Brno University of Technology
Faculty of Information Technology

WAP Course Project

Educational Web Application About Web Vulnerabilities

Jan Zbořil
xzbori20@stud.fit.vutbr.cz

General overview

This web application for educational purposes demonstrates few common vulnerabilities often found in web applications. In forms of labs, the user can learn how to exploit these vulnerabilities and how to prevent such exploits. The labs are targeting simulated web servers, which are vulnerable to specific exploits. The exploits shown are usually the easiest to perform, but they are still very dangerous. However, production servers are normally immune to these most simple versions of the exploits.

The application focuses on the following vulnerabilities:

- Stored XSS
- Reflected XSS
- Clickjacking
- CSRF (Cross-Site Request Forgery)

In each lab, the user will learn how to prevent the vulnerability and can continue to browse through linked external sources containing more information about given vulnerability.

Directory Structure

```
root
└ /api                    client side JavaScript – page and lab logic
  /public                 static files
  └   fonts
      images
      styles              CSS files
  /server-scripts         Server side scripts for runtime storage, etc.
  /views                  EJS templates for HTML pages
  Dockerfile
  documentation.pdf
  index.mjs               main application file
  package.json            dependencies
  README.md
```

Installation

Application runs at the `http://localhost:8000`. Requirements are specified in the `package.json` file. `NodeJS` and `npm` have to be installed first. Install other dependencies using the `npm install` command.

Local Installation

Run the application via `npm start` or `nodemon start`.

Docker Local Build

Build the image via `docker build -t wap .`

Run the container using `docker run -p 8000:8000 wap`

Docker Hub Installation

See https://hub.docker.com/repository/docker/rolllikerollo/wap/general

Pull the image: `docker pull rolllikerollo/wap:final`

run the container using `docker run -p 8000:8000 rolllikerollo/wap:final`

Used Technologies and application logic

The app is build using JavaScript and Node.JS runtime. It uses the EJS as its templating engine. All logic is written in JavaScript, views are combination of HTML with EJS templates and styling is provided by plain CSS.

The app contains of a HTML web interface which challenges the user to complete an easy task about the given topic – to carry out an attack exploiting a specific vulnerability. The app therefore simulates a web environment, where the user has access to the vulnerable website, and sometimes a special attacker website, both of which he uses to complete the assignment.

Vulnerable pages inside the labs are views rendered inside an `iframe` tag. The pages inside the `iframe` are intentionally made vulnerable either by not sanitizing their inputs or by not using the appropriate HTTP headers. In some cases, the lab web page communication with server is only simulated. This is the case when the entire communication is not necessarily needed to show the point of the attack. This approach is used for the stored XSS attack, where the storage of the ‘web server’ is represented by local storage provided by the user’s web browser. The full server functionality with sending/receiving HTTP traffic is used, for example, in the reflected XSS attack lab.

The logic of the web server is in the `index.mjs` file. It creates and runs the app and provides the routes, views, responses to HTTP requests, cookies, etc. It utilizes functions defined in files located in the `server_scripts` directory, which act as a runtime storage for some labs. Deployment of a full fledged database seemed as an extra complication which is not needed, as any of the web page parts are not supposed to store permanent files.

The server routing renders a requested website upon getting a `GET` or `POST` request, sometimes with an appropriate data embedded to the view using the EJS templates.

The logic of the labs is written in client side JavaScript. Sometimes, this code only simulates the logic of the web server/web page, sometimes it sends requests to the actual server which runs the application. This is the case with stolen cookies during the stored XSS attack, where the script send the cookies to the server, which stores them and they can be retrieved by an 'attacker's site'.

Among the lab logic, the client side scripts contains various element listeners and other ways to modify or further improve the interactivity of the page. For example, the entire clickjacking page is done using the client side JavaScript and local storage.

All elements of the website are styled using CSS files, which can be rewritten to change the look. The most used colours are defined as variables at the top of a CSS file and thus can be changed once to change the all occurrences of the colour at a page.

