# FIT5003-Software Security

Assignment 2

Q1: Complete three labs from PortSwigger Labs, one from SQL Injection, one from Cross-Site Scripting, and one from Cross-Site Request Forgery section. Please select labs designated as PRACTITIONER or EXPERT; APPRENTICE labs will not be accepted. You are permitted to utilize the solutions and demonstrations available on the PortSwigger website for assistance. However, please do not copy walkthroughs from the PortSwigger website. Your solution should include the logical steps that lead to the exploitation, which may not be covered in the walkthroughs on the PortSwigger website. [60 Marks]

Record a video and write a report to answer the following questions for each lab. At the beginning of each lab recording, please state your name, student ID, and the name of the lab you are solving; no marks can be awarded without this information.

1. How did you identify the vulnerability? (5 Marks)

2. Which payload was chosen for exploitation and why? (5 Marks)

3. What an attacker could achieve using the vulnerability? (5 Marks)

4. How the vulnerability can be mitigated? (theoretically, no demonstration is required) (5 Marks)

The video submission must demonstrate solving the lab, addressing the questions outlined above. In case time runs short during the video, you may use the report to address any unanswered questions, making references to relevant sections of the video. However, it is important that the video includes, at a minimum, a demonstration of the lab. The report does not need to be in detail, it should briefly address the mentioned questions, i.e. it can contain one or two-line answer for each question, payloads, important screenshots (if necessary) and the video link(s). The marks mentioned above are for the videos and report combined. **The word limit for each sub-question is 200 words, i.e. maximum 800 words are allowed for Q1 per lab.**

Solution:

SQL Injection:

Link:https://drive.google.com/file/d/1zJam4iCoxlP8zdrfduAqCy2oU0Im2Mbw/view?usp=sharing

1. An SQL injection in the product category filter is the lab's vulnerability. Because user input is used in a SQL query without sufficient validation, the application is susceptible to SQL injection attacks based on UNION.
2. ' UNION SELECT table_name, NULL FROM information_schema.tables—
   **Reason:**
   The names of the tables in the database were enumerated using this payload. By combining their query with the pre-existing SQL statement using **UNION SELECT**, the attacker is able to obtain data from the information_schema.tables database, which houses metadata for all tables in non-Oracle databases.
3. The following could be accomplished by an attacker by taking advantage of this SQL injection vulnerability:

   **List Database Tables and Columns:** By obtaining the names of every table and column in the database, an attacker can learn more about the layout of the database.
   **Extract Sensitive Data:** Passwords, usernames, and other private documents are among the sensitive data that the attacker can obtain.
   **Determine Admin Credentials:** With the use of this information, an attacker would be able to determine administrator credentials and gain unauthorized access to the application's restricted regions.

4. The following techniques can be used to lessen the impact of the SQL injection vulnerability:

   **Employ Prepared Statements (Parameterized Queries):** To keep code and data apart, make sure that SQL queries are built employing parameterized queries.

   Enable robust server-side input validation to prevent any unexpected characters or patterns from appearing in user input.

   **Escaping Special Characters:** To stop malicious payloads from being executed, properly escape special characters in SQL queries.

Link for XSS:

https://drive.google.com/file/d/1-vh_Qss5u_nCsPXNVKPEWJX5nAO9CWn2/view?usp=sharing

1. The given lab contains a DOM-based Cross-Site Scripting (XSS) vulnerability where a element is dynamically generated by the document.write method using untrusted input from location.search. Because this input hasn't been cleaned up, dangerous HTML or JavaScript code can be injected by an attacker.
2. "></select><script>alert(1)</script>
   **Reason:**
   This payload departs from the pre-existing element that the document established.write using ">, and then triggers alert(1) by injecting a <script> tag. The objective is to activate the JavaScript alert function, which verifies a DOM-based XSS vulnerability by displaying the capacity to execute arbitrary code.
3. An attacker can run any JavaScript on the affected website by taking advantage of the DOM-based XSS vulnerability. This could enable them to:

   **Take User Data:** Take advantage of session cookies, data from local storage, or other private information.
   Use phishing attacks to steal login credentials by inserting phony forms or webpages.
   **Regulate User Conduct:** change the content of pages or send users to fraudulent websites.
4. To lessen the risk of DOM-based XSS:

   **Don't Use This Document.write:** To avoid executing HTML, use safer substitutes like textContent or innerText.
   **Clean User Input:** Before using data in the DOM, make sure it has been properly validated and encoded.
   **Make Use of a Secure JavaScript Module:** Use tools such as DOMPurify to weed out harmful information.
   **Content Security Policy (CSP):** To stop injected scripts from running, enforce a strict CSP.

