwords as well.

```
!pip install bcrypt
import bcrypt
import math
import string

def strength_checker(password: str):
    """Checking the password's strength based on its length and character variety"""

    # Checking the variety of characters
    has_lower = any(c.islower() for c in password)
    has_upper = any(c.isupper() for c in password)
    has_num = any(c.isdigit() for c in password)
    has_symbol = any(c in string.punctuation for c in password)

    score = 0

    # Length points
    if len(password) >= 8:
        score += 1
    if len(password) >= 12:
        score += 1

    # Score based on character variety
    score += has_lower + has_upper + has_num + has_symbol

    # Pool size estimation for entropy
    pool_size = 0
    if has_lower:
        pool_size += 26
    if has_upper:
        pool_size += 26
    if has_num:
        pool_size += 10
    if has_symbol:
        pool_size += len(string.punctuation)

    # Calulating measure of randomness or unpredictability
    if pool_size > 0:
        entropy = len(password) * math.log2(pool_size)
    else:
        entropy = 0

    # Checking again week and common passwords
    COMMON_PASSWORDS = {'ihateyou', 'admin123' , 'password'}
    if password.lower() in COMMON_PASSWORDS:
        print('week password')
        score = 0

    return score, round(entropy, 6)
```

```
    return score, round(entropy, 3)


# Taking the user input for treating it as password
user_input = input('Enter password: ').strip()

# Checking the password strength, storing the result and printing it out
score, entropy = strength_checker(user_input)
print(f"Score : {score}, Entropy : {entropy} bits")
```

```
Requirement already satisfied: bcrypt in /usr/local/lib/python3.12/dist-packages (5.0.0)
Enter password: uhwohohfod
Score : 2, Entropy : 47.004397 bits
```

Reflection

Entropy increases only when both length and character pool expand. Thus, this is the reason why long passwords but with low variety are classified as weak. This lab notes also points out the limitations of this approach, particularly its inability to detect predictable and common dictionary words which remain vulnerable even if the entropy score is high.

Password Hashing Methods

Purpose

It basically compares fast hashing algorithms like MD5, SHA-256 with secure but slow hashing functions such as bycrypt.

Practical Implementation

The below fucntions contains logic for hashing with MD5, SHA-256, and bycrypt, and for verifying bcrypt hashes using checkpw() - function. Bycrypt automatically generates and embeds salts, cost factor, and version in a single formatted string.

```
!pip install bcrypt
import hashlib
import bcrypt

# Hashing with md5
def hash_md5(password: str) -> str:
    """Hashing the password with MD5"""
    return hashlib.md5(password.encode('utf-8')).hexdigest()

# Hashing with sha256
def hash_sha256(password: str) -> str:
    """Hashing the password with sha256"""
    return hashlib.sha256(password.encode('utf-8')).hexdigest()

# Hashing with bcrypt
def hash_bcrypt(password: str) -> bytes:
    """Hashing the password with bcrypt with random salt"""
    return bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())

# For verifying the user's password hash against stored hash
def verify_bcrypt(password: str, stored_hash: bytes) -> bool:
    """Verify password against stored bcrypt hash"""
    return bcrypt.checkpw(password.encode('utf-8'), stored_hash)


# Stored password hash
stored_password = '13342,Fdwfer3r'
stored_hash = hash_bcrypt(stored_password)

# Taking user input password for verification of hashes
user_input = input('Enter Password: ').strip()

# Verifying the user's input password hash with the stored hash
if verify_bcrypt(user_input, stored_hash):
    print('Correct password - verified')
else:
    print('Incorrect password - not verified')
```

```
Collecting bcrypt
  Downloading bcrypt-5.0.0-cp39-abi3-manylinux_2_34_x86_64.whl.metadata (10 kB)
Downloading bcrypt-5.0.0-cp39-abi3-manylinux_2_34_x86_64.whl (278 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 278.2/278.2 kB 6.8 MB/s eta 0:00:00
Installing collected packages: bcrypt
Successfully installed bcrypt-5.0.0
Enter Password: 445354
Incorrect password - not verified
```

Reflection

Algorithms like MD5 and SHA-256 are extremely fast which makes brute force attacks far more easier. And the lab outlines not to use these fast algorithms for password storage, instead more secure but slow algorithm bycrypt should be used. And as bycrypt is slow and adjustable through the cost factor, thus this makes it significantly harder for attackers to crack the hashes.

Salt and Pepper Implementation

Purpose

It shows why salting is essential to defend against rainbow tables and how pepper adds an additional layer of security on top of salting.

