Networks and Systems Security
Week 03
Authentication and Access Control

Aims of the Seminar

Welcome to the Authentication and Access Control workshop. In this workshop, we'll dive into the essential principles of modern authentication and access control. We'll explore why simple password checks aren't enough and build a robust system from the ground up using Python.

Workshop Outline:

- Password Strength Analysis
- Password Hashing Methods
- Salt and Pepper Implementation
- TOTP (2FA) Implementation
- Brute Force Attack Simulation
- Complete Authentication System

Feel free to discuss your work with peers, or with any member of the teaching staff.

Reminder

We encourage you to discuss the content of the workshop with the delivery team and any findings you gather from the session.

Workshops are not isolated, if you have questions from previous weeks, or lecture content, please come and talk to us.

Exercises herein represent an example of what to do; feel free to expand upon this.

Helpful Resources

Password strength

https://en.wikipedia.org/wiki/Password_strength

bcrypt is a password-hashing function

https://en.wikipedia.org/wiki/Bcrypt

PyOTP Python library

https://pypi.org/project/pyotp/

Setting up

Install the python libraries library

```
pip install bcrypt pyotp qrcode pillow
```

Section 1: Password Strength Analysis

Objective

To understand the fundamental criteria for a strong password and how to programmatically analyse it.

Key Concepts

- **Character Sets**: The types of characters used (lowercase, uppercase, numbers, symbols). Using a wider variety of character sets dramatically increases entropy.
- **Length**: The single most important factor in password strength. An 8-character password is weak, 12 is good, and 16+ is excellent.

Write a Code to analyse password strength

Write a function that acts as a simple "password meter." It uses a scoring system based on common best practices.

- **Length Check**: It awards points for meeting 8- and 12-character minimums.
- **Character Variety**: It checks for the presence of uppercase letters, lowercase letters, numbers, and special symbols (string.punctuation), awarding a point for each.
- **Common Password Check**: It checks against a tiny list of notoriously bad passwords. In a real system, you'd use a list of breached passwords (like the "Have I Been Pwned" list).
- **Entropy**: A measure of randomness or unpredictability. In passwords, higher entropy means it's harder to guess. You can use this approximation to calculate it:

```
Approximate entropy: length * log2(pool_size)
```

Where the pool size is the number of options that a password can be for example if only the lower-case alphabets, then the pool size would be 26

Discussion Points

- Run the code and test various passwords.
- Look at the feedback for "Pass123" vs. "MyP@ssw0rd". Why is the second one significantly better?

- What are the limitations of this checker? (e.g., It doesn't detect dictionary words like "CorrectHorseBatteryStaple" or keyboard patterns like "asdfghjkl").

🌓 Section 2: Don't Store Passwords, Store Hashes!

Objective

To understand the critical concept of password hashing and to compare insecure hashing algorithms (MD5, SHA-256) with a secure one (bcrypt).

Key Concepts

- Hashing: A one-way cryptographic function that transforms an input (like a password) into **a fixed-size** string of characters called a hash. It's designed to be irreversible.
- MD5 & SHA-256: Fast hashing algorithms. Their speed is great for checking file integrity but terrible for passwords, as it allows attackers to guess billions of passwords per second.
- bcrypt: A password hashing function that is intentionally slow and adaptive. You can increase the "work factor" (or rounds) over time as computers get faster, keeping your hashes secure.

💻 Write a code to test the Hashing Functions

- hash with md5 and sha256: These are straightforward. They take the password, encode it to bytes, and return the hexadecimal digest.
- Hash with bcrypt: This is the recommended approach.
- Generates a random salt (we'll cover this next!).
- Combines the password and salt and performs the slow hashing process.
- You don't "unhash" the stored value. Instead, you take the user's login attempt, hash it using the *same salt* stored in the original hash, and see if the results match.

🗣️ Discussion Points

- how the MD5 and SHA-256 hashes are just short strings.
- The bcrypt hash looks different: $2b$12$.... It contains the algorithm version, the work factor (12), the salt, and the hash all in one string.

Question: Why is a *fast* algorithm bad for storing passwords? (It makes brute-force attacks much faster for an attacker who steals the hash database).

🧪 Section 3: Adding Salt and Pepper

Objective

To understand how salting defeats pre-computation attacks (like rainbow tables) and how a pepper adds another layer of security.

Key Concepts

- **Rainbow Table**: A precomputed table of hashes for common passwords. If you don't use a salt, an attacker can find a matching hash in their table and instantly know the password.
- **Salt**: A unique, random string that is added to a password before hashing. Every user gets a different salt. It is stored in the database alongside the hashed password.
- **Pepper**: A secret, static string that is added to a password before hashing. Unlike a salt, the pepper is the same for all users and is NOT stored in the database. It's stored securely in the application's configuration or an environment variable.

