Lesson 9: Password Hashing and Basic Security in Python

Marcin Kurzyna

Lecture Goals

This lecture introduces:

- Why passwords should never be stored in plain text.
- The concept of hashing.
- Writing a simple hash function without libraries.
- Using Python libraries for secure password hashing.
- Comparing insecure and secure approaches.

1 Why Password Hashing Is Important

Storing passwords as plain text is dangerous. If a database is leaked, attackers gain immediate access to user accounts.

Correct approach:

- Store only a **hash** of the password.
- When a user logs in, hash the entered password and compare hashes.

2 What Is a Hash Function?

A hash function:

- Converts input data into a fixed-size value.
- Is deterministic (same input $\rightarrow$ same output).
- Is difficult to reverse.

3 Part A: Simple Hashing Without Libraries

Important: This approach is for **educational purposes only**. It is **NOT secure**.

Example: A Very Simple Hash Function

```
def simple_hash(password):
    hash_value = 0
    for char in password:
        hash_value += ord(char)
    return hash_value

print(simple_hash("password"))
print(simple_hash("admin"))
```

Improving the Hash Slightly

```
def simple_hash(password):
    hash_value = 0
    for char in password:
        hash_value = (hash_value * 31 + ord(char)) % 100000
    return hash_value

print(simple_hash("password"))
print(simple_hash("password1"))
```

Storing and Verifying Passwords

```
stored_hash = simple_hash("secret123")

def check_password(input_password):
    return simple_hash(input_password) == stored_hash

print(check_password("secret123"))  # True
print(check_password("wrongpass"))  # False
```

Why This Is Insecure

- Easy to reverse or brute-force.
- High probability of collisions.
- No protection against modern attacks.

4 Part B: Hashing with Python Libraries

4.1 Using `hashlib`

Python provides the `hashlib` module for cryptographic hashes.

```
import hashlib

password = "secret123"
```

```
hash_object = hashlib.sha256(password.encode())
password_hash = hash_object.hexdigest()

print(password_hash)
```

Password Verification

```
def hash_password(password):
    return hashlib.sha256(password.encode()).hexdigest()

stored_hash = hash_password("secret123")

def check_password(input_password):
    return hash_password(input_password) == stored_hash

print(check_password("secret123"))
print(check_password("wrongpass"))
```

5 Salting Passwords

A **salt** is random data added to the password before hashing.

```
import os
import hashlib

salt = os.urandom(16)

def hash_password(password, salt):
    return hashlib.sha256(salt + password.encode()).hexdigest()

stored_hash = hash_password("secret123", salt)

print(check_password("secret123"))
```

6 Using `hashlib.pbkdf2_hmac`

This method is designed specifically for passwords.

```
import hashlib
import os

password = "secret123"
salt = os.urandom(16)

hash_value = hashlib.pbkdf2_hmac(
    'sha256',
    password.encode(),
    salt,
    100000
```

```
)

print(hash_value)
```

7 Comparison of Approaches

Method	Secure	Real-world Use
Custom hash (no libs)	No	No
SHA-256 only	Medium	Limited
PBKDF2 / bcrypt	Yes	Yes

Summary

In this lecture, we covered:

- Why passwords must be hashed.
- Writing simple hash functions without libraries.
- Using `hashlib` for secure hashing.
- The importance of salts.
- Best practices for password storage.

8 Exercises

1. Write your own simple hash function using multiplication and modulo.
2. Show two different passwords that produce the same simple hash.
3. Hash a password using SHA-256 and verify it.
4. Modify the code to store both the salt and the hash.
5. Research task: Explain why `md5` is no longer considered secure.
6. Challenge: Implement a simple login system using `hashlib.pbkdf2_hmac`.