Chapter 17

More skills for securing a website

Objectives

Applied

1. Develop code that hashes and salts passwords.
2. Develop code that enforces password strength requirements.

Knowledge

1. Name and describe three common types of website attacks.
2. Describe how to prevent social engineering attacks.
3. Distinguish between a one-way hash and reversible encryption.
4. Distinguish between a dictionary attack and a rainbow table attack.
5. Describe the characteristics of a weak password.
6. Describe why it's a good practice to hash and salt a password before storing it in a database.

Common website security attacks

- *Cross-site scripting* (*XSS*) attacks allow an attacker to inject Javascript into your page
- *SQL injection attacks* allow an attacker to run malicious SQL code on your database.
- In a *social engineering attack*, an attacker tricks someone into revealing their username and password.
- Out of date or unpatched software running on your server can allow an attacker to exploit security vulnerabilities to gain unauthorized access to your data.

How to prevent social engineering attacks

- Make sure employees, customers, and users are aware that no one from your website will ever contact them and ask for their password.
- Be wary of unknown individuals talking to your employees who claim to be conducting surveys, research, and so on.
- Make sure your employees, customers, and users are aware of the danger of using passwords made of information attackers can easily discover about them.
- When seeking help with technical issues online, such as programming questions, don’t reveal any information that would be useful to an attacker.

Common cryptographic algorithms

Algorithm	Description
md5	Older 128-bit one-way hash algorithm. Known to be vulnerable to collisions and should not be used.
SHA-1	160-bit one-way hash algorithm. Theoretically vulnerable to collisions, but none have been found.
SHA-2	SHA-2 supports hash sizes from 224 to 512 bits. 256-bit key is considered uncrackable.
AES-128	Reversible encryption standard with 128-bit key. Considered suitable for most encryption needs.
AES-256	256-bit version of AES. Generally the minimum level of encryption required by the U.S. government for top secret data.

Introduction to cryptography

- A *one-way hash algorithm* takes a given input string, and hashes it to a string of a certain length.
  - Cannot be reversed.
  - Useful for encrypting things such as passwords, where there is no need for anyone to read the original value.
- *Reversible encryption* encrypts data against a key.
  - The key can be used to decrypt the data later when it needs to be read by a user.

Common password attacks

Method	Description
Social engineering	Attacker tricks a user into revealing his or her login credentials.
Dictionary attacks	Attacker tries different passwords until they find one that works. Typically, done using an automated program and an electronic dictionary.
Rainbow table attacks	Similar to a dictionary attack, but a pre-computed lookup table is used that contains the hashes. An attacker who has access to the hashed passwords can crack them much more efficiently and quickly.

A password is weak if it…

- Is made from information that can be easily discovered about the user.
- Is made only from words that are in the dictionary.
- Is too short.
- Is made of all lowercase letters.
- Doesn't include numbers or special characters.

Common problems with passwords

- *Weak passwords* are passwords an attacker can easily guess or crack.
- *Clear-text passwords*, or *unhashed passwords*, aren't encrypted. As a result, if an attacker gains access to your database, these passwords are easy to read.

A utility class for hashing passwords

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class PasswordUtil {
    /*  This code uses SHA-256. If this algorithm isn't available to
        you, you can try a weaker level of encryption
        such as SHA-128.
    */
    public static String hashPassword(String password)
            throws NoSuchAlgorithmException {
        MessageDigest md = MessageDigest.getInstance("SHA-256");
        md.update(password.getBytes());
        byte[] mdArray = md.digest();
        StringBuilder sb = new StringBuilder(mdArray.length * 2);
        for (byte b : mdArray) {
            int v = b & 0xff;
            if (v < 16) {
                sb.append('0');
            }
            sb.append(Integer.toHexString(v));
        }
        return sb.toString();
    }
}
```

Code that uses this class

