Cybersecurity – Security in Software

Practical Assignment

This assignment includes:
- practice with the *shellshock* vulnerability
- practice with both static and dynamic techniques to find vulnerabilities in a web application.

Delivery:
- Report with evidence and explanations requested in each question.
- Submit on Moodle until November 5, 2022

A. The Shellshock attack

In 2014 a vulnerability was identified in the *shell* application of several Linux-based systems, which became known as *Shellshock*. One way to exploit this vulnerability was for the attacker to remotely execute commands or programs on the vulnerable machine. The following instructions are based on the Shellshock vulnerability Lab published by the SEED Labs project (https://seedsecuritylabs.org/Labs_20.04/Files/Shellshock/Shellshock.pdf).

Present a report regarding the following points:

1. Setup the Lab by running a container[1] in your machine with the image available at `jsimao/seed-shellshock`

   a. Run the vulnerable system which has an Apache Server listening on port 80 and two scripts which use the vulnerable bash shell version. In your machine the system will be available at port 8080.

   ```
   docker run -it -p 8080:80 --name seed jsimao/seed-shellshock
   ```

   b. Ensure the system is running by calling the two programs available in the configuration of the Apache Server (`/cgi-bin/getenv.cgi`, `/cgi-bin/vul.cgi`). Use your favorite *browser* and the *curl* tool and show the results.

[1] https://dockerlabs.collabnix.com/docker/cheatsheet/

2. Now the objective is to execute some arbitrary commands in the vulnerable machine, which is running as a container, by making an HTTP request to the `/cgi-bin/vul.cgi` script.

   The `/cgi-bin/vul.cgi` script contains:

   ```
   #!/bin/bash_shellshock

   echo "Content-type: text/plain"
   echo
   echo "****** Environment Variables ******"
   echo "Hello World"
   ```

   The attack does not depend on what is in the CGI program, as it targets the vulnerable bash program, which is invoked before the actual CGI instructions are executed. Because the injected commands have a plain-text output, and you want the output returned to you, your output needs to follow a protocol (the HTTP response protocol): it should start with `Content-type: text/plain`, followed by an empty line, and then you can place your *plain-text* output. For example, if you want the server to return a list of files in its folder, the injected command should be:

   ```
   echo Content-type: text/plain; echo; /bin/ls -l
   ```

   a. Get the server to create a file inside the `/tmp` folder. You need to get into the container to see whether the file is created or not, or use another Shellshock attack to list the file created.

   b. Get the server to delete the file that you just created inside the `/tmp` folder.

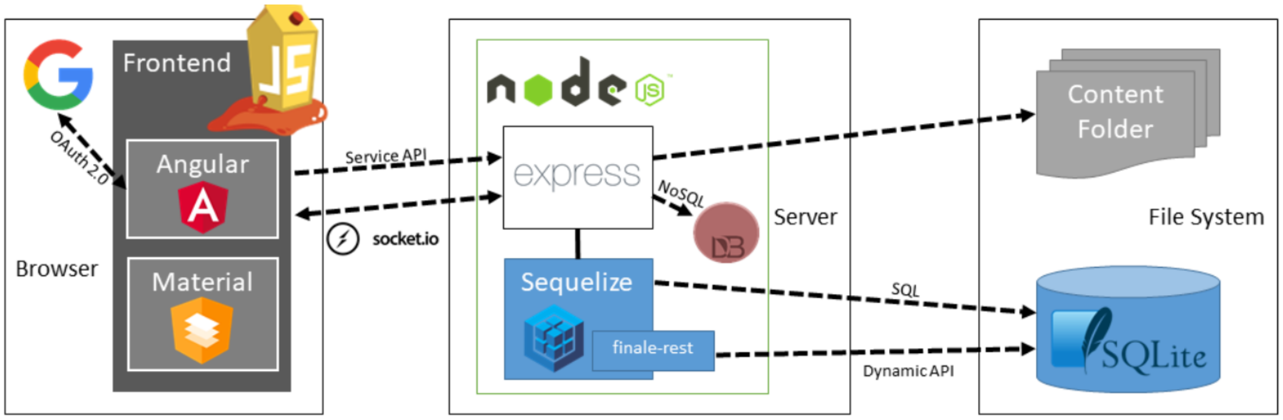
   c. Will you be able to steal the content of the shadow file `/etc/shadow` from the server? Why or why not?

   d. HTTP GET requests typically attach data in the URL, after the ? mark. This could be another approach that we can use to launch the attack. Can we use this method to launch the Shellshock attack?

