Optimizing and Vulnerability Testing of a Cloud-Based Intelligent Tutoring System

A thesis submitted in fulfilment of the requirements
for the degree of Master of Technology

by

Aditya Narhari Kadu
(19111006)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

June 2021

Certificate

It is certified that the work contained in this thesis entitled "**Optimizing and Vulnerability Testing of a Cloud-Based Intelligent Tutoring System**" by **Aditya Narhari Kadu** has been carried out under my supervision and that it has not been submitted elsewhere for a degree.

Prof. Amey Karkare
Associate Professor
Department of Computer Science and Engineering
Indian Institute of Technology Kanpur

June 2021

DECLARATION

This is to certify that the thesis titled **Optimizing and Vulnerability Testing of a Cloud-Based Intelligent Tutoring System** has been authored by me. It presents the research conducted by me under the supervision of **Prof. Amey Karkare**. To the best of my knowledge, it is an original work, both in terms of research content and narrative, and has not been submitted elsewhere, in part or in full, for a degree. Further, due credit has been attributed to the relevant state-of-the-art and collaborations (if any) with appropriate citations and acknowledgements, in line with established norms and practices.

Signature

Name: Aditya Narhari Kadu

Programme: MTech

Department: Computer Science and Engineering

Abstract

The advancement of web applications has been rapid; it has moved from just serving static information with text and images to platforms for learning, socializing, business etc. The increase in web application functionalities also increase the codebase size; the term "big code" is gaining traction lately. Big code is referred to the ever-expanding size and complexity of the codebase that the developers have to maintain. "Big code" matters because it increases the developer's efforts to maintain, update, and find security vulnerabilities in an application. More the lines of code in an application, the difficult it is to point out the source of a bug or ensure that a change in one aspect of the codebase does not cause a problem in another. The measure that can be taken to counter this problem is to eliminate the dead code, i.e. the code present in the codebase but is not used by any active component of the application. Another challenge web applications are facing these days is security breaches. Statistics shows that security breaches are rising every year. Most of the applications even contain common vulnerabilities listed in OWASP top 10; it lists the most critical web application security risks. Thus, it becomes essential to scan the application for vulnerabilities to avoid a potential security breachs.

Prutor is an Intelligent Tutoring System developed at the Indian Institute of Technology, Kanpur, by Mr Rajdeep Das under the guidance of professor Amey Karkare. It is used to teach introductory programming courses and collects data regarding how students solve programming problems for research purposes. It is a dockerised application with various technologies and frameworks used in the backend and frontend. It has been under active development since its initial release. Over time, some of the functionalities were depreciated or failed to deliver the expected outcome; this created a problem of dead code. A detailed analysis of used/unused code and vulnerability scan has not been done on Prutor.

In this thesis, we did a thorough analysis of used/unused code present in the codebase of Prutor. We analyzed by generating a code coverage report; this report highlights the unsued code present in the codebase. To generate the code coverage report for the backend of Prutor, we wrote a unit test suite using a popular javascript testing framework Jest. To generate the code coverage report for the frontend of Prutor, we wrote an end-to-end test suite using Jest and Puppeteer, a tool to control the Chrome browser over the devTools protocol. We have also generated a vulnerability scan report using the widely used web application vulnerability scanner OWASP ZAP. The vulnerability scan report identifies the security vulnerabilities present in the Prutor.

Acknowledgements

First and foremost, I would like to express my gratitude to my thesis supervisor, Professor Amey Karkare sir, for his precious guidance and support. This area was new for me, so it took me some time to get familiar with, and there were times I hit a dead-end and had to explore new paths, but through all this time, he was patient and let me work at my own pace. The weekly meetings were invaluable for the discussions we had and steering the project in the right direction. This project has given me a firm foundation in web application testing and vulnerability scanning, which will surely help me moving forward in my career.

I want to thank my parents and sister for providing me with constant support and encouragement throughout my years of study. I would also like to thank my friends; they stood by me in difficult times and motivated me whenever I felt low. Without them, this two-year journey would not have been so smooth.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Web development has progressed rapidly from just static web pages with just text and images to platforms for learning, socializing, business etc. This increase in web application functionalities has also increased the size of the codebase. A big codebase size opens various challenges like maintainability and security. One measure to tackle code maintainability is to eliminate the dead code [1] present in the codebase. A complete vulnerability scan can be carried out on the web applications to know the possible security breaches.

Prutor [2] is a web application [3] built to conduct programming lab sessions for an introductory programming course conducted at IITK. It is a dockerised [4] web application with containers, namely WebProxy, In-Memory Cache, Relational Database, NoSQL, Service Discovery, Webapp, Engine. Each container has a specified task assigned for the smooth functioning of the overall application. It was developed by Rajdeep Das [5] as his master's thesis. Over time many people have worked on it like Akshay Jindal [6], Somesh Rajoriya [7], etc., as their master's thesis and added features to make it more scalable and autonomous.

As mentioned above, many people have worked on Prutor. Over time some features were deprecated, or they failed to deliver the expected outcome; this lead to an accumulation of dead code in the codebase. A detailed code coverage analysis has not been done on Prutor. In this thesis, we have attempted to tackle this problem and eliminate the dead code from Prutor. We have used the standard testing tools [8], code coverage tools [9], and automation tools [10]. With these tools, we were able to locate and eliminate the dead code. We also have automated testing, so anyone with just a few commands can run the tests and generate the complete code coverage report. Along with this, we have done a vulnerability scan to discover the vulnerabilities present in Prutor.

1.1 Problem Statement

To eliminate the unused code present in the Prutor codebase and make it lean and mean by generating the code coverage reports and do a vulnerability scan on Prutor to identify the potential security breaches.

