**LESSON TWO ASSIGNMENT TWO SUBMISSION**

**GACANGA ALEX MWANGI**

**C028/401599/2023**

## References

I. **OWASP (Open Web Application Security Project)**. (2021). *OWASP Top 10 - 2021*. Retrieved from: https://owasp.org/www-project-top-ten/

II. **Halfond, W. G. J., & Orso, A.** (2005). *A Classification of SQL Injection Attacks and Countermeasures*. In Proceedings of the IEEE International Symposium on Secure Software Engineering.

III. **Fischer, M. J.** (2016). *Cross-Site Scripting (XSS) Attacks: The Complete Guide*. Retrieved from: https://www.rapid7.com/fundamentals/cross-site-scripting/

IV. **OWASP (Open Web Application Security Project)**. (2020). *Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet*. Retrieved from: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Request_Forgery_Prevention_Cheat_Sheet.html

## Report on modern security threats in web programming

### a) SQL Injection (SQLi)

**Overview:**

SQL Injection occurs when an attacker manipulates a web application's database query by inserting malicious SQL code into input fields. This attack happens when an application improperly handles user input and constructs SQL queries dynamically. Malicious users exploit these vulnerabilities by injecting their own SQL code into form fields, URL parameters, or cookies, with the goal of manipulating the application's backend database. SQL Injection can lead to unauthorized data access, data loss, or even complete system compromise, depending on the extent of the vulnerability. Attackers can use SQLi to bypass authentication mechanisms, read sensitive information from the database, modify or delete records, and sometimes even escalate their privileges to execute arbitrary commands on the server.

**Real-World Case Study:**

In 2012, the Yahoo! Voices platform suffered a breach where a hacking group exploited a SQL injection vulnerability to access approximately 450,000 user credentials. The attack allowed the attackers to execute unauthorized SQL queries on the vulnerable database, leading to the exposure of usernames, email addresses, and plaintext passwords. The fact that the passwords were stored in plaintext rather than using proper hashing techniques exacerbated the impact of the breach. Following the attack, Yahoo! issued a public statement and advised users to change their passwords to mitigate the damage. This incident highlighted the critical importance of securing user data and implementing robust security measures, such as proper input sanitization and encryption.

**Mitigation Strategies:**

1. **Use Prepared Statements:**
   Prepared statements are one of the most effective ways to prevent SQL injection. These statements separate SQL code from user input by using placeholders for user-supplied values. Prepared statements ensure that user inputs are treated as data, not executable code, which eliminates the risk of malicious code execution.

For example, in PHP, using PDO (PHP Data Objects), developers can safely execute SQL queries with parameters:

php

CopyEdit

```
$stmt = $pdo->prepare("SELECT * FROM users WHERE username = :username");

$stmt->bindParam(':username', $username);

$stmt->execute();
```

2. **Input Validation:**
   Input validation is another critical strategy in preventing SQLi attacks. By validating all user inputs and ensuring they conform to expected formats, developers can prevent malicious characters from being included in SQL queries. For instance, input fields like usernames and email addresses should be validated against a regular expression that checks for valid values.

3. **Least Privilege Principle:**
   Limiting the permissions of database users is a fundamental security principle. By applying the least privilege principle, administrators can ensure that database users have only the permissions necessary for their role. For example, a user account used by a web application should have read-only access to the database unless write permissions are absolutely necessary.

4. **Regular Security Audits:**
   Regular security audits, penetration testing, and code reviews are essential for identifying and addressing vulnerabilities before they can be exploited. Conducting these checks regularly helps ensure that new vulnerabilities are detected early and that any potential issues are addressed in a timely manner.

**b) Cross-Site Scripting (XSS)**

**Overview:**
Cross-Site Scripting (XSS) is a type of injection attack in which malicious scripts are injected into web pages viewed by other users. These scripts can steal session cookies, deface websites, or redirect users to malicious websites. XSS attacks target the client-side code of a website, taking advantage of vulnerabilities in how web applications handle user-generated content. These attacks are particularly dangerous because they can be used to steal sensitive information such as login credentials and session tokens, potentially compromising user accounts.

