Implement Hashing Techniques for Securing Data in C#

Objective:

By the end of this lesson, students should:

1. Understand **what hashing is** and its role in security.
2. Be familiar with **common hashing algorithms** and their strengths/weaknesses.
3. Learn how to **implement hashing securely in C#**.
4. Apply **best practices for password hashing and data integrity**.

Lesson Outline

Part 1: Introduction to Hashing

Objective: Define hashing and explain its importance in security.

1.1 What is Hashing?

- A **one-way transformation** that converts input data into a fixed-size output (hash).
- Hashes **cannot be reversed** (unlike encryption).

1.2 Properties of a Good Hashing Algorithm

- **Deterministic:** Same input → same hash.
- **Irreversible:** Cannot be converted back to the original input.
- **Fast to compute** but hard to crack.
- **Collision-resistant:** Different inputs should not produce the same hash.

1.3 Real-Life Examples of Hashing

- **Password Storage:** Websites store hashed passwords instead of plain text.
- **Digital Signatures:** Ensuring document authenticity.
- **File Integrity Verification:** Verifying downloads (e.g., MD5/SHA-256 checksums).
- **Blockchain Transactions:** Bitcoin uses SHA-256 for mining.

1.4 Demonstration: Hashing a Simple String in C#

```
using System;
using System.Security.Cryptography;
using System.Text;

class Program
{
    static void Main()
    {
        string input = "HelloWorld";
        string hash = ComputeSHA256(input);

        Console.WriteLine($"Input: {input}");
        Console.WriteLine($"SHA-256 Hash: {hash}");
    }

    static string ComputeSHA256(string rawData)
    {
        using (SHA256 sha256 = SHA256.Create())
        {
            byte[] bytes = sha256.ComputeHash(Encoding.UTF8.GetBytes(rawData));
            return Convert.ToBase64String(bytes);
        }
    }
}
```

Part 2: Common Hashing Algorithms & Their Use Cases

Objective: Compare different hashing techniques and their security implications.

Algorithm	Strengths	Weaknesses	Use Cases
MD5	Fast, widely available	Weak (collisions, not secure)	File integrity checks (not for security)
SHA-1	More secure than MD5	Broken (vulnerable to attacks)	Legacy applications
SHA-256	Strong, widely used	Vulnerable to brute force without salt	Password hashing (with salt), Blockchain
HMAC-SHA256	Adds secret key for security	Requires key management	API authentication, data integrity
PBKDF2	Slows down brute-force attacks	Requires iterations tuning	Password hashing
bcrypt	Secure, adaptive (cost can increase)	Slightly slower	Password hashing (modern web apps)
scrypt	Memory-hard (resists GPU attacks)	More CPU-intensive	Cryptocurrency, key derivation
Argon2	Most secure, resistant to modern attacks	Newer, not as widely supported	Best for password hashing

Part 3: Implementing Secure Hashing in C#

Objective: Show how to implement and compare different hashing techniques in C# with practical examples.

3.1 SHA-256 with Salt (Basic Security)

```
using System;
using System.Security.Cryptography;
using System.Text;

class Program
{
    static void Main()
    {
        string password = "MySecurePassword";
        string salt = GenerateSalt();
        string hash = ComputeSHA256(password + salt);

        Console.WriteLine($"Password: {password}");
        Console.WriteLine($"Salt: {salt}");
        Console.WriteLine($"SHA-256 Hash: {hash}");
    }

    static string GenerateSalt()
    {
        byte[] saltBytes = new byte[16];
        new RNGCryptoServiceProvider().GetBytes(saltBytes);
        return Convert.ToBase64String(saltBytes);
    }

    static string ComputeSHA256(string input)
    {
        using (SHA256 sha256 = SHA256.Create())
        {
            byte[] bytes = sha256.ComputeHash(Encoding.UTF8.GetBytes(input));
            return Convert.ToBase64String(bytes);
        }
    }
}
```

3.2 Using HMAC-SHA256 (Better for Data Integrity)

```
static string ComputeHMACSHA256(string data, string key)
{
    using (var hmac = new HMACSHA256(Encoding.UTF8.GetBytes(key)))
    {
        byte[] hashBytes = hmac.ComputeHash(Encoding.UTF8.GetBytes(data));
        return Convert.ToBase64String(hashBytes);
    }
}
```

3.3 PBKDF2 (Password Hashing with Iterations)

```
using System;
using System.Security.Cryptography;

class Program
{
    static void Main()
    {
        string password = "MySecurePassword";
        byte[] salt = new byte[16];
        new RNGCryptoServiceProvider().GetBytes(salt);

        var hashedPassword = HashPasswordPBKDF2(password, salt);
        Console.WriteLine($"Hashed Password: {Convert.ToBase64String(hashedPassword)}");
    }

    static byte[] HashPasswordPBKDF2(string password, byte[] salt)
    {
        return new Rfc2898DeriveBytes(password, salt, 10000).GetBytes(32);
    }
}
```