1.2 Motivation

As mentioned by Rajdeep Das in his thesis [5], the primary motivation behind Prutor's development was to abstract the factors such as programming environments, language-specific build commands from the process of problem-solving. It addresses those issues head-on by providing simple UI actions or keyboard shortcuts. It also provides a web-based editor interface where a user can compile, execute and evaluate the code, but as [1] mentions that many people have worked on Prutor. Some features were deprecated or failed to deliver the expected outcome. To make Prutor codebase maintainable it must be free from dead/unused code, and we must also know its vulnerabilities. So we need to work on the dead code elimination and vulnerability scanning aspects of Prutor. This thesis is a step in this direction.

1.3 Contributions of the thesis

The contributions of this thesis are a dead and unused code-free, lean and mean version of Prutor and a comprehensive vulnerability scan report on Prutor. It also demonstrates the use of testing tools like Jest [11] to write the unit and end-to-end test suites for a website, control headless Chrome browser [12] over the DevTools Protocol [13] by the use of Puppeteer [14], and to use the website vulnerability scanning tool OWASP ZAP [15].

It presents in detail the file system organization of the backend and frontend of Prutor; this helps to understand the comprehensive insights and working of Prutor, which is necessary to understand Prutor at routes [16] and query [17] level.

1.4 Organization of the thesis

This chapter declared the problem statement, expressed the motivation for this project and concluded with our contribution. The rest of the thesis is organised as follows:

In Chapter 2, we have briefly described the various components of Prutor from a developer's perspective. Then we defined the tools used for writing the unit and end-to-end test suite. Next, we have talked about OWASP ZAP, a tool used for vulnerability scanning. It finally ends with the description of previous work done on Prutor.

In Chapter 3, we have described the two main components of Prutor, namely Webapp and Engine, in detail. The knowledge of these components is critical in understanding the thesis completely.

In Chapter 4, we have discussed about all the aspects of writing unit test suite, end-to-end test suite.

In Chapter 5, we have discussed in detail about the vulnerability scan.

In Chapter 6, we presented the results of the decrease in the codebase of the webapp component of Prutor and have shown the findings from the vulnerability scan.

In Chapter 7, we have concluded our thesis work on Prutor and we have talked about more work that needs to be done to make Prutor more secure and future-ready.

Appendix A contains the step-by-step instructions on how to carry out vulnerability scan using OWASP ZAP. In the end, we have Appendix B, which includes projects links.

Chapter 2

Background, Tools and Previous work

This chapter starts with a short description of all the components of Prutor. It then talks about the tools that we have used for writing the test suite and vulnerability scan. It ends with a brief description of previous work that is done on Prutor.

2.1 Prutor

Prutor is a dockerised system [18]; i.e. every system component runs on a docker container [19]. Primarily the system is made up of 6 components. The subsequent subsections of this section describe each component of the system. Figure 2.1 shows the Prutor architecture.

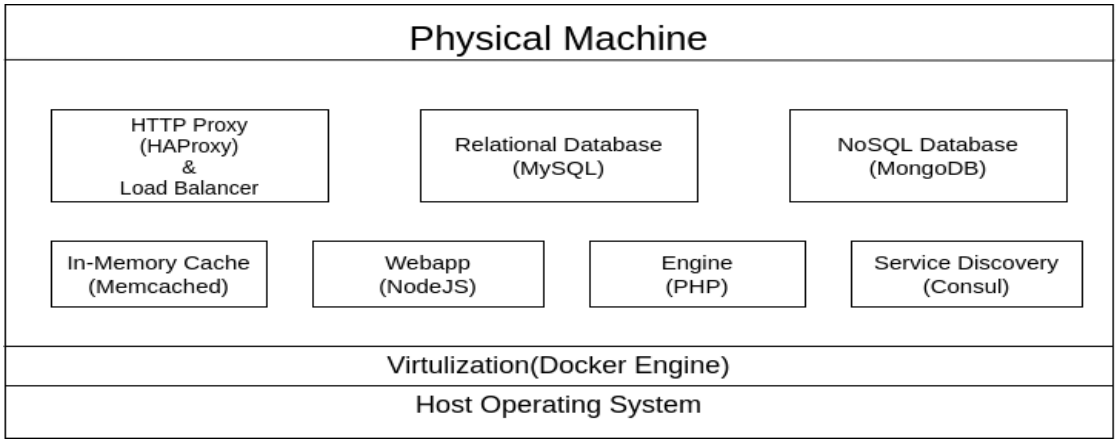

FIGURE 2.1: Prutor architecture

2.1.1 HAProxy

HAProxy handles all the HTTP requests and forwards them to the appropriate container; it also handles load balancing. It is the entry point of Prutor, i.e., all the HTTP requests first come here. It works at layer seven and filters these requests. Its configuration files contain the filtration logic. In Prutor, the filtration of HTTP requests is done based on URL. The requests of type /compile, /execute and /evaluate are carried out by the Engine (sec. 2.1.6) component, and the rest of the requests are handled by the WebApp (sec. 2.1.5) component. Figure 2.2 shows the role of HAProxy.