B. Static and Dynamic analysis

In this part you will use static and dynamic analysis techniques targeting the Juice Shop web application. This application has purposeful programming errors which can be used to gain knowledge about vulnerabilities in general and web applications in particular.

The Juice Shop architecture includes a frontend developed in Angular, a backend developed in node.JS and the use of relational and non-relational databases, as illustrated in Figure 1.



Static analysis

1. Get the source code for the Juice Shop app, https://github.com/juice-shop/juice-shop and add the code to your group's GitHub repository.

   A message was sent through moodle with a link to create a public repository within the organization "isel-deetc-computersecurity".

2. Check that a GitHub Action to run CodeQL platform CWE analytics is executed when there is a push event in the repository. Update the CodeQL by changing the description of the CodeQL workflow:
   a. Change action version to v2;
   b. Change the workflow trigger to be manually triggered. (https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows#workflow_dispatch)

3. On the output logs after executing the action, identify the step and command used by the CodeQL Action to initialize the code analysis database.

4. Consider the CWE-89 vulnerability “Improper Neutralization of Special Elements used in an SQL Command” (https://cwe.mitre.org/data/definitions/89.html). Look for the entry “Database query built from user-controlled sources” under the routes/search.ts file in the list of vulnerabilities detected by Github Action CodeQL. Justify why CodeQL identified this code as vulnerable. Include *source* and *sink* information. (more information about SQL Injection can be found here https://owasp.org/www-community/attacks/SQL_Injection)

5. Static code analysis can have errors such as false positives or false negatives. Explain the difference between these two situations using an example.

Dynamic analysis

6. Check the correct access to an instance of the Juice Shop application running at address http://10.62.73.125:40**<GROUP>**, where <GROUP> is your group number (01, 02, 03, 04, …, 12). This instance is only accessible at the ISEL campus or connecting using ISEL’s VPN.

7. With the browser's programmer tools (e.g. F12 in chrome or firefox) discover the path that gives access to the ratings list - “score board”, which is accessed in one of the javascript files loaded by the frontend. Present the actions taken. This question corresponds to the “Find the carefully hidden 'Score Board' page” challenge:
https://pwning.owasp-juice.shop/part2/score-board.html

8. Perform an SQL Injection attack on the Login form, targeting the admin user. Tutorial help available at this link:
https://demo.owasp-juice.shop/#/hacking-instructor?challenge=Login%20Admin.

9. Install and verify the correct operation of the Zed Attack Proxy (ZAP) available here https://owasp.org/www-project-zap/:
    a. In the “Quick start” option, choose the “Manual Explore” option, and indicate the site https://www.example.org, with the HUD active, choosing the Firefox browser (it may be necessary to install the browser). Check you can access the site and the ZAP commands in the same window.
    b. Access the Juice Shop application through the browser launched in the previous point.

10. Using the ZAP tool, find out the password of the admin user `admin@juice-sh.op`. The password starts with “admin” and has a 3-digit numeric suffix. Show how you can use fuzzing to find the correct password.

11. Look for products with the word “Lemon”. Notice the address bar. Try changing the search criteria to “Lemon1” directly in the address bar. On the results page, which HTML element is used to present the search criteria text?

12. Solve the “DOM XSS” challenge by injecting the text <iframe src="javascript:alert(`xss`)"> so that the browser has to render a page with the injected text (https://pwning.owasp-juice.shop/part2/xss.html#perform-a-dom-xss-attack)

13. Show how to solve the “Post some feedback in another users name” challenge using the ZAP proxy.

14. Describe how, through a Cross-site scripting (XSS) and social engineering attack, an attacker can obtain the cookie named “token” stored in the victim's browser. You don’t have to perform each of the steps, but you should include a compelling explanation of the process and describe what could be obtained from the “token”.