Practical Implementation

The differences between hashing with and without salting & pepper are clearly demonstrated.

- Hashing without salt - identical output for identical inputs no matter how many times we repeat the hashing process.
- Hashing with random salt - different output every time for identical inputs.
- Hashing with salt + paper - adds a secret stored in the environment variable and adds a salt and then produces the hash.

```
import hashlib
import secrets
import os

# Hashing with sha256 algorithm
def hash(password: bytes) -> str:
    return hashlib.sha256(password).hexdigest()

# Generating a random salt value of 16-bytes
def generate_salt(nbytes: int = 16) -> bytes:
    return secrets.token_bytes(nbytes)

# Storing pepper value in an variable
DEFAULT_PEPPER = "abcxyz"
# Getting pepper value from the environment, if it is not found then using the local variable
def get_pepper():
    return os.environ.get('DEMO_PEPPER', DEFAULT_PEPPER)



# Hash without salt twice gives same output
password = '13342,Fdwfer3r'
h1 = hash(password.encode('utf-8'))
h2 = hash(password.encode('utf-8'))
print('Hashing same password twice without salt: ', h1, h2)

# Hash with salt twice gives different output cause salt was different each time
salt_a = generate_salt()
salt_b = generate_salt()
h1_salt = hash(password.encode('utf-8') + salt_a)
h2_salt = hash(password.encode('utf-8') + salt_b)
print('Hashing same password twice with random salt: ', h1_salt, h2_salt)

# Hash with salt and pepper
salt_c = generate_salt()
pepper = get_pepper()
hash_salt_pepper = hash(password.encode('utf-8') + salt_c + pepper.encode('utf-8'))
print('Hashing password with salt & pepper: ' ,hash_salt_pepper)
```

```
Hashing same password twice without salt:  4a05cae2d58227e47232a104be23e31dbf461580394318b02cb3db59518368f3 4a05cae2d58227e4
Hashing same password twice with random salt:  ea9929e81135dd5f8fea9daa3dfafafec1630e8e9e6af596080fb625d673c961 837819f19410
Hashing password with salt & pepper:  05d5eb13dccef21ca7bdb69b4d22ce791fb3774261ed8b37d307abf2a60cd4d4
```

Reflection

Unsalted SHA-256 allows precomputed lookup attacks because the same password always the same hash value. Adding salt per user actually defeats the rainbow tables. Pepper must be stored separately from the database or otherwise the attacker can still compute the hash.

TOTP (2FA) Implementation

Purpose

Here in this step, Time-based One-Time passcode is implemented to provide 2 factor authentication.

Practical Implementation

In our implementation, we did the following things:

- Generated a secure Base32 secret
- Created a provisioning URi
- Generated a QR code (saved as totop_setup.png)
- Verifies user-supplied OTP codes

```
!pip install pyotp
!pip install qrcode
import pyotp
import qrcode
import base64
import time


# Server Setup
def generate_totp_secret():

    # Generating a random base32 secret
    secret = pyotp.random_base32()
    # Creating a TOTP object based on that secret
    totp = pyotp.TOTP(secret)

    # URI encodes the secret, issuer and account info
    uri = totp.provisioning_uri(
        name = 'example.com',
        issuer_name = 'Secure-Authenticar'
    )

    # Generating the qrcode & saving that image
    img = qrcode.make(uri)
    img.save('totp_setup.png')

    return secret


# verification phase
def verifytotp(secret, user_otp):

    # Generarting TOTP object from shared code
    totp = pyotp.TOTP(secret)
    # Verifying the OTP against the TOTP generated from the shared secret
    return totp.verify(user_otp)


# Authentication Demo
if __name__ == "__main__":

    # Generating TOTP Object based on the shared secret and corresponding QR code
    shared_secret = generate_totp_secret()

    # Continue authentication process until user enters a valid OTP
    while True:

        # Just for testing - printing the current server OTP
        current_server_otp = pyotp.TOTP(shared_secret).now()
        print('for reference current_server_otp: ', current_server_otp)

        # For taking user input of OTP
        user_otp_input = input('enter otp:')

        # For verifying the user otp
        if verifytotp(shared_secret, user_otp_input):
            print('OTP Verified')
            break
        else:
            print('OTP Unsuccessful')
```

```
        # Continue the authentication process - wait for the next TOTP time window
        print('\nWaiting for the next OTP window... \n')
        time.sleep(30)
```

```
Requirement already satisfied: pyotp in /usr/local/lib/python3.12/dist-packages (2.9.0)
Requirement already satisfied: qrcode in /usr/local/lib/python3.12/dist-packages (8.2)
for reference current_server_otp:  980852
enter otp:178227
OTP Unsuccessful

Waiting for the next OTP window...

for reference current_server_otp:  173263
enter otp:954708
OTP Verified
```

Reflection

This lab hightlights that TOTP is stronger than SMS-based 2FA because SMS can be intercepted or hijacked through SIM-swapping. The time-windows synchronisation and shared secret ensures resilience to interception.