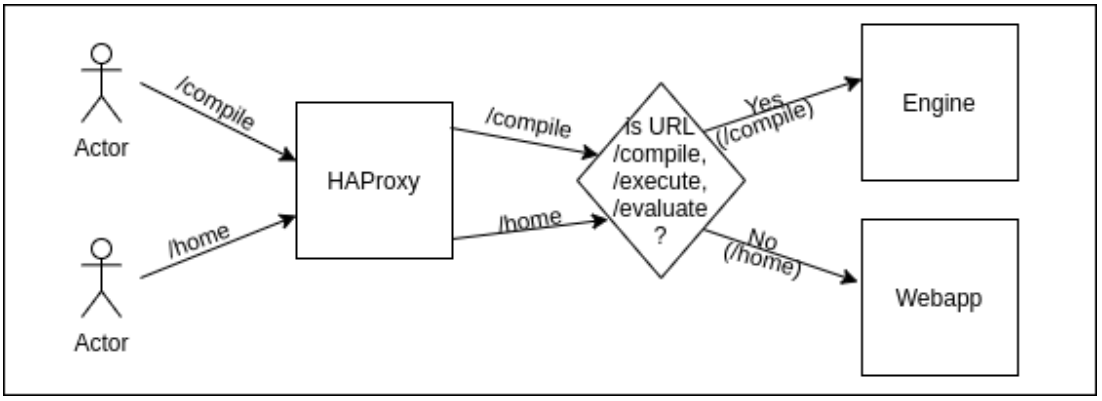

FIGURE 2.2: Role of HAProxy

2.1.2 Relational Database

The relational database carries all the data related to the users, like their login details, code they saved/submitted for events or practice, any support request they sent, etc. It also holds the data related to the problems like problem statement, test cases, sample solution, etc., and the data related to an event like its schedule, problems asked, type of event, etc. In Prutor, MYSQL [20] is used as a relational database.

2.1.3 NoSQL Database

NoSQL database is utilised for storing environment settings used by plugins and system data. Webapp and Engine components then use this information. MongoDB [21] is utilised to store this information.

2.1.4 In-Memory cache

The In-Memory cache holds all the data in memory in the form of key-value pair. In Prutor, it stores user sessions and also some database queries results. Memcached [22] is used as an In-Memory cache in Prutor.

2.1.5 Webapp

The webapp component handles the majority of HTTP requests; thus, it is one of the main components of Prutor. It mostly performs database operations which makes it an I/O intensive component. In the backend, it contains APIs for the interaction with the database. The backend is written in NodeJS [23] with Express [24] as the web framework. In the frontend it uses doT [25] templating library and JQuery [26] for interaction with the backend.

2.1.6 Engine

This component is one of the essential components of Prutor. The compilation, execution and evaluation requests of the user are handled by this component. These requests are managed by the Apache server. PHP [27] is used to write the code for this component. It first fetches the programming environment-specific parameters from the NoSQL database and then compile/execute/evaluate the code in a sandboxed [28] environment. The response is then saved in relevant tables of the relational database. Finally, the response is converted in a readable format, and a response is sent back to the user.

2.2 Tools used

In this section we have briefly discussed about the tools that are used in this thesis for writing the unit test suite, end-to-end test suite and vulnerability scanning.

2.2.1 Jest

Jest [11] is a testing library for javaScript; it is used for structuring, creating and running tests. It is available as an NPM [29] package. It is one of the most popular test runners

for javaScript, and it is the default test runner for React [30] projects. Some of the terms used in the context of Jest and test runners are discussed below.

- **Testing**: It is a method to check whether our code is doing what we expect it to do. The purpose of testing is to identify errors, gaps, or vulnerabilities present in our code. Testing is mainly divided into three categories unit testing [31], integration testing [32] and end-to-end testing [33].
- **Unit testing**: The level of software testing where individual component of codebase is tested in isolation. The component is the smallest testable part of the codebase; it has one or more input and a single output. So basically, in our case, the "component" is a single function. Testing in isolation means that if the current function at the test depends on the other function in any form, we mock/stub the other function to focus on testing the current function.
- **Integration testing**: Integration testing is testing with dependencies, for example, testing a function that calls a function. So the function we are testing depends on the results of another function; this is an integration test because we are testing more than just a single unit/function. Thus we test the integration of the feature into another feature.
- **End-to-End testing**: End-to-End testing validates the entire codebase along with the integration of external interfaces like a database. The purpose of end-to-end testing is to test the whole software for dependencies, data integrity, communication with other systems interfaces and databases; to exercise complete production like scenarios.
- **Mock/Stub**: The current function at test may depend on one or more functions. So to test the function in isolation, i.e. without calling other functions, we simulate/mimic the behaviour of other functions. Generally, we mock the database calls so that we do not have to call the actual database.

2.2.2 Chrome code coverage

The Chrome code coverage [34] is a tab present in the Chrome DevTools. It can help find the unused JavaScript and CSS code of the web application the Chrome browser is currently accessing. In this project, it is mainly used to explore and experiment with Prutor, like checking which DOM event triggers which part of the codebase and then with this knowledge, we wrote end-to-end tests for that particular DOM event.

2.2.3 Puppeteer

Puppeteer [14] is a Node.js library that contains APIs to control Chrome or Chromium browsers over the DevTools Protocol. It can be configured to run in a headless or non-headless mode in the browser. As it provides control over the Chrome and Chromium browsers, we can do anything and everything that we manually do with these browsers using Puppeteer. A few of the applications of Puppeteer as mentioned on its website [14] are listed below:

- We can generate the screenshots and PDFs of pages.
- Automate the tasks like form submissions, keyboard inputs.
- We can also automate the task of end-to-end testing and create a complete testing environment. Run our test suite directly in the latest version of the Chrome browser.
- Crawl the Single-Page applications [35] and generate the Server-Side rendering [36].

In this thesis, Puppeteer is used with Jest for controlling the Chrome browser and perform actions like form submission, create problems, create events, write the dummy codes etc., on Prutor to write the end-to-end tests.

2.2.4 OWASP ZAP

The Open Web Application Security Project (OWASP) is an open-source community with the aim of making web applications more secure. The OWASP Zed attack proxy (ZAP) is a popular web application security testing tool. The OWASP ZAP tool can be used by the web application developers as well as during the penetration tests [37] to know about the web application's vulnerabilities. It is a Java [38] based tool; thus, it requires Java for its operation. It has a simple and intuitive graphical interface that allows web application vulnerability testers to perform actions like spidering, scanning, proxying etc. Some of the advantages of OWASP ZAP are:

- It is open-source and free software.
- It is straightforward to use and beginner-friendly.
- Zap supports cross-platforms, i.e. it works across all OS like Linux, Windows, Mac.