XSS attacks can be categorized into three types:

1. **Stored XSS:** The malicious script is permanently stored on the server and executed when a user accesses the infected page.
2. **Reflected XSS:** The malicious script is reflected off the server in the response, typically via a URL parameter.
3. **DOM-based XSS:** The script is executed in the browser as a result of modifying the DOM (Document Object Model) without involving the server.

**Real-World Case Study:**
In 2010, the LizaMoon campaign used SQL injection to insert malicious scripts into thousands of websites. When users visited these websites, they were prompted to download fake antivirus software, leading to widespread scareware infections. The attack was carried out by injecting a script into vulnerable websites that redirected users to a malicious page, which then presented a fake security warning. The fake antivirus software would then trick users into downloading malware, further compromising their systems. This campaign exploited the combination of SQL injection and XSS vulnerabilities to deceive users and spread malicious software.

**Mitigation Strategies:**

1. **Content Security Policy (CSP):**
   Implementing a Content Security Policy (CSP) is one of the most effective defenses against XSS. CSP allows web developers to specify which sources of content are allowed to be loaded by the browser, which prevents malicious scripts from being executed. For example, a strict CSP might only allow scripts to be loaded from the domain where the web application is hosted.
2. **Encode Output:**
   Properly encoding user-generated content is another important measure in preventing XSS attacks. By encoding special characters, such as <, >, and &, web applications can ensure that they are treated as text rather than executable HTML or JavaScript. This prevents malicious code from being executed in the user's browser.
3. **Sanitize Inputs:**
   Sanitizing user inputs before they are rendered in a webpage is essential to prevent XSS. Frameworks and libraries like OWASP's Java HTML Sanitizer or the ESAPI (Enterprise Security API) can help automatically sanitize and escape special characters, reducing the risk of malicious code injection.
4. **Regular Security Audits:**
   As with SQL Injection, regular security audits and code reviews are crucial for identifying and fixing potential XSS vulnerabilities. Penetration testing can help identify any places where untrusted data is reflected in the page without proper sanitization or encoding.

**c) Cross-Site Request Forgery (CSRF)**

**Overview:**
Cross-Site Request Forgery (CSRF) is an attack where a user is tricked into performing unintended actions on a web application in which they are authenticated. This can happen when a user is logged into a website and then unknowingly submits a request to perform an action on the website, such as

transferring funds or changing account settings. CSRF exploits the trust that a web application has in the user's browser, causing it to perform actions on behalf of the attacker without the user's consent.

For example, an attacker might send a link or craft a malicious form that automatically submits a request to a financial website to transfer funds, all while the user is logged in. Since the browser automatically includes authentication credentials (like cookies) with each request, the website may process the request as if it was made by the legitimate user.

**Real-World Case Study:**
In 2016, a CSRF vulnerability was discovered in the GitHub API. Attackers could craft a malicious request that, if executed by an authenticated user, could delete repositories without their consent. The attackers used the CSRF vulnerability to trick authenticated GitHub users into sending a request to delete a repository, effectively causing data loss for the user. GitHub quickly patched the vulnerability and issued security advisories to inform users about the issue.

**Mitigation Strategies:**

1. **Anti-CSRF Tokens:**
   One of the most common and effective methods for preventing CSRF attacks is the use of anti-CSRF tokens. These tokens are unique to each user session and are included in every state-changing request, ensuring that requests can only be made by the user who originally initiated them.

2. **SameSite Cookies:**
   The SameSite attribute for cookies can be set to restrict when cookies are sent with cross-origin requests. By setting the SameSite attribute to Strict or Lax, web applications can prevent browsers from sending cookies with requests that originate from third-party websites, effectively mitigating the risk of CSRF.

