DD2525 – Lab 2
Web Application Security

Deadline: May 7, 2021

1 Introduction

In this assignment we provide a lite forum application, which is a fork of the *forumerly* project available at `https://github.com/jayvolr/forumerly`. The forum enables registered users to interact on various topics of interest like technology, entertainment, sports, as well as a general discussion on the web site itself. We have modified the forum application by inserting a few vulnerabilities which you are expected to find, exploit, and fix.

Goal of the assignment The goal of this assignment is to detect, exploit, and fix vulnerabilities in *forumerly*. To do so, you may need to read articles and blog posts that describe these vulnerabilities and the corresponding fixes. When implementing a fix, you should also discuss how existing web defenses (such as those presented in Lecture 4 and 5) can help protecting against the attacks that you found.

2 Overview of the web application and frameworks/languages

The web application contains 9 fixed topics which allow registered users to engage in conversations either by starting a new thread or by replying to existing threads. A user needs to be logged in in order to create a new thread or reply to existing threads, however any user can read existing threads.

forumerly allows registered users to edit their profiles, e.g., by uploading a profile picture or adding information about themselves. However, there exists only one admin user and it is not possible to create an admin profile directly from the *forumerly* interface.

The project is built with ExpressJS (NodeJS), MongoDB, Redis and our Authentication Service (we are reluctant to trusting another service for authentication). Third-party services are delivered into a Docker container. Figure 1 provides a high level overview of *forumerly*'s architecture.

forumerly is a web application that listens on port 3000. You can open `http://localhost:3000` in a browser after setting up the server (see below how to) and explore it. The source code of the application is located in the folder `forumerly` provided with the assignment. It has 3 internal folders: `public` contains static resources like CSS, images and client-side JavaScript. All files in this folder are available for download by the browser and may be included in HTML pages. `routes` contains server-side JavaScript files that handle client requests. This is the business core logic of the application. You can read more about routing in ExpressJS at this link. `views` contains templates for Web pages. At runtime, the template engine replaces variables from a template file with actual values, and transforms the template into an HTML file sent to the client. You can see what forms are rendered by the template code even if you do not have the necessary user permissions to open these forms in the browser. The root folder `forumerly` contains some utility files and an entry point of the NodeJS application `app.js`.

If you are curious see what is stored in your MongoDB database you can use MongoDB Compass tool `https://www.mongodb.com/products/compass`.

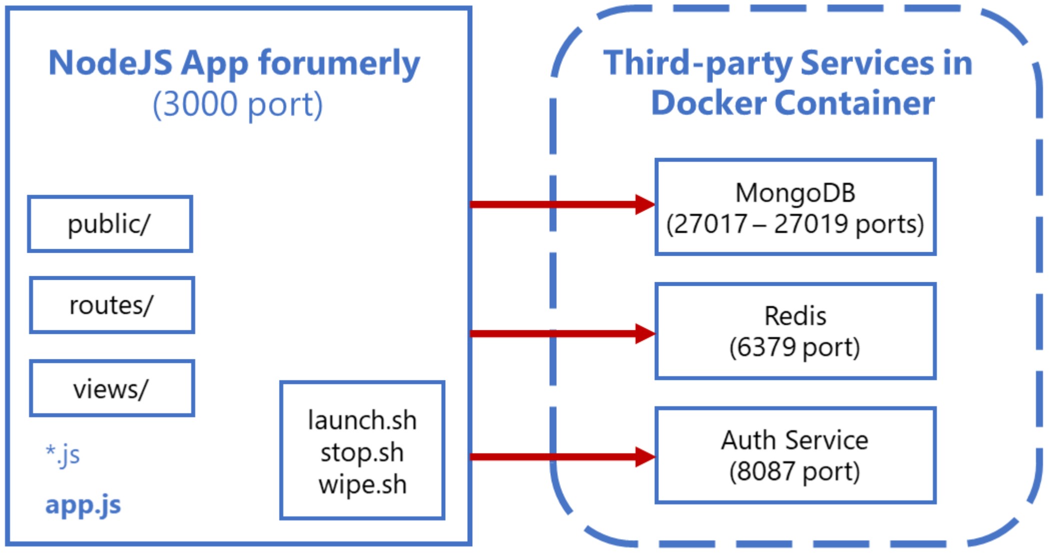

Figure 1: The project architecture

The zip archive with source code also contains the folder `tools`. The files from this folder are needed only for subsection 5.4.