- It can generate detailed vulnerability scan reports in various formats like HTML, XML [39], Markdown [40], JSON [41].

In this thesis we have used OWASP ZAP for proxying our end-to-end tests through the OWASP ZAP server to perform an active scan and generate the HTML vulnerability scan report for Prutor.

2.3 Previous Work

Prutor was developed in-house at IIT Kanpur by Mr Rajdeep Das as his Master's thesis. Over time many people have worked on it, We are mentioning a couple of them here. Scaling any application is very crucial to mitigate the increasing demand of more and more users. Mr Akshay Jindal, a former student of IIT Kanpur, has done this work of scaling the Prutor. He analysed Prutor in his Master's thesis [6], and successfully achieved a scale of 3000 and 1000 users on the Webapp and Engine components, respectively, on an 8-core machine with Core i7 3.40 GHz processor. Next, another former student of IIT Kanpur, Mr Somesh Rajoriya [7], used Kubernetes [42] technology on Bare Metal server [43] to manage all the components of Prutor and provide the functionalities like auto-scaling, storage orchestration and load balancing.

Chapter 3

Analysis of Components of Prutor

Prutor has a large codebase with many different technologies, like Node.js and Express for the backend, PHP used in the engine component, doT templating and JQuery for the frontend, etc. On top of that, it is an entirely dockerised system, so it can be a bit overwhelming to grasp the codebase of Prutor completely. This chapter is an attempt to make this process of understanding the codebase a little easy easy. It discusses the codebase of two main components, namely Webapp and Engine.

3.1 Webapp

The webapp component has a vast codebase with most of the code for the backend and frontend of Prutor. We will first discuss the backend codebase and file structure, then the frontend codebase and file structure.

3.1.1 Webapp Backend

The important files and folders related to the backend codebase are the *app.js* file, *routes* folder, *app_modules* folder and the *bootstrap* folder. The main file in the webapp backend is app.js; it contains NoSQL database connection, view engine setup for doT templating library, globally available variable definitions, route handling, error handling etc. The figure 3.1 shows the backend routes flow and table 3.1 shows definitions of the global variables.

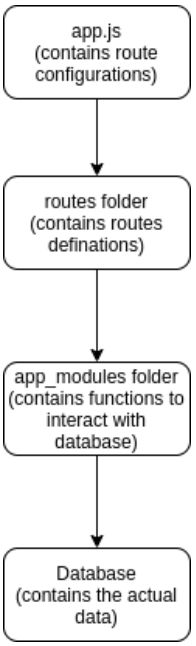

FIGURE 3.1: Backend flow

- The app.js file contains configurations of different routes, and these routes are defined in the routes folder. The routes folder contains all the definitions of different (get/post) routes. Many of these routes require interaction with the database for authentication, accessing the user data, modifying the user data, creating entries in the table, etc. All this logic of database interaction, validation etc., is defined in the app_modules folder. The app_modules folder contains the logic for interaction with the database tables. So the complete flow is from app.js file to routes folder to app_modules folder to database.

- There are some global variables defined in app.js file. These are variables for SQL database, NoSQL database, cache etc. These variables are mostly used in the app_modules folder. The definition of these variables is present in the bootstrap folder. There are four such variables, namely "sqlpool", "sqlquery", "sqlcache", "nosql".

Variable	Defined in file	Usage
sqlpool	bootstrap/database.js	connection to SQL database
sqlquery	bootstrap/database.js	executing the SQL query
sqlcache	bootstrap/cache.js	handles caching logic
nosql	mongoose instance	handles noSQL database

TABLE 3.1: Global variables.

3.1.2 Webapp Frontend

The essential files and folders related to the webapp frontend are the *views folder* and the *public folder*. Below is the discussion about the views folder then the public folder.

- The views folder contains the ".dot" files; these are the files of the doT templating library. Most of these files contain a variable "controller" in the script tag, which is used to load the appropriate javascript files from the public folder. The _admin files also contain the "_adminMode" variable it signifies that the "admin" account can only access the page. There are some other variables as well in these files that have a specific purpose.
- The *public folder* contains different folders like *vendor*, *styles*, *scripts*. The purpose of them is described below one by one.
  - The vendor folder contains all the different modules used in the frontend like "jquery", "requirejs", "socket.io" etc. These modules are then used throughout the frontend.
  - The styles folder contains the CSS styles applied on the user interface, the "bootstrap" module present in the vendor folder is also used for styling the pages.
  - The scripts folder contains the actual code for DOM [44] manipulations. The entry point or starting file of the scripts folder is the *main.js* file. The main.js file imports/requires various modules from the vendor folder, and it also contains the code for CSS imports. One of the vital tasks of the main.js file is to load the appropriate file from the scripts folder based on the value of controller and _adminMode variables. These loaded files contain the DOM manipulations, event listeners [45] and routes to send and receive data from the backend.

So to summarise the working of the frontend code, the views folder contains doT templating files. These files contain two important variables controller and _adminMode. These two variables are important because they connect the views folder and public/scripts folder. The scripts folder contains the actual code for DOM manipulation, event listeners and routing to communicate with the backend. The entry point of the scripts folder is the main.js file; it loads the helper modules and CSS files. The main.js file then loads the appropriate files in the scripts folder based on controller and _adminMode variables. The public/vendor folder contains the modules that are used in the public/script folder. The public/styles contain the CSS files for styling the user interface.

