**Hands-On Lab: XSS Scripting Attack**

**Introduction: XSS attacks and their types**

According to the OWASP foundation, the Cross-Site Scripting (XSS) attacks are “a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites.”

There are several types of XSS attacks:

* Reflected XSS:

Reflected attacks are the ones where “the injected script is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as a part of the result” (OWASP). Links to the websites where the malicious code is injected can be transferred through emails and trusted websites to a user. Then, the malicious code is executed on the user’s web browser side since the code came from the “trusted” source.

* Stored XSS:

Stored attacks are the ones where “the injected script is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc.” (OWASP). When the user requests the information from the web server, he also retrieves the malicious script that is then executed on his side.

* Other XSS attack types:

According to OWASP, there are more less widespread types of the XSS attacks, such as Blind Cross-site Scripting, where “the attacker’s payload saved on the server and [is] reflected back to the victim from the backend application,” and DOM Based XSS, where “the attack payload is executed as a result of modifying the DOM ‘environment’ in the victim’s browser used by the original client-side script.”

Thus, there exists a variety of possible XSS attacks; however, the current Hands-On Lab will focus on the first two types: the reflected and the stored attacks.

**Criteria for the selection of the attack**

In the modern technological world web applications are now much more prevalent than desktop applications (although, they may compete with the mobile applications). Thus, their vulnerability to different types of attacks is crucial to the safety of the Internet users who access them daily.

In general, the XSS attacks is just as popular as the SQL injection attack, and it is especially affecting on the websites they do not sanitize their inputs. Besides the popularity, the attack was picked for its ease of reproduction (only basic JavaScript knowledge is required), which is beneficial for the learning purposes of the lab.

Thus, the main criteria for the selection of the were its popularity and reproducibility.

**Lab objectives**

The developed Hands-On Lab on the XSS scripting introduces students to the basics of the reflected and stored XSS attacks. It also teaches the students basics of the web application’s protection against such kind of attacks. Thus, the objectives of this lab are:

* Learn about reflected and stored XSS attacks
* Perform a basic XSS attack by writing malicious JS code
* Learn the measures for the protection of web application against XSS attacks

The achievement of the objectives is guaranteed by the supplied tools and described techniques.

**Lab topology**

Graphical user interface, application

Description automatically generated with medium confidence

**Details of the used tools and techniques**

The Hands-On Lab leverages the functionality of a Flask web application, specifically developed for the current lab.

The Flask web application is based on Python and uses SQLite to access the database to demonstrate different types of the XSS attacks. The application is structured in a way that it responds to the HTTP requests made directly from the user’s browser by leveraging the route system available within the Flask framework and serves the HTML pages from the predefined templates (see the templates folder of the application), substituting certain attributes with the values retrieved from the database.

For the successful completion of the lab the student has to have a client installed (which can be any browser, e.g. Chrome or Firefox), and the Flask application running (which can be accessed as a Docker container or as a Flask application, but more on this in the lab instructions). The students will also be instructed to write JavaScript code and execute it by placing it into the input elements on the web page served by the application; for the purposes of writing JavaScript code one can use a notepad or VSCode editor. For the illustration of the protective measures against the attack the student should have a Python IDE installed, for example, PyCharms. Thus, all the above-mentioned software, such as a web browser, Docker, and Python and JS IDE are necessary for the successful completion of the lab.

**Instructions**

In this lab you will be introduced to the basics of executing reflected and stored XSS attacks on a vulnerable web application. You will write and execute JavaScript code to retrieve sensitive data and perform malicious actions in the client browser. Then you will learn how to protect your Flask application from such attacks and will modify the code according to the security standards.

The lab setup was tested on Windows 11 machine, however, if you are using Linux or MacOS, the setup should work analogously.

Best of luck!

**Step 1. Setup**

In order to complete the lab, make sure that you have installed: a web browser (such as Chrome or Firefox), Python and pip packet manager, Docker, and an IDE for editing JavaScript and Python code (e.g. you can use Notepad, VSCode, PyCharms, etc.).

There are two options of how you can run the application:

* By activating the virtual environment and running the application from the terminal;
* By deploying a Docker container

**Option 1:**

* 1. Navigate to the root folder of the project.
  2. In the terminal, run “pipenv shell” – this activates a virtual environment.
  3. In the same terminal, run “pip install -r requirements.txt” – this installs the necessary libraries, including Flask, needed for the correct work of the application.
  4. Run “flask run” in the terminal.
  5. Navigate in the browser to localhost:5000 – you should be able to access the index page of the application.

**Option 2:**

* 1. Make sure Docker is running.
  2. Navigate to the root folder of the project.
  3. In the terminal, run the command “docker build -t xss-demo:latest –build-arg APP\_IMAGE=python:3.7. -f Dockerfile .”
  4. After the image is built, run “docker container run -p 5000:5000 -dit –name xss-demo xss-demo:latest”
  5. Once the container is running, navigate in the browser to localhost:5000 – you should be able to access the index page of the application.

Graphical user interface, text

Description automatically generated

Fig. 1: Index.html page of the application.