All files containing code are commented at the function level. A link is stated if the function is taken from online source.

Entire application is ready to be containerized by using the attached Dockerfile.

The Labs

Clickjacking

The objective of the clickjacking lab is for the user to crate few users and then try to send money to one of them. Upon a click on send money button, the user, to which money was to be sent, is deleted. This is because the lab user became a victim of clickjacking. The malicious `iframe` is then shown together with a paragraph about what happened, why, and what are the common defense mechanisms. A user

can open a safe version of the page, where the `iframe` is not loaded, because it's web page is protected using the `Content-Security-Policy` header set to "`frame-ancestors none`".

The complete logic of this example is written as a client-side scripts. Only the `iframe` contents are fetched from the server. User database is simulated using local browser storage.

Stored XSS

Stored XSS lab presents user with two pages – one vulnerable to XSS and the second is supposed to simulate "attacker's page or server".

User first has to examine the page and find a way how its weak security could be exploited. User has to successfully carry out a XSS attack via crafting a HTTP request which will send user's cookies to attacker's server. The cookie to stole is not secure and is generated from user's session ID on the server. This request is then stored on the server under the session ID and is fetched back to be shown on the "attacker's website". The logic of the lab is client side – this eliminates the possibility to infect the server with actual XSS code and display it to other users. Logic for storing and displaying the stolen cookies is on the server.

Reflected XSS

Reflected XSS lab challenges a user to examine a vulnerable website and craft a malicious link which could be use together with phishing or social engineering to make a victim to click it. In the example solution, the user is bamboozled by one very common viral video.

The vulnerable also shows the simulated "browser search bar" with current url, where the `GET` parameters are encoded. This is to show user a malicious link which is then sent to the victims. This lab logic is completely on the server, so the "reflection" of the user input actually happens.

CSRF

CSRF or Cross-Site Request Forgery lab presents user with three interactive elements – first is the vulnerable bank website, where the victim is logged in, and he is thus authenticated. Then there is a code editor (source [9]), where an user can directly write an HTML code which is rendered below and simulates a malicious blog. The goal of the user is to write a malicious script into the blog site, which will make a request to the bank site using the victim's authenticated session. To be able

to this, attacker has to get accustomed to the bank page interface and to find correct IDs of elements to craft a request. During this, the user can use the authenticated session to send money as a victim, simulating the normal transfer. This victim-authenticated transfers are then showed together with the ones made by an attacker without directly manipulating the bank page.

The logic of the bank site, editor and malicious blog is handled locally. However there is and endpoint at the server simulating the bank API, to which the malicious blog sends the request for money transfer. The server then stores the stolen money from the request to its runtime storage under a session ID key. This transaction logs can be retrieved by the victim's bank account page and thus show the completed CSRF attack.

Testing

Testing was done manually by few real users both on the directly run and dockerised version of the application. Reported bugs were remedied for.

References

1. https://portswigger.net/web-security/cross-site-scripting/preventing
2. https://portswigger.net/web-security/cross-site-scripting/stored
3. https://owasp.org/www-community/HttpOnly
4. https://owasp.org/www-community/attacks/xss/
5. https://owasp.org/www-project-web-security-testing-guide/v41/4-Web_Application_Security_Testing/07-Input_Validation_Testing/01-Testing_for_Reflected_Cross_Site_Scripting.html
6. https://portswigger.net/web-security/cross-site-scripting/reflected
7. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/frame-ancestors
8. https://cheatsheetseries.owasp.org/cheatsheets/Clickjacking_Defense_Cheat_Sheet.html
9. https://css-tricks.com/creating-an-editable-textarea-that-supports-syntax-highlighted-code/
10. https://www.youtube.com/watch?v=vRBihr41JTo
11. https://portswigger.net/web-security/csrf
12. https://owasp.org/www-community/attacks/csrf
13. https://www.w3schools.com/howto/howto_js_autocomplete.asp