3. **User Interaction Confirmation:**
   For sensitive actions, such as transferring funds or changing account settings, requiring the user to confirm their action (e.g., by re-entering their password or using multi-factor authentication) adds an extra layer of protection against CSRF.

4. **Regular Security Audits:**
   Regularly reviewing application code and performing penetration testing can help identify and fix CSRF vulnerabilities before they can be exploited by attackers.

**d) Session Hijacking**

**Overview:**

Session hijacking is a form of cyber attack where an attacker steals a valid session token, which is used to authenticate a user’s identity on a web application. With the session token in hand, the attacker can impersonate the user and gain unauthorized access to the user's account. This type of attack can occur through several techniques, including packet sniffing, Cross-Site Scripting (XSS), and Man-in-the-Middle (MITM) attacks. These methods allow attackers to intercept the communication between the user and

the web server, particularly when sensitive data, like session tokens, are transferred over insecure channels.

In a typical web application, when a user logs in, the server generates a session token, which is then stored in the user's browser as a cookie or other storage mechanism. This session token is used to authenticate the user's requests to the server. If an attacker is able to intercept this session token, they can use it to make requests on behalf of the legitimate user. This grants the attacker full access to the user's session, which can lead to unauthorized actions such as accessing sensitive information, performing actions in the user's name, or compromising the account entirely.

**Real-World Case Study:**

In 2008, attackers targeted a popular social media platform using session hijacking techniques. The attackers exploited a vulnerability in the platform's session management system, which allowed them to steal session tokens from unsuspecting users. By using techniques like packet sniffing on unsecured networks or manipulating cookies, the attackers were able to intercept and steal the session tokens. Once they obtained the session tokens, the attackers impersonated the legitimate users and accessed their private messages, account details, and other sensitive information.

This breach had significant consequences for the affected users, as attackers gained full access to their accounts, potentially causing financial loss, reputational damage, or the exposure of private information. The social media platform responded by updating its session management system and increasing security measures to prevent similar attacks in the future. However, this incident highlighted the importance of securing session management and the risks associated with improper handling of session tokens.

**Mitigation Strategies:**

1. **Secure Cookies:** One of the most effective methods to mitigate session hijacking is to use secure cookies. Secure cookies are only transmitted over secure HTTPS connections, which ensures that session tokens are not exposed during transmission on unencrypted networks. Without this protection, attackers could potentially intercept session tokens through packet sniffing on public or unsecured networks, such as Wi-Fi hotspots. Additionally, the HttpOnly flag on cookies ensures that the session token cannot be accessed via JavaScript running in the browser, reducing the risk of XSS-based attacks. By securing cookies in this manner, the likelihood of session hijacking via interception is significantly decreased.
2. **Session Expiry:** Another critical mitigation strategy is to implement session timeouts and re-authentication mechanisms. Session expiry limits the time a session token remains valid, reducing the window of opportunity for attackers to misuse a hijacked session token. For instance, web applications should automatically expire session tokens after a certain period of inactivity or a predefined duration. Additionally, sensitive actions, such as changing account settings or making financial transactions, should require users to re-authenticate or provide an additional layer of verification, like multi-factor authentication (MFA). This makes it much harder for attackers to hijack sessions and use them for malicious purposes.
3. **Encryption and Secure Communication Channels:** Ensuring that all communications between the client and the server are encrypted using protocols like HTTPS is a fundamental practice to protect session tokens. Without encryption, attackers could intercept unprotected data and

steal sensitive session information during transmission. Utilizing secure communication channels prevents attackers from accessing the session tokens in transit.

4. **IP and Device Binding:** Some advanced systems use additional layers of security by binding sessions to specific IP addresses or devices. This ensures that a session token is valid only if it originates from a particular device or IP address. If the session token is accessed from a different IP address or device, the system can detect this anomaly and invalidate the session, requiring the user to log in again. This adds another layer of protection against session hijacking by making it much harder for attackers to exploit stolen tokens from other locations or devices.