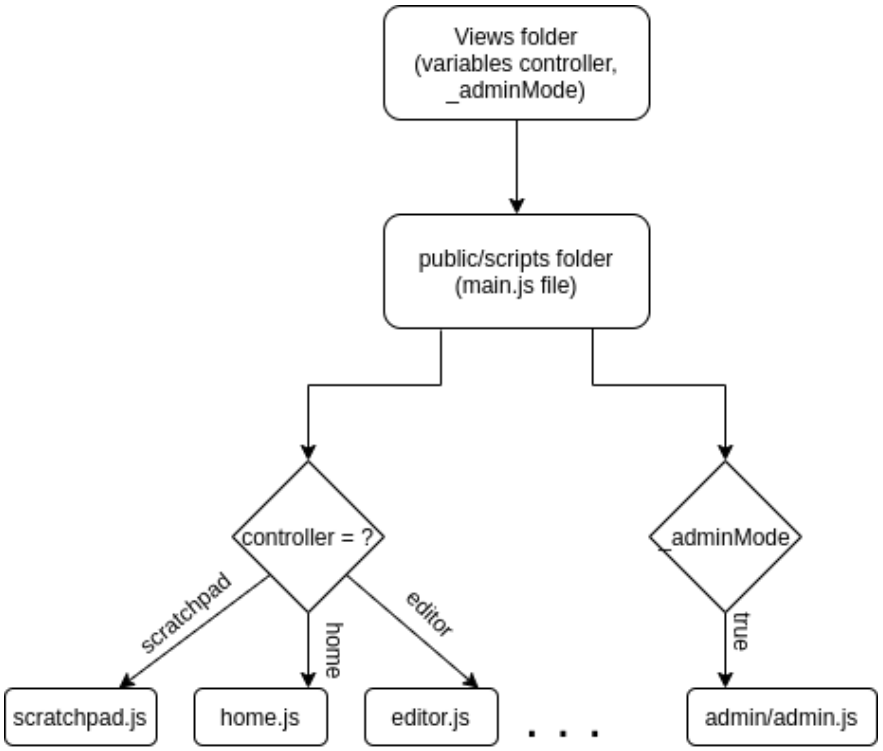

FIGURE 3.2: Frontend flow

3.2 Engine

All the files related to the engine component is present in the *its-engine repository* [46]. When we run the deploy script from the Prutor repository to deploy Prutor on our system, all the "engine" component files get stored in the *codebase/engine* folder. Let us try to understand the file structure of the engine component.

1. The entry point in the engine component is the *index.php* file. This file first loads/imports/requires the *core/bootstrap.php* [item:2] file. It then gets all parameters of the request with the help of the function "getURIParameters, this function is present in the *core/common.php* [item:3] file. The index.php file then separates the parameters that it got from the getURIParameters function into "action" and "uriParams". The action variable contains the main command that the user wants to perform like compile, execute, evaluate, etc. The uriParams variable contains the additional parameters that the request has along with the main command. Finally, the index.php file passes the action and uriParams to the "resolve" function, this function is present in the *core/resolver.php* [item:4] file.

2. The *bootstrap.php* file first loads the environment variables like NOSQL_USER, DB_PORT, DB_USER, ENGINE_LANG_TYPE etc., from the */etc/environment* file into the super global variable "$_SERVER" [47]. The /etc/environment file can be found in the engine docker container. Next, the bootstrap.php file loads *core/-globals.php* [item:5], *core/autoload.php* [item:6], *core/common.php* [item:3], *core/resolver.php* [item:4] files present in the core folder of the its-engine repository. It then also loads the *vendor/autoload.php* file. Next, with the help of PHP's in-built function "spl_autoload_register" [48], the class autoloading logic is implemented so that there is no need to write the class include statements in any other file of the entire engine codebase. Next, the bootstrap.php file has the functions "on_error, "on_shutdown", "prutorlog; these function definitions are used to register the shutdown and error handling logic in the production environment.

3. The *common.php* file is present in the core folder. This file contains two functions "getURIParameters", "generate_random_string". The getURIParameters function returns all the parameters passed with URI [49] request like compile/execute/evaluate and any other additional parameters. The generate_random_string function generate and return random string from the pool of "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789" characters with a default length of a string as 25.

4. The *resolver.php* file is present in the *core* folder. It contains two functions "resolve", "call_method". The resolve function takes "action", "uriParams" as parameters. The action variable has the values like compile/execute/evaluate; thus, it holds the value of the main action requested by URI. The uriParams contains the other additional parameters for the main action. Next, it gets all the configurations for the given value of the action variable from the "lookupPort" function of the *Registry class* present in the *system* folder. The configurations include controller class name present in the *app/controllers* folder, the method name of the particular controller class and port type 'PUBLIC' or 'PRIVATE'. The function then creates the object of the particular controller class. Finally, it passes as parameters controller class object, a method name of that particular controller class and additional parameters of the requested URI to the call_method.

5. The *globals.php* defines global constants like "PATH_SYSTEM", "PATH_MODULES", "PATH_CONTROLLERS" etc. these constants are used through the codebase.

6. The *autoload.php* file contains the functions "load_module", "load_system", "load_redbean" etc. these functions are than used in "spl_autoload_register" function of core/bootstrap.php file as described above [item:2].

7. Point 4 mentions that function resolve creates an object of the controller class. These controller class files are present in the *controllers* folder, the one that contains methods of compile, execute and evaluate is the *MainController.php* file. The methods "compile()", "execute()" and "evaluate()" of MainController.php first, check the access permission whether or not the user is allowed to access the resources or not, with the help of method "checkAccess()" present in the same file. Next, based on the conditions, the variables "scratch", "code" and "env" are initialized. Next, an object of class Tutor [item:8] is created, based on the user's role like "admin", "students" and whether or not the "assignment_id" is present or not the suitable method from the Tutor class is invoked. The return values from the Tutor class method are first encoded to the JSON format and then passed on to the webapp frontend code to respond to the request sent to the engine container.