3 A primer on NodeJS engine

Designed with scalability in mind, Node.js (and JavaScript in general) is an asynchronous event-driven framework. When an application wants to perform a potentially blocking action, such as opening a file or writing to the network, it registers a callback function, triggers the relevant action, and terminates. When the action completes (e.g., the file was opened), the callback event is called by the event loop.

The event-driven model is extremely scalable, as threads never ”sit and wait”, but it introduces problems when a function takes long time to complete. Since the Node.js server runs only one thread per core (by default), such a long-running function would take up the entire core. NodeJS is particularly susceptible to vulnerabilities as you will notice in this lab. You can read more about NodeJS here.

4 Getting Started

Download the zip file from the course web page in Canvas. Once you have downloaded the project, please check the requirements to run the application.

4.1 Setup

1. To run the forum application you need the NodeJS engine; follow the instructions at `https://nodejs.org/en/` to install it in your computer. Note that you might already have NodeJS installed

from Lab 1.

2. Since the web application uses third-party services, you need to install Docker. Please follow the instructions at `https://docs.docker.com/install/` to install docker. Note that you might already have NodeJS installed from Lab 1.

3. Once docker is installed, run the bash script `launch.sh` present in the zip folder. This script launches the services needed to run the application: MongoDB with the database fixtures, Redis for user sessions and the Authentication Service.

4. Node applications usually use several third-party packages; to install all of them, run the command `npm install` in the *forumerly* project root.

In folder *forumerly* you will find two more scripts, `wipe.sh` and `stop.sh`. `wipe.sh` will stop and remove the services from the docker engine. Therefore, you will lose all the information in the *forumerly* database if you added any data to it. The `stop.sh` command will stop the services without deleting the containers. These scripts are useful if you want to clean up the database in case it is corrupted. In this case you should run wipe.sh followed by `launch.sh`, i.e., deleting all the services and then creating them again.

4.2 Launch application

At this point, you are ready to run the forum application. Go to the root folder of the project and type `node app.js` – this command will launch the application server in port 3000. Now, open your browser in port 3000 (type localhost:3000) and get ready to break and fix.

You can use your favorite IDE or a text editor to explore the code. We recommend to install and use Visual Studio Code. The VS Code has built-in debugger for NodeJS and you can see how your requests are processed in real time. This can be useful when preparing exploits and figuring out how some features work.

4.3 Troubleshooting

- **The aplication is not running in the browser:** Check that port 3000 is free, this is the one that *forumerly* uses

- **The application runs, but I cannot login or signup:** Check port 6379, this port is used by the Redis service. Check port 8087, the authentication service uses this port for communication. Check MongoDB ports 27017 - 27019, that also must be free on your machine. Take a look at these links to check busy ports in Windows and Linux

- **Docker services do not work in my Windows machine** Check that your docker environment is set to linux.

- **The ports are freed but services are not running**. Run `wipe.sh` followed by `launch.sh` script.

- **Other issues?** Drop us a line on Canvas discussion forum and we will help you.

5 Vulnerabilities

This section describes the classes of vulnerabilities that you are expected to find, exploit, and fix in *forumerly*. section 8 contains reference material that you may need to read to complete the tasks.

5.1 Regular Expression Denial of Service (ReDoS)

Regular expressions are widely used in today's web applications. For instance, the backend framework of a web application server may use regular expressions to route and catch URL parameters in regular GET requests. Regular expression denial of service (ReDoS) is a class of algorithmic complexity attacks, where matching a regular expression against an attacker-provided input takes unexpectedly long time.

