Problem: The Digital Fortress
Securing User Passwords

BlockChain Web3
Cyber Security Club

August 7, 2025

Objective

Understand why storing passwords in **plaintext** is a critical security flaw and learn to implement modern, secure password storage using **hashing** and **salting** with the `bcrypt` library in Python.

The Intelligence Briefing

Threat: The Plaintext Peril

Imagine you're building a new app. If you store user passwords as they are (e.g., `"P@ssw0rd123!"`) in your database, you're sitting on a time bomb. If a hacker breaches your database, they instantly have the keys to every user's account. Worse, since people often reuse passwords, the hacker can use these stolen credentials to attack user accounts on other websites.

Countermeasure 1: Hashing - The One-Way Blender

To prevent this, we use a **hash function**. Think of it like a magical, irreversible blender.

- You put your password (the ”ingredient”) into the blender (the hash function).
- It produces a unique, fixed-length smoothie (the **hash**).
- **One-Way:** You can't turn the smoothie back into the original ingredient. Similarly, you can't reverse a hash to get the password.
- **Deterministic:** The *exact same* ingredient will always produce the *exact same* smoothie.

Countermeasure 2: Salting - The Secret Ingredient

What if two users choose the same common password, like `"password123"`? Their hashes would be identical. Hackers use pre-computed lists of common password hashes, called **rainbow tables**, to find matches instantly. To defeat this, we add a unique, random

”secret ingredient” called a **salt** to each password *before* putting it in the blender. Now, even if two users have the same password, their hashes will be completely different because their unique salts are different.

Your Toolkit: The `bcrypt` Library

Instead of building this complex logic from scratch, we use a battle-tested library designed for this exact purpose: `bcrypt`. It securely handles both hashing and salting in one go.

- `bcrypt.hashpw(password, bcrypt.gensalt())`: Takes a password, generates a fresh random salt, and returns the final secure hash.
- `bcrypt.checkpw(password, stored_hash)`: Takes a login password and a stored hash. It automatically extracts the original salt to perform a secure comparison, returning `True` or `False`.

Your Mission

Your mission is to implement two Python functions for a secure authentication system using the `bcrypt` library.

1. `hash_password(plain_text_password)`: This function will take a regular password string and convert it into a secure, salted hash ready for storage.
2. `check_password(plain_text_password, hashed_password)`: This function will verify a user’s login attempt by comparing their submitted password against the stored hash.

Implementation Guide

1. Installation

First, you need to install the `bcrypt` library. Open your terminal or command prompt and run:

```
pip install bcrypt
```

2. Encoding is Key

`bcrypt` operates on **bytes**, not regular strings. You must encode your password strings to bytes using `.encode(’utf-8’)` before passing them to any `bcrypt` function.

Function Skeletons & Example Usage

```python
import bcrypt

def hash_password(plain_text_password: str) -> bytes:
    """
    Hashes a plaintext password using bcrypt.
    """
    # Remember to encode the password to bytes
    password_bytes = plain_text_password.encode(’utf-8’)

    # Generate a salt and hash the password
    salt = bcrypt.gensalt()
    hashed_bytes = bcrypt.hashpw(password_bytes, salt)

    return hashed_bytes

def check_password(plain_text_password: str, hashed_password: bytes) -> bool:
    """
    Checks if a plaintext password matches a stored bcrypt hash.
    """
    password_bytes = plain_text_password.encode(’utf-8’)

    # bcrypt’s checkpw handles the salt extraction and comparison
    return bcrypt.checkpw(password_bytes, hashed_password)

# --- Example Usage ---
password = "mySuperSecureP@ssword!"

# 1. User registers -> Hash their password and "store" it
stored_hash = hash_password(password)
print(f"Password: {password}")
print(f"Stored Hash: {stored_hash}")

# 2. User logs in -> Check their attempt
is_correct = check_password(password, stored_hash)
print(f"Login with correct password successful? {is_correct}")

is_correct_again = check_password("WrongPassword!", stored_hash)
print(f"Login with wrong password successful? {is_correct_again}")
```