8. The *Tutor* class present in the *app/modules/Tutor.php* file contain methods to customise the compile/execute/evaluate requests and some helper methods to accompany these methods. These methods first load various configurations of delays, environments etc., from the NoSQL database container, i.e. the "its" database of MongoDB. It then collects some information like "assignment_id", "user_id", finally creates the instance of *Engine class* [item:9] and calls appropriate methods from the "Engine class" for actual compilation, execution and evaluation. After the result arrives from the "Engine class", it saves the code, does logging, and sends back the result to the "MainController" file.

9. The *Engine* class present in the *app/modules/Engine.php* file contains the actual compile and execute commands to run the user's request. It contains the methods like "compile", "execute", "interpret", etc. Let us examine the compile method the other methods have similar structure. Firstly, the compile method gets all the configurations like command to compile, execute, output format etc. from the "environments" collection of the "its" NoSQL database. It then stores the command, source file info it got from the environments collection into variables. Next, stores the user's code in the data folder, it then goes inside the *data* folder. Finally, executes the command using the PHP "proc_open" [50] function. Next, it stores the output and errors (if any) into variables "buffer", "errors", closes the process by the PHP

function "proc_close" [51]. Next, it stores all the outputs, errors and other relevant info into the variable "result" and returns.

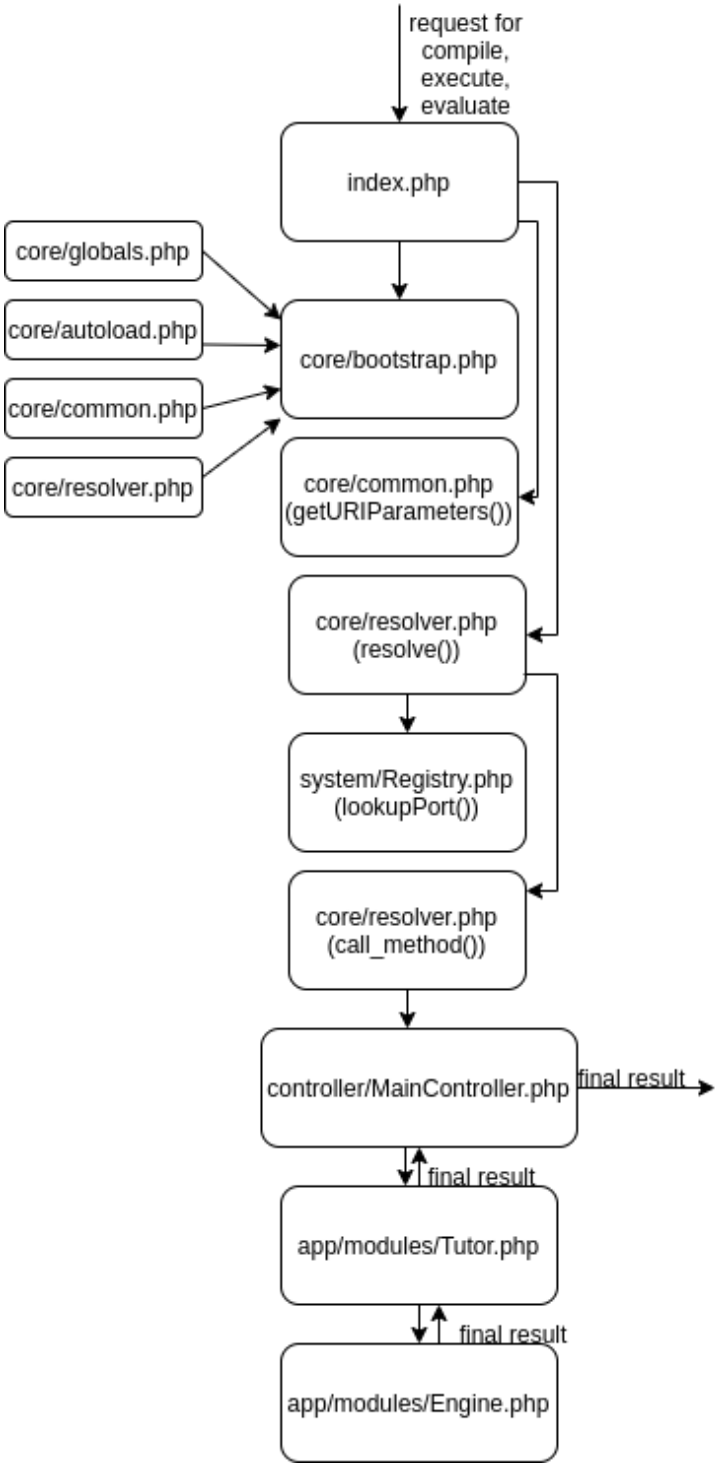

FIGURE 3.3: Engine flow

Chapter 4

Writing the Tests

In this chapter, we have talked about the unit test suite and end-to-end test suite in detail. It starts with a section on the unit test suite wherein we discuss some definitions required to understand the unit tests, line by line explanation of a typical unit test and finally discuss the file structure and nomenclature of the unit test suite. Next, we have a section on the end-to-end test suite. Here we discuss our failed attempts while trying to generate the frontend coverage, some concepts required to understand the end-to-end test suite, folder structure of the end-to-end test suite. Then we have a line by line explanation of a typical end-to-end test, interpretation of the coverage report generated after running the test suite. Finally, it ends with some drawbacks of the end-to-end test suite.

4.1 Unit test suite

We first attempted to write unit tests for the backend code of the webapp component of Prutor. When we searched for ways to get the backend code coverage, we came across the "Jest" framework. Jest is an open-source javascript testing framework, and Facebook [52] developed it to test their codebase. It supports code coverage out of the box; all we have to do is use the "–coverage" flag to get the code coverage for our test suite, hence our choice to write our tests.

4.1.1 Defination

There are few terms used in Jest testing (and testing in general). We have mentioned a few in chapter 2, but let us discuss them here because they are essential for understanding the unit test suite.