**Step 2. Basic functionality**

On the index.html page of the application you can see three main sections: the article, the search field, and the submit new comment field. All of them are interactive.

To familiarize yourself with the application’s functionality, perform the following actions:

**2.1.** In the bottom comment field enter any text comment (e.g. “A very nice article!”) and click on “Submit new comment” button. The new comment should appear on the page.

**2.2.** In the search field in the middle try entering part of what you wrote entered in the comments (e.g. “nice”). The page should refresh and display your comment.

**2.3.** Try entering a non-existent value, e.g. “security”. See that no comments are displayed, since no such comments exist.

Graphical user interface, text, application, email

Description automatically generated

Fig. 2: Comments section of the page.

**Step 3. Executing JavaScript**

Remember that usually JavaScript code can be injected into the page using <script> tags that are placed in the head section of the HTML page. Within the script tags you can write basic functions that execute once the page loads, e.g. alert().

Firstly, let’s remember JavaScript and try to execute some basic functions.

**3.1.** Open the DevTools of your browser (e.g. by pressing F12 key in Chrome) and navigate to the “Console” tab.

**3.2.** In the console, try logging something, using console.log() command, typing the text within the quotations marks inside the parenthesis.

**3.3.** Next, try to use the console to display a pop-up window with the text “Hello user!” (hint: user the alert() function).

**3.4.** Let’s try something more complicated. Inspect the HTML of the page and find the id attribute of the heading.

**3.5.** In the console, write a function that gets the element by id and modifies its text to “You got hacked!” (hint: use getElementById() function and .innerText property).

**Step 4. Reflected XSS**

Now is time to hack the website. This step will demonstrate the use of reflected XSS on the page.

**4.1.** Navigate to the filtering field of the page.

**4.2.** In the field, try to insert an alert function to display any message you like. Don’t forget the <script> tags!

**4.3.** Click on “Filter comments” button. Did your message get displayed?

**4.4.** Try using different functions in the filtering field: e.g. try to change text of the HTML elements, log information to the console, etc. Feel free to play around with the field.

Pay attention to what happens to the URL of the page. You can see that the script gets inserted after the q parameter, specified in the url, e.g.:

*http://localhost:5000/?q=<script>alert(“You got hacked”);</script>*

**4.5.** Put your script in after the = of the q parameter in the address bar. Press Enter to send the request to the application. Did the script work?

Often hackers would send malicious links with the scripts embedded as parameters to the users in hope that those will click on them. Once the user clicks on such a link, the script gets executed and the hacker wins.

Graphical user interface, application

Description automatically generated

Fig. 3: Reflected XSS attack execution example.

**Step 5. Stored XSS**

This step will demonstrate you how a malicious script can be injected in the database of the application and then executed by the user upon entering the page.

**5.1.** Navigate to the comment field of the page. Whatever you enter in the comment field, as we already mentioned, gets stored in the database. It means that the next time you will enter the page, the information will be displayed (or, in our case, an injected script will be executed).

**5.2.** In the comment field, try to enter a script that alerts any message of your choice.

**5.3.** Press on “Submit new comment” button.

**5.4.** See that the script got executed.

**5.5.** Close the window with the website.

**5.6.** In the address bar, navigate to localhost:5000 again. Did the script get executed?

**Step 6. Stealing sensitive data**

XSS attacks can be used to still sensitive information stored on the user side of the application, for example, cookie of the session, personal user tokens, etc.

Pay attention to line 17 of the app.py and the line 9 of the index.html files. These lines generate a unique user identification token and, upon the return of the index.html page, set the token to a generated value in the local storage of the user’s browser.

To “steal” the token, we will leverage the stored XSS attack:

**6.1.** Navigate to the comment field of the page.

**6.2.** Write a function that gets the token item from the local storage and displays in the console or the pop-up on the screen (hint: use localStorage variable to access local storage).

**6.3.** Close the window with the website.

**6.4.** In the address bar, navigate to localhost:5000 again. Do you see the session token displayed?

Thus, the next user who will enter the website will have his token “stolen”.

**Bonus task:** write the code for doing the same but performing reflected XSS attack.

Graphical user interface, text, application, email

Description automatically generated

Fig. 4: Stolen user token displayed.

**Step 7. Protecting from XSS attacks**

General approach for protecting from XSS attacks is to sanitize any inputs that happen on the page and disallow passing and execution of any unsafe JavaScript. Besides, in case of the Flask application, you can do the following:

**7.1.** Open the folder with the project in your favorite IDE.

**7.2.** Navigate to templates/index.html file.

**7.3.** Replace *{% autoescape false %}* with *{% autoescape true %}.* This provides standard HTML context filtering for variables in the served templates.

**7.4.** Additionally, in the app.py file, modify the code in the following way:

**7.5.1.** Import the make\_response function from flask.

**7.5.2.** Do not immediately return the rendered template: wrap the template rendering in

make\_response function and store it into a variable.

**7.5.3.** Pass the following to the headers of that variable: *‘Content-Security-Policy', "script-src 'none'"*. This disallows unsafe inline JavaScript.

**7.5.** Go again through steps 4-5. Are you able to execute the scripts?

**Congratulations, you have completed the lab!**