Brute Force Attack Simulation

Purpose

It demonstrates why fast hashing fuctions allow extremely efficient brute-force attacks.

Implementation Summary

A dictionary attack simulation has been implemented that hashes passwords using either MD5 or SHA-256 and compares them with a target hash.

```
import hashlib
import time
from typing import List, Optional, Tuple

# Small list of common passwords for the demo
COMMON_PASSWORDS: List[str] = [
    "123456", "password", "123456789", "12345678", "12345",
    "111111", "1234567", "sunshine", "qwerty", "iloveyou",
    "admin", "welcome", "monkey", "dragon", "letmein",
    "football", "baseball", "password1", "abc123", "trustno1"
]

# Hashing Sha256 Algorithm
def hashing_sha256(password: str) -> str:
    return hashlib.sha256(password.encode('utf-8')).hexdigest()

# Hashing md5 Algorithm
def hashing_md5(password: str) -> str:
    return hashlib.md5(password.encode('utf-8')).hexdigest()

# Finding the password hash within the common passwords hash based on the specific hash type
def dictAttack(target_hash: str, candidates: List[str], hash_type: str) -> Tuple[bool, Optional[str], int, float]:

    # keeping track of attempts
    attempts = 0
    # recording the starting time
    start = time.perf_counter()

    # going through all the passwords in common password list
    for i in candidates:
        # increasing the attempt count as we go through the common passwords list
        attempts += 1

        # determining the hashing method
        if hash_type == 'md5':
            candidate_hash = hashing_md5(i)
        elif hash_type == 'sha256':
            candidate_hash = hashing_sha256(i)
        else:
            raise ValueError(f'Unsupported hash_type: {hash_type}')

        # checking given password hash against the common password hashes
        if candidate_hash == target_hash:
```

```
            # counting the time it took to perform the whole action if hash is found
            end = time.perf_counter()
            elapsed = end - start

            # returning the result
            return True, i, attempts, elapsed

    # counting the time it took to perform the whole action if hash is not found
    end = time.perf_counter()
    elapsed = end - start

    # returning the result
    return False, None, attempts, elapsed



# Taking user input password and hash_type to find the corresponding hash in the common passwords
pass1 = input('Enter Password: ').strip()
hash_method = input('Enter hash_method (md5 / sha256):').strip()
target_hash = hashing_md5(pass1) if hash_method== 'md5' else hashing_sha256(pass1)

# Storing the result and printing it out
found, matched, attempt, elapsed = dictAttack(target_hash, COMMON_PASSWORDS, hash_method)
print(f'Found: {found}, Matched: {matched}, Attempts: {attempt}, Elapsed: {elapsed:.6f}s')
```

```
Enter Password: monkey
Enter hash_method (md5 / sha256):sha256
Found: True, Matched: monkey, Attempts: 13, Elapsed: 0.000025s
```

Reflection

The extemely small elapsed time proves us that why fast hashing is unsafe for password storage. This lab highlights that bycrypt is intentionally slow which makes brute force impractical. Thus, this aligns with the real-world need for the adoption of adaptive, computationaly expensive hashing functions.