- **Matchers**: A matcher or assertion is a function that is used to check for specific values or conditions; commonly, they are used for comparison. Let us take a small example to understand them better.

  expect(2*2).toBe(4)

  The "expect" function returns an object in the above example, and on that object, we call **.toBe** matcher. Depending on the actual value in the expect function and value in the matcher (in this case .toBe matcher), the assertion can fail or pass. When Jest runs the test, it tracks all the matchers/assertions and gives a final message of failed and passed matchers. There is an extensive list of available matchers; the complete list can be found in Jest documentation [53]. In table 4.1, we have mentioned the ones that are commonly used.

Matcher	Functionality
.toHaveBeenCalled()	Check if the function is getting called.
.toHaveBeenCalledTimes(number)	Check if the function is getting called a given "number" of times.
.not	Used to negate the assertion made by the matcher.
.toHaveReturned()	Check if the function successfully returned.
.toHaveReturnedTimes(number)	Check if the function successfully returned the given "number" of times.
.toBeFalsy()	Check if the value is false in the boolean context. Falsy values are false, 0, '', null, undefined, NaN.
.toBeTruthy()	Check if the value is true in the boolean context.
.toBeNull()	Check if the value is null.

TABLE 4.1: Matchers in Jest

- **Mocking**: Mocking is a technique to isolate the component we are testing by replacing its dependencies on other components with mock/dummy components. The component can be a function or a feature. The dependencies can be anything our component depends on; it can be another module or component or a database calls etc. In Jest, mocking means replacing dependencies with a mock function. The mock functions provide the functionality to capture calls, set return values, change the implementation. Table 4.2 shows three main types of mocking in Jest.

Mock type	Usage
jest.fn	Mock a function
jest.mock	Mock a module
jest.spyOn	Spy or mock a function

TABLE 4.2: Types of mocks

- **describe block**: The describe block groups together similar types of tests. All the tests in one describe block run sequentially. We can even have nested describe blocks, but the use is discouraged as they add unnecessary complexity to the test suite. Different constructs in the describe block are used in writing the tests; they are defined in table 4.3.

Construct name	Definition
beforeAll	Do some pre-processing before any of the tests starts.
afterAll	Do some post-processing after all the tests finished.
beforeEach	Do some pre-processing before each of the tests run.
afterEach	Do some post-processing after a test has finished.
test OR it	We write our actual test here with the help of matchers and assertions.

TABLE 4.3: Types of test constructs

4.1.2 A typical unit test file rundown

Let us now go through unit test file. Figure 4.1 shows the typical unit test file.

1. Initially, we require/import the file from the app_modules folder present inside the webapp folder. We test a function from this imported file.

2. The "sqlquery" is a database call, so we mock this call to isolate the function we are testing from external dependencies.

3. The function we are testing has some formal parameters; we declare these parameters in the "beforeAll" block. As these parameters are present in the beforeAll block, they will be available in all the tests we write in this file.

4. In the "afterEach" block we clear the context of sqlquery mock function with the "mockClear()" function. The mockClear() function is used to clean up the context between two assertions or tests.

5. In the "describe" block, we write our actual tests. The first test is to ensure the function we are testing is imported or not. In the next test, we call the function we are

```
const accounts = require('../../../app_modules/accounts');

sqlquery = jest
.fn()
.mockImplementationOnce((query, [], cb)=>cb(true, null))

beforeAll(()=>{
    csvData='fk@fake.com,fake name2,2,456\n,fake name1,1,123\nfk@fake.com,fake name3,3,789';
    role=1;
    authType=0;
    type='STUDENT';
});

afterEach(()=>{
    sqlquery.mockClear();
});

describe('Test Suite for app_modules/accounts.js/createUserList', ()=>{
    test('Existence test', ()=> {
        expect(typeof accounts.createUserList).toBe('function');
    });
    test('single test to cover complete code', done=>{
        accounts.createUserList(csvData,role,authType,function(err, res){
            expect(sqlquery).toHaveBeenCalledTimes(1);
            console.log('err', err);
            console.log('res', res);
            done();
        });
    });
});
```

FIGURE 4.1: A typical unit test file

testing. This function uses some variables that we have defined in the beforeAll block. The function has a sqlquery database call, but as we have mocked the implementation of sqlquery, the mocked version will be called instead of the actual database call. We "expect" that sqlquery is called one time using the *toHaveBeenCalledTimes* matcher and "console.log()" the response or any error occurred. The "done" is used because the functions we are testing have a callback function.

4.1.3 File structure and Nomenclature

All the tests are inside the "test/unit" folder. There are multiple folders in the unit folder; each folder corresponds to a file in the app_modules folder. The files inside these folders are the functions of the files in the app_modules folder. The naming convention used for these files is "app_modules.$\langle file\_name \rangle$.$\langle functionInsideThatFile \rangle$.test.js". The figure 4.2 shows the file structure and nomenclature in a better way.

