Password Encoding and Security Best Practices

Introduction

We'll explore the fundamental methods used to store passwords within systems securely. Topics include hashing techniques (e.g., bcrypt), key derivation functions (e.g., PBKDF2), and adaptive hashing methods, all essential for safeguarding sensitive user credentials.

We'll dissect CSRF attacks, understanding how malicious actors exploit authenticated sessions to manipulate users into unintended actions on separate websites. You'll discover preventive measures like CSRF tokens and same-site cookies to fortify against these attacks.

We'll uncover the workings of XSS attacks, where malicious scripts are injected to compromise web pages, potentially leading to data theft or unauthorised actions. Learn mitigation techniques such as input validation and deploying security measures like Content Security Policy (CSP) to counter these vulnerabilities.

Password Encoding Algorithms

Password encoding algorithms are fundamental for securely storing passwords within a system. They convert plaintext passwords into a hashed or encrypted format to enhance security. Here are explanations of standard password encoding techniques:

1. **Hashing Algorithms**:
    a. **Purpose**: Irreversibly converts plaintext passwords into fixed-length hashes.

    b. **Properties**:
        i. **One-way function**: Reversing the hash to obtain the original password is computationally infeasible.
        ii. **Deterministic**: The duplicate input produces the same output.
        iii. **Collision resistance**: It's difficult to find two different inputs producing the same hash.

    c. **Common Hashing Algorithms**:
        i. **MD5 (Message Digest Algorithm 5)**: Deprecated due to vulnerabilities.

        ii. **SHA-1** (Secure Hash Algorithm 1): Vulnerabilities identified; deprecated for security-critical applications.

        iii. **SHA-256**, **SHA-512**: Part of the SHA-2 family, considered more secure.

iv. **bcrypt, scrypt, Argon2**: Explicitly designed for password hashing, resistant to brute-force attacks.

2. **Key Derivation Functions** (KDFs):
   a. **Purpose**: Designed to stretch hashed passwords to make brute-force attacks more time-consuming.

   b. **Properties**:
      i. **Slow hashing**: Introduces computational expense to make hashing slow, slowing down attackers.

      ii. **Salted hashes**: Incorporates a salt (random data) to enhance security.

      iii. **Example**: **PBKDF2** (Password-Based Key Derivation Function 2): Applies a hash function multiple times to protect against brute-force attacks.
3. **Adaptive Hashing Functions**:
   a. **Purpose**: Designed to adjust hashing difficulty over time or with computational resources.

   b. **Properties**:
      i. **Parameterised**: Allows tuning parameters like iteration count to increase difficulty.

      ii. **Memory-hard functions**: Introduce memory constraints, hindering brute-force attacks.
      iii. **Examples**:
         1. **bcrypt**: Incorporates a cost factor, increasing the work factor for hashing.
         2. **scrypt**: Uses memory-hard functions, requiring significant memory resources for computation.
         3. **Argon2**: Winner of the Password Hashing Competition, balancing time and memory requirements.

4. **Salting**:
   a. **Purpose**: Combats rainbow table attacks by adding random data (salt) to passwords before hashing.

   b. **Properties**:
      i. **Unique salts**: Each password gets a different salt, even if passwords are the same.
      ii. **Stored alongside hashes**: Salts are stored alongside the hashes in databases.

   c. **Usage**: Typically concatenated with passwords before hashing to produce unique hashes.

Modern best practices involve using adaptive hashing functions (like bcrypt, scrypt, or Argon2), applying to salt, and incorporating key derivation functions to enhance password security. Regularly updating hashing algorithms based on industry standards helps maintain robust security against evolving threats.

CSRF Attack

Cross-Site Request Forgery (CSRF) is an attack that deceives users into executing unintended actions on a website where they are authenticated. It's also known as a one-click attack or session riding. Here's a detailed breakdown:

1. **Attack Workflow**
    a. **Authenticated User**: The victim is authenticated on a legitimate website (e.g., an online banking site) in one browser tab.

    b. **Malicious Request**: Without their knowledge, the attacker tricks the victim into executing a malicious action by visiting another website controlled by the attacker or clicking a crafted link.
2. **How CSRF Works**:
    a. **Exploiting Session Authenticity**: The attacker crafts a request (e.g., fund transfer, change password) that the victim's session is authorised to perform on the legitimate site.

    b. **Deceptive Execution**: The victim, while logged into the legitimate site, unknowingly triggers the attacker's request (via a link, image tag, or script) on the malicious site.

3. **Example Scenario**:
    a. **Scenario**: The victim is logged into their bank account and visits a forum (controlled by the attacker) where a post contains an image tag or script.

    b. **Attack Execution**: The image tag or script includes a request (e.g., fund transfer) to the victim's bank with the victim's session cookie attached.

    c. **Outcome**: The victim's browser sends the request to the bank, and since the victim is authenticated, the bank processes the transaction, unaware that an attacker initiated it.

4. **Key Characteristics**:
    a. **Indirect Execution**: The attack isn't directly initiated by the attacker but by the victim's browser while visiting the attacker-controlled page.

    b. **Session Cookie Usage**: Relies on the victim's authenticated session, exploiting the automatic inclusion of session cookies in requests.

5. **Mitigation and Prevention**:
    a. **CSRF Tokens**: Include random tokens in forms or requests verified by the server to ensure the request originated from a legitimate source.

b. **Same-Site Cookies**: Use cookies marked as "SameSite" to restrict cross-origin requests.

c. **Double Submit Cookies**: Send a token in a cookie and as a request parameter, validating that they match on the server side.

d. **Security Headers**: Implement security headers like X-Frame-Options and Content-Security-Policy to prevent unauthorised embedding of your site in iframes or limit the execution of scripts.