The project may contain vulnerable application logic that is subject to ReDOS attacks. Whenever you find a ReDOS attack in the project, you should answer the following questions in the final report:

- Exploit: What is the actual attacker's input and why does it cause a ReDOS attack?
- Mitigation: How can you prevent the exploitation of the discovered ReDOS and why does it work?

5.2 Remote timing attack

In a remote timing attack, the attacker attempts to learn sensitive information by measuring the time it takes to perform secret-dependent computation in an remote (e.g., server-side) application. The attacker can then leverage these timing measurements to reconstruct the secret input. For instance, as you might have noticed in previous lab and lectures, secret-dependent branch conditions can provide such timing information.

If you find a timing attack in the project, answer the following questions in the report:

- Attack strategy. Why is the application vulnerable to remote timing attacks? How did you exploit this specific attack vector?
- Mitigation. How can you prevent the exploitation of the timing attack?

5.3 Cross Site Scripting (XSS)

Cross Site Scripting (XSS) attacks occur when an application takes untrusted data and sends it to a web browser without proper validation or sanitization, thus allowing attacker's malicious code to be executed on a victim's browser.

This project may contain several XSS attacks. You need to detect and exploit at least one. We recommend to explore the *forumerly* application and test it with input data that contains HTML tags and XSS vectors, identify uses of these tags, and examine how this input data is validated on the server-side. The documentation of ExpressJS and mustache.js can be helpful to figure out when an application escapes HTML in variables.

When you find an XSS in the project, answer the following questions in your report:

- How did you exploit the XSS? What requests did you need to send to the server?
- Does the server-side code contain any input data validation against XSS attack?
- What impact does this vulnerability have? What data can an attacker steal? If you aren't sure, you can always test possible attacks.
- How did you fix this vulnerability? Can you set up a web application to prevent the execution of any malicious script? How would you do it?

5.4 Remote Code Execution (RCE)

Remote Code Execution (RCE) is one of the most dangerous types of cyberattacks as it allows to remotely run malicious code within the target system on the local network or over the Internet. RCE can exploit various kinds of weaknesses in the code: buffer overflows, use-after-free, insecure deserialization, business logic, and software design flaws. *forumerly* contains an insecure deserialization vulnerability. Your goal is to detect and exploit the RCE thus getting a remote shell on the web server. Moreover, you should discuss mitigation methods against this vulnerability.

The blog post Deserialization Vulnerabilities: Attacking Deserialization in JS will help familiarizing with insecure deserialization vulnerabilities and exploits in JavaScript code. Then you can examine source code of *forumerly* against such vulnerabilities. The following questions will help you focus on right direction for finding the attack.

- What deserialization package is used in the project? Is it a vulnerable package?
- What request should an attacker send to execute the deserialization code?
- Should an attacker be authorized in the system to execute deserialization code?

You should prepare serialized data to execute arbitrary code during the deserialization process. For testing purposes, you can copy the deserialization code from the project to your own JavaScript file and run it separately by `node <file-name>.js` command. The function in `obj` that must be executed after deserialization may contain `console.log("TEST")` instead of actual malicious code for testing. You can use the following skeleton in your test:

```
const serializer = require(/*serialization package here*/);

var obj = {
  /*add an malicious function here*/
};

var serializedData = serializer.stringify(obj);

// if needed, you can replace some text of the payoad
var payload = serializedData.replace(/*replaced text*/, /*new text*/)

// printing the generated payload
console.log(payload)

// TESTING:
// copy an insecure deserialization code from the project
var newObj = serializer.parse(payload /*additional arguments?*/)

// copy usage of deserialized object newObj (method calls, etc) from the project
```

The next step is generating a Reverse Shell payload in order to run it on the web server and get full remote control of the server. The following recommendations can be helpful.