Complete Authentication System

```
# Complete Authentication System
# This covers: password strength checking, bcrypt hashing, pepper usage,
# TOTP setup + verification, and final login process.

import bcrypt
import pyotp
import math
import string
import qrcode

# ---------------------------------------------------------
# Password Strength Checker
# ---------------------------------------------------------
def strength_checker(password: str):
    """Checking the password's strength based on its length and character variety"""

    # Checking the variety of characters
    has_lower = any(c.islower() for c in password)
    has_upper = any(c.isupper() for c in password)
    has_num = any(c.isdigit() for c in password)
    has_symbol = any(c in string.punctuation for c in password)

    score = 0

    # Length points
    if len(password) >= 8:
        score += 1
    if len(password) >= 12:
        score += 1

    # Score based on character variety
    score += has_lower + has_upper + has_num + has_symbol

    # Pool size estimation for entropy
    pool_size = 0
    if has_lower: pool_size += 26
    if has_upper: pool_size += 26
    if has_num: pool_size += 10
    if has_symbol: pool_size += len(string.punctuation)

    # Calculating entropy
```

```
    if pool_size > 0:
        entropy = len(password) * math.log2(pool_size)
    else:
        entropy = 0

    # Checking again weak and common passwords
    COMMON_PASSWORDS = {'password', 'admin123', '123456', 'letmein'}
    if password.lower() in COMMON_PASSWORDS:
        score = 0
        entropy = 0

    return score, round(entropy, 6)


# ---------------------------------------------------------
# Hashing Functions (bcrypt + pepper)
# ---------------------------------------------------------
DEFAULT_PEPPER = "SECRET_PEPPER"

def hash_password(password: str, pepper: str = DEFAULT_PEPPER) -> bytes:
    """Hash the password using bcrypt and pepper."""
    combined = password + pepper
    return bcrypt.hashpw(combined.encode('utf-8'), bcrypt.gensalt())

def verify_password(password: str, stored_hash: bytes, pepper: str = DEFAULT_PEPPER) -> bool:
    """Verify the password using bcrypt + pepper."""
    combined = password + pepper
    return bcrypt.checkpw(combined.encode('utf-8'), stored_hash)


# ---------------------------------------------------------
# TOTP (2FA) Setup
# ---------------------------------------------------------
def setup_totp(username: str):
    """Create a TOTP secret and generate a QR code for the user."""
    secret = pyotp.random_base32()
    totp = pyotp.TOTP(secret)

    uri = totp.provisioning_uri(
        name=username,
        issuer_name='Secure-Authenticar'
    )

    img = qrcode.make(uri)
    img.save(f'{username}_totp.png')

    return secret


# ---------------------------------------------------------
# Complete Authentication System
# ---------------------------------------------------------
class AuthenticationSystem:

    def __init__(self):
        # mock storage for users
        self.users = {}

    # Register new user
    def register_user(self, username: str, password: str):
        print("\n--- Registering User ---")

        if username in self.users:
            print("User already exists.")
            return False

        score, entropy = strength_checker(password)

        # Reject weak password
        if score < 3:
            print("Weak password. Registration rejected.")
            print(f"Score: {score}, Entropy: {entropy} bits")
            return False

        # Hash password
        pwd_hash = hash_password(password)

        # Setup TOTP
        totp_secret = setup_totp(username)

        # Save user record
        self.users[username] = {
```

```
            "hash": pwd_hash,
            "totp_secret": totp_secret
        }

        print("User registered successfully.")
        print(f"TOTP QR generated: {username}_totp.png")

        return True

    # Authenticate user
    def authenticate(self, username: str, password: str):
        print("\n--- Authenticating User ---")

        if username not in self.users:
            print("User does not exist.")
            return False

        user_record = self.users[username]
        stored_hash = user_record["hash"]
        totp_secret = user_record["totp_secret"]

        # Step 1: Verify password
        if not verify_password(password, stored_hash):
            print("Incorrect password.")
            return False

        # Step 2: Verify TOTP code
        totp = pyotp.TOTP(totp_secret)
        otp_input = input("Enter the TOTP code: ").strip()

        if not totp.verify(otp_input):
            print("Invalid TOTP code.")
            return False

        print("Authentication successful.")
        return True
```

Running The Authentication System

```
# Step 1 — Create system
auth = AuthenticationSystem()

# Step 2 — Register user
auth.register_user("hammad", "MyP@ssw0rd123")

# Step 3 — Authenticate user
auth.authenticate("hammad", "MyP@ssw0rd123")
```

```
--- Registering User ---
User registered successfully.
TOTP QR generated: hammad_totp.png

--- Authenticating User ---
Enter the TOTP code: 279653
Authentication successful.
True
```

Summary

- **strength_checker()** - Reponsible for password strength analysis
- **hash_password()** - Generates a bycrypt-based hash, using a system-wide pepper
- **setup_totp()** - Creates a Base32 secret and corresponding QR code so user can use it with an authenticator app.
- **register_user()** - Rejects weak passwords, and accepts strong password - hashes it, creates a TOTP secret and stores the resulting credentials.
- **authentication()** - Verifies the user's password against the stored bycrypt hash and validates the TOTP code before it grants access.

Final Reflection

The integrated system which we have built, demonstrates that secure authentication depends on combining metrics like strong password evaluation, using slow and adaptive hashing fuction which are harder to crack, additional protection through peppering, and

using TOTP-based multifactor authentication will lead to practical understanding of these layers and how collectively they reduce the common attack risks includeing brute force, weak credentials guessing and unauthorised access.