**CSRF:**

Link:
https://drive.google.com/file/d/1v6ERcmXr2ncQua6RKC8Zg2avV08G6mRa/view?usp=sharing

1. This lab contains a Cross-Site Request Forgery (CSRF) vulnerability where the HTTP request method determines the token validity. For POST queries, the server verifies the CSRF token; however, for GET requests, it does not. By switching the method to GET, an attacker can get beyond the CSRF security and carry out harmful operations such as changing the victim's email address without authorization.
2. in order to circumvent CSRF protection, a forged GET request was used as the payload of the hack. The attacker changed the request method to GET and sent the required parameters directly through the URL to alter the user's email address because the server only validated the CSRF token for POST requests. This completely got around the token validation, which made the assault possible.
3. An attacker could take advantage of this vulnerability to carry out illegal operations on behalf of the victim, like altering other personal data or their email address or account settings. Depending on the precise purpose of the targeted endpoint, this could lead to account takeover, loss of control over the victim's account, or unauthorized changes to sensitive data.
4. In order to address the CSRF issue in which the request method determines the token validation:

   **Verify CSRF Tokens Using Every Request Method:** Limit state-changing operations to POST or other non-GET methods, and make sure that CSRF tokens are validated for both GET and POST requests.
   **Enforce the attribute of SameSite cookies:** If you want to stop cookies from being delivered with cross-site requests, set the SameSite property.
   **Put Strict Content Security Policies (CSP) in Place:** To stop CSRF attacks from loading external resources, implement a rigorous CSP.

Q2: Download the Q2.html file from Moodle. Assume you are browsing monash.edu, and it is hypothetically vulnerable to various web attacks (although it is not). While navigating monash.edu, assume you open another tab in the same browser, and visit attacker.com (assuming attacker convinced you to do that). You click the Submit button on the attacker.com webpage, which contains Q2.html, initiating attacks on monash.edu. Examine Q2.html (you can open the file in the browser and intercept the request in BurpSuite if desired) and respond to the following questions. No video is required for this question. **The word limit for each sub-question is 200 words, i.e. maximum 600 words are allowed for Q2. [20 Marks]**

1. Which vulnerability/vulnerabilities attacker.com is trying to exploit on monash.edu? (please explain the scenario outlining how this exploitation could occur) (10 Marks)

2. If successful, what is the consequence of the attack(s)? (5 Marks)

3. What mitigation(s) would you suggest for monash.edu to counter attack(s) launched by attacker.com? (5 Marks)

Note: The parameter values in the HTML file are URL encoded.

Solution:

1. **Cross-Site Request Forgery** is probably the main flaw being used in this situation (CSRF). A cross-site request forgery (CSRF) attack involves deceiving the user's browser into submitting a malicious request to a reliable website (monash.edu in this example), where the user is authorized. This is how it operates:

   The user has an active session and is already logged into monash.edu in one tab.
   The attacker persuades the user to open a second tab to attacker.com, which is home to Q2.html.
   The malicious form uses the user's authenticated session to send a fake request to monash.edu when the user clicks the Submit button on Q2.html.
   In the event that monash.edu lacks CSRF safeguards (such as CSRF tokens), it will interpret this request as a valid user action.
   Additional possible weak points
   **Cross-Site Scripting (XSS):** This can happen if there are JavaScript injections in the HTML file that are directed at monash.edu and the website does not check input parameters.
   **Clickjacking:** The victim may unintentionally carry out actions on monash.edu if the attacker embeds the website in an iframe.
2. The CSRF attack may have a number of serious repercussions if it is successful, depending on the user's session privileges and the nature of the request. Among the possible results are:

   **Account Takeover or Data Manipulation:** If the program processes sensitive financial transactions, the attacker may be able to transfer money, alter personal data, or even alter account settings.
   **Unauthorized Actions:** The attacker may post messages, alter the password, or erase crucial data, among other things.
   **Privilege Escalation**: If the user has administrator rights, the attacker may be able to take over and alter permissions, disable security settings, and add new users, among other crucial functions of the program.
   **Exfiltration of Sensitive Data:** When used in conjunction with XSS, an attacker may be able to obtain cookies, session tokens, or other private data.