- Read the blog post Exploiting Node.js deserialization bug for Remote Code Execution for information on NodeJS reverse shell as well as an example of exploitation for another deserialization vulnerability. We highly recommend using the script in `tools/nodejsshell.py` to generate the JS code for Reverse Shell. This is an customized version of the script described in the blog post above.

- Run `nc -l <ip-address> <port>` on your computer to listen a port for incoming connections from Reverse Shell. If you use Windows, you can install Nmap and use `ncat -l <ip-address> <port>` instead of `nc`.

- Replace your test command (`console.log("TEST")`) from the payload generated in the previous step with the new payload containing Reverse Shell code.

The final step is a combination of all parts of the exploit and delivery of the payload to the web server. You need to prepare a request that contains the serialized data with Reverse Shell payload and get remote access on the server. The simplest way is to recognize the HTML form that sends the same request to the server. Take a look at the code of HTML form that sends serialized data and create your own HTML file that sends a similar request with your payload. You can read more information about HTML forms at these links (Link1, Link2).

5.5 Privilege Escalation via XSS

Privilege Escalation means getting privilege to access resources that should not be accessible. Malicious parties often use web attacks like XSS to gain basic access to certain resources and then perform privilege escalation attacks to gain control of other resources. The goal of this task is to get access to another user account on the forum, using one of the previous XSS vulnerabilities as the first step. You should also describe how to mitigate such attack.

The following questions should lead you towards finding an exploit:

- Take a look at XSS detected in subsection 5.3. What is the maximum impact you can achieve by exploiting this attack on the system?

- Can you steal a session cookie and use it to login as administrator? If so, prepare a demo of the attack. If not, explain why.

- What other attacks can you do against Session Management? Information about Session Fixation and the OWASP Testing Guide can be helpful.

- How can you check if a web application is vulnerable to Session Fixation attacks? Do it for *forumerly*.

- Can you set a session cookie from JavaScript in *forumerly*? Try experiments with cookies that have the same name, but different attributes. See more information about cookie attributes. Can you set cookies with the same name and different *path* or *domain* attributes? Which of the same-name cookies does the browser choose when sending a request to the server?

- Try to get another user privileges in *forumerly* by combining XSS and Session Fixation, and describe the results in the report. You can create a new user Bob and use another browser to emulate Bob's activities. If you get access to Bob's profile from another user account, it would enough to demonstrate the Privilege Escalation attack.

6 Additional Vulnerabilities

The original version of *forumerly* turns out to contain other vulnerabilities. We did not fix them, hence you can try to detect and describe these vulnerabilities in the report. You need to show a demo for each extra vulnerability and explain how to mitigate it.

Read about other popular client-side vulnerabilities, Path Traversal and Unrestricted File Upload, and try to use this knowledge for penetration testing of *forumerly*. We do not provide any hints for these vulnerabilities and it is as close to the wild as it gets. Act like a real hacker and get 2 extra points for each vulnerability found.

7 Requirements and Grading

7.1 Report

An zip file to be submitted via Canvas that should include the following:

1. Source code of your solutions.
2. A report describing how you approached the problem and the design of mitigation strategies. The report should contain, for each reported vulnerability, how you exploited it, and what mitigation strategy you implemented. Finally, the report should contain answers to questions inlined in specific tasks.
3. Clear statement of contributions of each group member.

7.2 Grading (max 25 points)

- You can get **5 points** for completing each task in section 5. Thus, you can get 25 points for the main part.
- To pass the lab you should be able to solve 2 tasks from section 5 and get 10 points.
- You can get **2 points** for every additional vulnerability in section 6, for a total of max 6 points.

Observe that the points above assume that the report is satisfactory and every group member is able to explain the solution and answer questions during the live presentation of the lab. Failure to do so may result in deduction of points. Moreover, any change to the third-party services is **NOT** accepted as a solution.

8 Reference material

- Best security practices in NodeJS
- Node bcrypt package isues
- Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers
- HashWick vulnerability
- Crypto library should have a constant-time equality function
- Using Node.js Event Loop for Timing Attacks