6. **Impact of CSRF**:
   a. **Unauthorised Actions**: Attackers can execute actions on behalf of users without their consent.

   b. **Data Modification**: This could result in data modification, fund transfers, changing settings, or account takeover, depending on the action allowed by the targeted site.

Protecting against CSRF requires implementing preventive measures like CSRF tokens, secure session handling, and strong security practices to verify the authenticity of requests, thereby reducing the risk of successful CSRF attacks.

XSS Attack

Cross-site scripting (XSS) is a web security vulnerability that allows attackers to inject malicious scripts into web pages viewed by other users. Here's a detailed explanation of XSS attacks:

1. **Attack Workflow**:
   a. **Injection of Malicious Scripts**: Attackers inject malicious scripts (JavaScript, HTML, etc.) into web applications that get executed in the browsers of unsuspecting users.

   b. **Execution of Malicious Code**: When victims visit the compromised web page, the injected script executes in their browsers.

2. **Types of XSS**:
   a. **Stored (Persistent) XSS**: Malicious script is permanently stored on the server and executed every time a user accesses the affected page (e.g., in a forum post, or comment).

   b. **Reflected (Non-Persistent) XSS**: Malicious script is included in the URL or input fields, and when a victim clicks on a crafted link, the script executes.

3. **How XSS Works:**
   a. **Injection Points**: Attackers exploit input fields, URL parameters, or other user-controllable data where the web application fails to sanitise or validate input.

   b. **Script Execution**: The injected script executes within the victim's browser in the context of the compromised website, allowing the attacker to steal sensitive information and session cookies, redirect users, or perform other malicious actions.

4. **Example Scenario**:
   a. **Scenario**: An attacker injects a script into a comment field on a website.

   b. **Execution**: When a user views the page with the compromised comment, the injected script executes in their browser, potentially leading to cookie theft, session hijacking, or defacement of the site.

5. **Impact of XSS**:
   a. **Session Hijacking**: Attackers can steal session cookies and impersonate legitimate users.

   b. **Data Theft**: Access to sensitive data (user credentials, payment information) entered on compromised pages.

   c. **Malware Distribution**: Spread of malware by redirecting users to malicious sites.

6. **Prevention and Mitigation**:
   a. **Input Sanitization**: Validate and sanitise user input to prevent script injection.

   b. **Escape Special Characters**: To render them harmless, Encode or escape characters like <, >, &, etc.

   c. **Content Security Policy (CSP)**: Define a whitelist of trusted sources for content and scripts to restrict the execution of unauthorised scripts.

   d. **HTTPOnly Cookies**: Use the HttpOnly flag on cookies to prevent client-side script access.

7. **Security Best Practices**:
   a. **Secure Coding Practices**: Implement proper input validation, output encoding, and secure development practices.

   b. **Regular Security Audits**: Conduct security audits and vulnerability assessments to identify and patch vulnerabilities.

   c. **User Awareness**: Educate users about phishing attempts and malicious links to reduce the likelihood of XSS exploitation.

By implementing secure coding practices, regular security checks, and educating both developers and users, the risks associated with XSS attacks can be significantly reduced, protecting both users and the integrity of web applications.

RestTemplate with JWT

Using RestTemplate with JWT involves integrating JWT (JSON Web Token) for authentication when making HTTP requests to remote servers. This process includes the following:

JWT Overview: JWT is a compact, URL-safe means of representing claims to be transferred between two parties. It consists of three parts: a header, payload, and signature. Tokens are digitally signed using a secret or public/private key pair, enabling secure transmission of information between parties.

Integrating JWT with RestTemplate: Obtain JWT Token: Users typically obtain a JWT token after successful authentication. This token needs to be included in subsequent requests to access protected endpoints.

Include JWT in RestTemplate Requests:

1. **Header Configuration**: Set the JWT token in the Authorization header.
2. **Making Requests**: Use RestTemplate to perform HTTP requests with the token included.

Example Code: Hotel Application

```
@Service
public class RatingServiceCommunicator {

    private final RestTemplate restTemplate;

    @Autowired
    publicRatingServiceCommunicator(RestTemplateBuilder restTemplateBuilder) {
        this.restTemplate=restTemplateBuilder.build();
    }

    public RatingResponse getRating(String id, String jwtToken) {
        String url ="http://localhost:8081/rating/id/";
        HttpHeaders headers = new HttpHeaders();
        headers.set("Authorization", "Bearer " + jwtToken); //adding jwt token

    HttpEntity<Map<String, Long>> requestEntity = new HttpEntity<>(headers);

    ResponseEntity<RatingResponse> ratingResponse = restTemplate.exchange(
                url+id,
                HttpMethod.GET,
                requestEntity,
                RatingResponse.class);

          return ratingResponse.getBody();
      }
}
```

Conclusion

Password Encoding Algorithms: Vital for secure password storage; include hashing (e.g., bcrypt), key derivation functions (e.g., PBKDF2), and adaptive hashing methods.

CSRF Attacks: Trick authenticated users into unintentionally executing actions on a different site; mitigated by using CSRF tokens, same-site cookies, and security headers.

XSS Attacks: Inject malicious scripts into web pages to steal data or execute unauthorised actions; prevented by input validation, output encoding, and security measures like Content Security Policy (CSP).

Each plays a critical role in web security: password encoding secures credentials, while defences against CSRF and XSS prevent unauthorised actions and data theft.

Instructor Codes

- Hotel Application with Password Encoding
- Hotel Application with JWT

References

1. Using different encoding algorithms
2. Password Encoding with Spring Security
3. CSRF Attacks
4. XSS Attacks
5. Using RestTempate with basic authentication
6. REST Security with JWT