Introduction

Passwords are the first line of defense for user accounts, yet many systems still store them in ways that are vulnerable to attacks. In this tutorial, we will explore different methods of handling passwords, from storing them in plain text (which should never be done) to using modern techniques like salting, slow hashing, and even moving toward passwordless authentication.

The goal is to understand why certain methods are weak and how attackers exploit them. By the end of this tutorial, you'll have a solid grasp of why secure password storage is essential and how to implement it effectively.

Tutorial: Understanding Password Hashing for Beginners

1. Storing Passwords in Plain Text (The Wrong Way)

When users create accounts on websites, they enter passwords to secure their information. Storing these passwords in plain text (e.g., password123) is a terrible practice because if a hacker gains access to the database, they can see every user's password immediately.

Pros:

- Easy to implement.
- No processing power required.

Cons:

- Extremely insecure—if the database is leaked, all passwords are exposed.
- Users often reuse passwords, leading to further security risks.

Why move to encryption? Encryption can protect passwords by making them unreadable without a key.

username | password

---

user1 | password123

user2 | mysecurepassword

This method is insecure and should never be used.

2. Encrypting Passwords (A Step Forward, but Not Ideal)

Encryption converts the password into a coded format using a key. The main issue with encryption is that if someone obtains the encryption key, they can decrypt all passwords.

Pros:

- Passwords are not stored in plain text, adding a layer of security.
- Encrypted data can be reversed if needed.

Cons:

- If the encryption key is compromised, all passwords can be decrypted.
- Still vulnerable to insider threats.

Why move to hashing? Hashing is a one-way process, making it impossible to reverse-engineer passwords.

Example of encryption:

Original password: password123

Encrypted: B24sdK93jd82Jd==

---

3. Hashing Passwords (The Secure Way)

Hashing is a one-way process that converts a password into a fixed-length string. Unlike encryption, hashing cannot be reversed. Common hashing algorithms include MD5, SHA-1, and SHA-256.

Pros:

- No encryption key needed, making it safer from insider threats.
- One-way process prevents easy recovery of the original password.

Cons:

- Simple hashing is vulnerable to brute force and rainbow table attacks.
- If two users have the same password, they will have the same hash (no uniqueness).

Why move to salting? Salting ensures that even identical passwords have unique hashes, making attacks harder.

Example of hashing:

Original password: password123

Hashed: ef92b778bafe771e89245b89ecbc6c6a6e21a2e57862cf66b60c57419a0c9d75 (SHA-256)

Moving Away from MD5

MD5 is no longer considered secure due to **password collisions**, where two different passwords can produce the same hash. This makes it easier for attackers to crack passwords using precomputed hash tables. More secure alternatives like **SHA-256, bcrypt, and Argon2** should be used instead.

Real-world example: **The LinkedIn 2012 data breach** exposed millions of user passwords hashed with SHA-1, which were cracked within hours due to vulnerabilities in the algorithm.

---

4. Adding Salt (Making Hashing More Secure)

Salting means adding a unique, random value to each password before hashing it. This prevents attackers from using precomputed hash tables.

Pros:

- Prevents rainbow table attacks.
- Ensures different users with the same password have different hashes.

Cons:

- If the salt is not stored securely, attackers can still crack hashes.
- Does not prevent brute force attacks.

Why move to slow hashing? Slow hashing functions make brute-force attacks impractical by increasing computational cost.

Example of hashing with salt:

Salt: Xy7!@e

Password: password123

Hashed: 7d2b3e8e6d08a58a6537b3f9d0d82acff6e5a36a499fd76c94b9c69b2344fbd1

---

5. Using Slow Hash Functions (Preventing Brute Force Attacks)

Fast hash functions like MD5 and SHA-1 are vulnerable to brute-force attacks. Instead, slow hash functions such as **bcrypt, scrypt, and Argon2** should be used. These are designed to take more time and computational power to generate hashes, making them more resistant to attacks.

Pros:

- Computationally expensive, making brute force attacks impractical.
- Adjustable difficulty (work factor) ensures security can be improved over time.

Cons:

- More CPU-intensive than standard hashing, making it slower.

Understanding Bcrypt

- The **first two characters ($2)** indicate that the hash was generated using bcrypt.
- The **next character (a)** represents the bcrypt version.
- The **work factor (12)** determines how many rounds of hashing are applied. A higher work factor increases security but also slows down hashing, making brute-force attacks more difficult.

Why Work Factor Matters

Increasing the work factor exponentially increases the time needed to compute a hash. For example:

- **Work factor 10** → Hashing takes 100ms
- **Work factor 12** → Hashing takes 400ms

This delay makes brute-force attacks significantly harder.

---

7. Passwordless Authentication (OAuth 2 and Beyond)

OAuth 2.0 is an authorization protocol, not an authentication protocol. For authentication, **OpenID Connect** (OIDC) is built on top of OAuth 2.0, allowing users to log in via trusted providers like Google or Facebook.

Pros:

- No passwords to steal or crack.
- More user-friendly experience.
- Eliminates risks of phishing attacks targeting passwords.

Cons:

- Users must trust third-party providers (Google, Facebook, etc.).
- Requires internet access.

Conclusion

Security is about making an attacker's job as difficult as possible. No system is 100% unbreakable, and with enough determination, a data breach can still occur. The goal is to implement security measures that make an attack so time-consuming and expensive that hackers give up.

Best Practices:

- Use **slow hashing algorithms** like **bcrypt** or **Argon2** combined with **salting**.
- Implement **multi-factor authentication (MFA)** for added security.
- Enforce strong password policies.
- Consider **passwordless authentication** with OAuth 2.0 and OpenID Connect.

By following these best practices, we can significantly reduce security risks and create a safer online environment.