💻 Write a code to demo salt and pepper

1. **Without Salt**: show that hashing the same password ("user123password") twice produces the exact same hash. This is what makes it vulnerable to a rainbow table attack.
2. **With Salt**: then hash with salt twice. It creates a new random salt each time, the final hashes are completely different, even though the password was the same. This defeats rainbow tables.
3. **With Pepper**: The final step add a pepper. An attacker who steals the database would get the user salts and hashes, but they would still be missing the secret pepper, making the hashes much harder to crack.

Note: Modern functions like bcrypt handle salting for you automatically, which is why it's the preferred method. This section demonstrates the underlying principle.

🗣️ Discussion Points

- If an attacker steals your database, what can they do if...

- You only used SHA-256 with no salt? (Look up all hashes in a rainbow table).
- You used SHA-256 with a unique salt for each user? (They can't use a rainbow table, but they still have to attack each password individually. This is much slower!).
- You used SHA-256 with salt AND a pepper? (They can't begin cracking passwords until they also find and steal the secret pepper).

📱 Section 4: Implementing Two-Factor Authentication (2FA)

Objective

To add a second layer of security using Time-based One-Time Passwords (TOTP).

Key Concepts

- **2FA (Two-Factor Authentication):** A security process where users provide two different authentication factors. This is typically "something you know" (password) and "something you have" (your phone).
- **TOTP (Time-based One-Time Password):** An algorithm that uses a shared secret key and the current time to generate a temporary, single-use code.
- **Provisioning URI**: A special link (often shown as a QR code) that contains the secret key, account name, and issuer information needed for an authenticator app (like Google Authenticator or Authy) to start generating codes.

💻 Write a Code and experiment using the TOTPAuthenticator

Implement the One-Time Password using the pyotp TOTP library. You can find it on: https://pyauth.github.io/pyotp/

🗣️ Discussion Points

- Run the demo. It will generate a totp_qr.png file. Open it and scan it with an authenticator app on your phone.
- Watch the codes the script prints and compare them to the codes on your phone. They should match!

Question: Why is TOTP considered more secure than 2FA codes sent via SMS? (SMS messages can be intercepted, and phone numbers can be stolen via "SIM swapping" attacks).

(Challenge) Section 5: Simulating a Brute-Force Attack

Objective

To visually demonstrate why fast hashes are weak and why slow, salted hashes are strong.

Key Concepts

Brute-Force Attack: An attack where a threat actor tries every possible password combination until they find the correct one.

Dictionary Attack: A more efficient form of brute-force where the attacker uses a list of common words and passwords (a "dictionary" or "wordlist").

Write a code to simulate brute force

1. Mimics a dictionary attack.
2. That takes a target_hash and a hash_type.
3. It iterates through a small list of common_passwords.
4. In each iteration, it hashes the common password using the specified algorithm (MD5 or SHA-256).
5. It compares the result to the target_hash.
6. If it finds a match, it reports success, showing how quickly it was found.

Discussion Points

- Observe how quickly the MD5 and SHA-256 hashes for "password" are cracked (likely in milliseconds).
- The demo doesn't even *try* to crack the bcrypt hash. Why? Because bcrypt is designed to be slow. A single guess might take 100 milliseconds. Trying the tiny list would be noticeably slower, and trying a real password list of millions would take days, weeks, or years, making the attack impractical.

This is the entire point of using bcrypt.

🧩 Section 6: Building the Complete System

Objective

To integrate all the concepts, we've discussed into a single, secure authentication class.

💻 Write a code for a system that combine all the covered ideas

Simulate a real-world user management system with:

1. Registration (register_user):
   - It first analyse password strength to reject weak passwords.
   - It then securely hash the password (which automatically generates and includes a salt).
   - It creates a TOTP Authenticator instance to generate a secret key for the new user.
   - Finally, it stores the password hash and totp secret in its users dictionary (our mock database).
2. Authentication (authenticate):
   - It finds the user in the database.
   - It verifies the password. This is secure because the salt is extracted from the stored hash and used in the comparison.

☑ Key Takeaways & Best Practices

This workshop covered the foundational pillars of modern, secure authentication.
Always remember:

- **Enforce Strong Passwords:** Use a password strength meter and reject weak or common passwords.
- **NEVER Store Plaintext Passwords:** This is the #1 rule.
- **Use a Slow, Adaptive Hash Function:** Use **bcrypt** or a modern alternative like **Argon2**. Avoid MD5 and SHA for passwords.
- **Always Salt Passwords:** Modern libraries like bcrypt do this automatically. It protects you from rainbow table attacks.
- **Implement 2FA:** Offer TOTP as a standard security feature. It protects users even if their password is stolen.
- **Consider a Pepper:** For an extra layer of defence, a system-wide pepper can protect a compromised database.

The end 😊