```
try {
    String hashedPassword = PasswordUtil.hashPassword("sesame");
} catch (NoSuchAlgorithmException e) {
    System.out.println(e);
}
```

How to hash passwords

- Use the MessageDigest class to compute a fixed-length hash value for an array of bytes.
- Use the getBytes method of the String class to convert a string such as a password to an array of bytes.
- Call the update method of the MessageDigest class to specify the array of bytes that you want to hash.
- Call the digest method of the MessageDigest class to hash the input and return a fixed-length array of bytes for the hashed input.
- Code a for loop to convert the array of bytes (which are 8 bits) to a string of characters (which are 16 bits in Java).
- Some versions of Java may not have certain hash algorithms available. If you attempt to use an algorithm that isn't available, the MessageDigest object throws a NoSuchAlgorithmException.
- A *collision* occurs when two input strings hash to the same value.

The classes used to salt a password

```
java.security.SecureRandom;
java.util.Random;
java.util.Base64;
```

A method for producing a salt value

```
public static String getSalt() {
    Random r = new SecureRandom();
    byte[] saltBytes = new byte[32];
    r.nextBytes(saltBytes);
    return Base64.getEncoder().encodeToString(saltBytes);
}
```

A method for combining the password and salt

```
public static String hashAndSaltPassword(String password)
        throws NoSuchAlgorithmException {
    String salt = getSalt();
    return hashPassword(password + salt);
}
```

A User table with salted passwords

User
UserId
HashedAndSaltedPassword
Salt

How to salt passwords

- A *salt* is a random string appended to a password. To salt a password, append the salt value to the original password before hashing it.
- A salt prevents rainbow table attacks.
- The salt value is only created the first time the password is created. It must be stored in the database so it can be used later to regenerate the hash.
- SecureRandom class is a subclass of the Random class that generates random numbers suitable for cryptography purposes.
- Use the nextBytes method of the SecureRandom class to populate an array with a series of random bytes.

A class that uses Java 8 classes to hash and salt passwords

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;
import java.util.Random;
import java.util.Base64;

public class PasswordUtil {
    public static String hashPassword(String password)
            throws NoSuchAlgorithmException {
        MessageDigest md = MessageDigest.getInstance("SHA-256");
        md.reset();
        md.update(password.getBytes());
        byte[] mdArray = md.digest();
        StringBuilder sb = new StringBuilder(mdArray.length * 2);
        for (byte b : mdArray) {
            int v = b & 0xff;
            if (v < 16) {
                sb.append('0');
            }
```

A class that uses Java 8 classes to hash and salt passwords (continued)

```
            sb.append(Integer.toHexString(v));
        }
        return sb.toString();
    }

    public static String getSalt() {
        Random r = new SecureRandom();
        byte[] saltBytes = new byte[32];
        r.nextBytes(saltBytes);
        return Base64.getEncoder().encodeToString(saltBytes);
    }

    public static String hashAndSaltPassword(String password)
            throws NoSuchAlgorithmException {
        String salt = getSalt();
        return hashPassword(password + salt);
    }
```

A class that uses Java 8 classes to hash and salt passwords (continued)

```
    /*  This code tests the functionality of this class.
    */
    public static void main(String[] args) {
        try {
            System.out.println("Hash for 'sesame'       : "
                    + hashPassword("sesame"));
            System.out.println("Random salt             : "
                    + getSalt());
            System.out.println("Salted hash for 'sesame': "
                    + hashAndSaltPassword("sesame"));
        } catch (NoSuchAlgorithmException ex) {
            System.out.println(ex);
        }
    }
}
```

A method for enforcing password strength

```
public static void checkPasswordStrength(String password)
throws Exception {
    if (password == null || password.trim().isEmpty()) {
        throw new Exception("Password cannot be empty.");
    } else if (password.length() < 8) {
        throw new Exception("Password is too short. "
            + "Must be at least 8 characters long.");
    }
}
```

Code that uses this method

```
try {
    checkPasswordStrength("sesame");
    System.out.println("Password is valid.");
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

A method for enforcing password strength requirements

- Start by checking to verify that the password isn't equal to a null value.
- Use the trim and length methods to verify that the password isn't empty.
- Use the length method to verify that the password is a minimum number of characters.
- If the password doesn't meet minimum requirements, throw an exception that contains a message that describes why the password didn't meet the requirements.
- Use Java's regular expression API to enforce requirements such as minimum length, mandatory special characters, and a mandatory mix of upper and lowercase characters.