3.4 bcrypt (More Secure)

```
using BCrypt.Net;

class Program
{
    static void Main()
    {
        string password = "MySecurePassword";
        string hashedPassword = BCrypt.HashPassword(password);

        Console.WriteLine($"Hashed Password: {hashedPassword}");
        Console.WriteLine($"Password Match: {BCrypt.Verify(password, hashedPassword)}");
    }
}
```

3.5 Argon2 (Best for Modern Applications)

```
using System;
using Konscious.Security.Cryptography;
using System.Text;

class Program
{
    static void Main()
    {
        string password = "MySecurePassword";
        byte[] salt = Encoding.UTF8.GetBytes("SomeRandomSalt");

        using (var argon2 = new Argon2id(Encoding.UTF8.GetBytes(password)))
        {
            argon2.Salt = salt;
            argon2.DegreeOfParallelism = 2;
            argon2.MemorySize = 65536;
            argon2.Iterations = 4;

            byte[] hash = argon2.GetBytes(32);
            Console.WriteLine($"Argon2 Hash: {Convert.ToBase64String(hash)}");
        }
    }
}
```

Part 4: Best Practices for Secure Hashing

1. **Always use a salt** for password hashing.
2. **Increase iterations** in PBKDF2, bcrypt, and Argon2 to slow down attacks.
3. **Use bcrypt or Argon2 for password storage** (not SHA-256).
4. **Use HMAC for data integrity checks** when secret keys are needed.
5. **Never store passwords in plaintext.**

Part 5: Hands-on Activity

Task:

- Implement a user registration system that securely hashes passwords using **bcrypt** or **Argon2** and verifies them during login.

Exercises for Hashing Algorithms in C#

Objective:

These exercises will help students reinforce their understanding of hashing algorithms by implementing them in C#. They include hands-on tasks ranging from basic hashing to securing user passwords.

📝 Exercise 1: Understanding Basic Hashing

Task:

Write a C# console application that:

1. Prompts the user to enter a message.
2. Computes the SHA-256 hash of the message.
3. Displays the original message and the hashed output.

Expected Output Example:

Enter a message: HelloWorld
SHA-256 Hash: xQhR7Us4wr1S4v+2EN8eVR9zQ7UpLgPjCxtbb9h+ph4=

📝 Exercise 2: Implementing SHA-256 with Salt

Task:

Modify the previous exercise to:

1. Generate a **random salt** (16 bytes).
2. Append the salt to the message before hashing.
3. Display the salt and the final hash.

Expected Output Example:

Enter a password: MySecurePassword
Generated Salt: 3Gv5JvK9rEklj1a6sY3f==
Salted Hash: x+PpJzG3kZkKVo9+0EMjKTxT0m9X1JgBjG6n43d08CY=

📝 Exercise 3: Verify a Hashed Password

Task:

1. Ask the user to **register** by entering a password.
2. Hash the password using SHA-256 with salt and store the hash.
3. Ask the user to **log in** and enter their password.
4. Hash the entered password with the same salt and **compare** it to the stored hash.
5. Print “Login Successful” if the hash matches; otherwise, print “Invalid Password.”

📝 Exercise 4: Implementing PBKDF2 for Password Hashing

Task:

1. Use **PBKDF2** to hash a password with at least **10,000 iterations**.
2. Store the hash and salt.
3. Allow the user to verify their password against the stored hash.

💡 *Hint: Use `Rfc2898DeriveBytes` in C#.*

📝 Exercise 5: Password Hashing with bcrypt

Task:

1. Use the `BCrypt.Net` package to hash a password.
2. Allow users to enter their password and verify it against the stored bcrypt hash.

Expected Output Example:

Enter a password: SecurePass123
Hashed Password: $2a$11$3mBbyVbSxRNE9U...
Enter password to verify: SecurePass123
Password Match: True

📝 Exercise 6: Implement HMAC for Data Integrity

Task:

1. Implement **HMAC-SHA256** to hash a message using a secret key.
2. Simulate sending a message over an insecure channel.
3. The recipient should verify the message's authenticity using the HMAC.

💡 *Hint: Use `HMACSHA256` in C#.*

📝 Exercise 7: Implement Argon2 for Secure Password Storage

Task:

1. Use **Argon2id** to hash a password.
2. Store the salt and hash securely.
3. Verify user login by rehashing the entered password and comparing it to the stored hash.

💡 *Hint: Use `Konscious.Security.Cryptography.Argon2id`.*

📝 Bonus Exercise: Build a Simple Authentication System

Task:

1. Implement a basic **user registration and login system** using bcrypt or Argon2.
2. Store usernames and hashed passwords in a dictionary or file.
3. Allow users to **register, log in, and verify passwords securely.**

📝 Discussion Questions

1. Why is it important to use **salt** when hashing passwords?
2. How does **PBKDF2 improve security** over SHA-256?
3. What makes **Argon2 better** than bcrypt for password storage?
4. When should you use **HMAC instead of a normal hash function**?
5. How do **attackers try to crack hashed passwords**, and how do secure hashing methods prevent this?