3. **Put Anti-CSRF Tokens in Place:**
   Before processing any state-changing requests, include unique, unpredictable CSRF tokens in all sensitive forms and validate these tokens on the server side.

   **Put Cookies on the SameSite Attribute:**
   To stop browsers from transmitting cookies together with cross-site requests, set the SameSite property in cookies to Lax or Strict. This would prevent unauthorized domains from launching CSRF attacks.

   **Content Security Policy (CSP) should be enabled.**
   To stop injection attacks, enforce a tight CSP and limit the sources of stylesheets and scripts. This lessens the impact of XSS attacks.

   **Verify Data Entry and Encrypt Output:**
   If an attacker tries to change the URL or insert harmful scripts, you should always verify user inputs and use the appropriate output encoding to stop XSS attacks.

**Put X-Frame-Options Header to Use:**
To stop clickjacking, use the X-Frame-Options header to prevent the site from being embedded in an iframe on untrusted websites.
These techniques can help monash.edu greatly lower its vulnerability to CSRF and other web-based attacks.

Q3: Assume you visit monash.edu and it tries to talk to lms.monash.edu, the browser issues an OPTIONS method to lms.monash.edu and gets a response, below is the HTTP request and its response:

```
OPTIONS /doc HTTP/1.1

 Host: lms.monash.edu

 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:71.0)

 Accept: text/html,application/xhtml+xml,application/xml

 Accept-Language: en-us,en;q=0.5

Accept-Encoding: gzip,deflate

Connection: keep-alive

Origin: monash.edu

Access-Control-Request-Method: POST

 Access-Control-Request-Headers: x-requested-with

 HTTP/1.1 204 No Content

Date: Mon, 01 Dec 2008 01:15:39 GMT

Server: Apache/2

Access-Control-Allow-Origin:

* Access-Control-Allow-Methods: POST, GET, OPTIONS

 Access-Control-Allow-Headers: x-requested-with

Access-Control-Allow-Credentials: true

 Access-Control-Max-Age: 86400

Vary: Accept-Encoding, Origin

Keep-Alive: timeout=2, max=100

Connection: Keep-Alive
```

Explain the Cross-Origin Resource Sharing (CORS) HTTP headers in the above HTTP request and response. Would browser change future requests based on the above HTTP response? **No video is required for this question. The word limit for Q3 is 300 words. [10 Marks]**

Solutions:

Web browsers use a security feature called Cross-Origin Resource Sharing (CORS) to limit the access that a web server's resources can have from various origins. In order to find out if the target server will accept the real request, preflight requests, or HTTP OPTIONS requests, are sent by monash.edu in an attempt to contact with lms.monash.edu. Now let's dissect the headers:

**Headers for HTTP requests:**
Source: monash.edu
The request's origin is indicated in this header. In this instance, monash.edu is trying to use lms.monash.edu's resources.
**POST is the access-control-request-method**
This header is sent by the browser to lms.monash.edu to let them know that the real request is going to be made via POST. Preflight requests use this to determine whether the requested HTTP method is permitted.
**Methods of Access Control Allow: POST, GET, OPTIONS**
indicates the HTTP methods that cross-origin requests are permitted to use. POST, GET, and OPTIONS are allowed in this scenario, providing flexibility for various request kinds.
**Access-Control-Allow-Headers:** x-requested-with Signals the server's acceptance of the x-requested-with header that was included in the preflight request. This enables the custom header to be included in the request itself.
**Credentials-Access-Control-Allow: true**
This header indicates that credentials (such as cookies and HTTP authentication) may be supplied in the cross-origin request and that the server accepts them. The client can make requests that are authenticated thanks to it.

**Maximum Age, Access, Control: 86400**
This indicates how long the preflight answer can be cached, in this example 86400 seconds or 24 hours. This increases efficiency by preventing repeated preflight requests for the same origin during this time.
The browser will cache this preflight response for 24 hours (86400 seconds) based on the CORS headers. After that time, if the subsequent requests from monash.edu to lms.monash.edu meet the approved HTTP methods, headers, and credentials criteria, they won't require any further preflight checks.
To limit access to only reliable origins, it is advised to replace Access-Control-Allow-Origin: * with a specific domain (Access-Control-Allow-Origin: https://monash.edu).