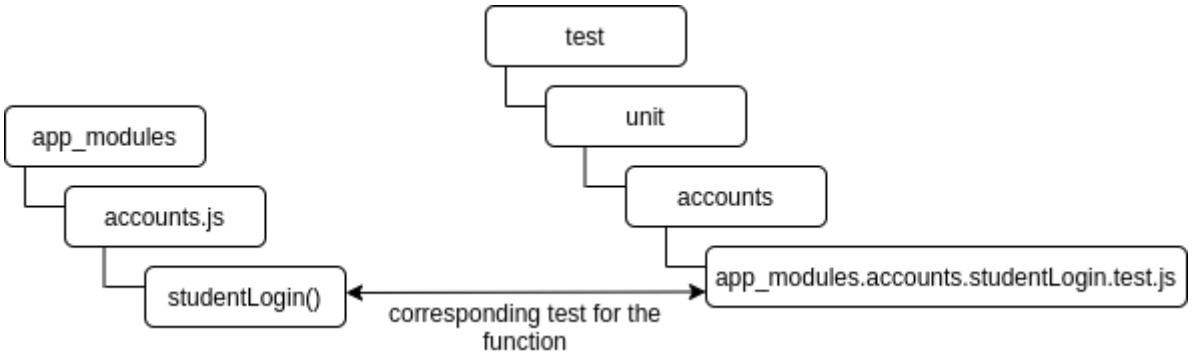

FIGURE 4.2: unit tests file structure

4.2 End-to-end tests

Once the backend unit test suite was done, we shifted our focus to the frontend codebase. While searching for the tools for flagging the dead code in the frontend, we tried various tools like Jalangi [54], Tree Shaking [55], google closure compiler [56] but nothing seemed to work the way we wanted. Finally, we came across Chrome code coverage and Puppeteer with Jest. This combination was giving us the expected result we wanted. With the Chrome code coverage, we could check which DOM event triggers what part of the codebase. With this knowledge, using Puppeteer and Jest, we wrote the tests.

4.2.1 Falied attempts

Let us now discuss our failed attempts to flag the dead code present in the frontend codebase.

- **Jalangi** [54]: It is a selective record-replay and dynamic analysis framework for Javascript. The issue that we faced with Jalangi was that there is not much documentation and community help present to guide us properly on how to use it correctly. Also, it has no active development for the last three years. So, when we tried it on Prutor, there were many errors and warnings and no help to resolve them correctly. This led us to drop the use of Jalangi.
- **Tree Shaking** [55]: Tree Shaking in Javascript generally means eliminating the dead code present in the codebase. The problem we faced was the implementation of tree shaking relies on the static structure of ES2015 module syntax, i.e. import and export. However, the code of Prutor relies on CommonJS require and exports

structure to interact with modules. So, in the current state, Tree Shaking with webpack cannot be applied on Prutor.

- **Google closure compiler** [56]: It is a powerful tool that can do dependency management [57], simple code optimizations like white space removing, local scope names, minification and advanced optimizations. All these tools help in removing the dead code and deliver the absolute minimum code that is required. However, these optimizations do not come for free, we need to tell the closure compiler the intent of our code, and the way this is done is through JSDoc comments [58]. There are JSDoc comments written for every function in the backend codebase, but for the frontend codebase, they are not written for every function. Using the closure compiler's advanced optimization without specifying the intent of the code gives unexpected results like removing the code that is used. So currently, using the google closure compiler for the frontend of Prutor is not feasible.

- **Chrome code coverage** [34]: The coverage tab in the Chrome DevTools highlights the CSS and Javascript code currently being used. We need to be careful while using this tool because some code is used only when a particular DOM event triggers like click event, hover event etc. So we need to examine the currently used webpage properly and fire all the DOM events present on it to get the complete code coverage analysis. Secondly, we cannot export the used code to any file; it is only displayed on the coverage tab. Thus using it alone for flagging the dead code had issues like we had to check the used/unused code manually; there is no automation to get the code coverage any time we want. We did use the Chrome code coverage tab to get all the selectors that triggered the DOM events, and based on this knowledge; we wrote our tests with the help of Puppeteer and Jest.

4.2.2 Concepts of Puppeteer

We have explained Puppeteer in section 2.2.3. In short, Puppeteer is a Node library that provides a high-level API to control Chrome or Chromium over the DevTools protocol. Let us now understand some of the concepts related to Puppeteer.

- **Browser**: When we install the standard Puppeteer installation, it also downloads the Chrome browser for us. This is done so that we do not rely on any external dependency; everything is downloaded by standard Puppeteer installation. Invoking the browser is as simple as calling the method "puppeteer.launch()". We can launch

the browser in headful and headless mode by giving the value of true or false to the "headless" parameter. We can also control the speed of the browser with the "slowMo" parameter. We can also pass some args like ignore the certificate errors, start the browser in full screen etc. There are many such options with which we can customize our browser launch. The complete list can be found on page [59]. Closing the browser instance is done by calling the method "browser.close()".

- **Page**: Once we have launched the browser by invoking the puppeteer.launch() method, we can create a new browser page instance or a new tab in this browser by invoking the "browser.newPage()" method. This method will return a page object, and with the help of this page object, we can go to some URL, we can control the webpage we visited with the help of CSS selectors [60]. We can do much more with the page object. Table 4.4 shows some of the essential methods; complete list can be found on the page [61].

Method name	Usage
page.goto(url[, options])	Navigate to the given 'url'
page.waitForSelector(selector[, options])	Puppeteer will wait until the 'selector' appears on the page.
page.waitForTimeout(milliseconds)	Puppeteer will wait for given 'milliseconds'.
page.goBack([options])	Puppeteer will press the 'back' button present on the browser.
page.goForward([options])	Puppeteer will press the 'froward' button present on the browser.
page.evaluate(pageFunction[, ...args])	Evaluate the given 'pageFunction' in the page Context.
page.setViewport(viewport)	Sets the view size of the page.
page.click(selector[, options])	click on the particular 'selector'.

TABLE 4.4: page object important methods

4.2.3 Jest global setup

We have used Jest testing framework for writing the end-to-end tests. The goal was to generate the code coverage report, so to achieve that, we had to run all the tests in same browser and page context; otherwise, the coverage report was not what was expected. To achieve this, we set up a global browser object and a global page object. This single global browser object and page object is used to run the tests. Let us now discuss the way to define these global objects step by step.

- In Jest the global objects are defined through the *jest.config.js* file. This file contain various options to configure Jest; a complete list can be found on page [62]. The options that are important to us are "testEnvironment", "globalSetup and "globalTeardown".

- The testEnvironment configuration is specified in the *puppeteer_environment.js* file. The Jest configuration documentation [63] states that a file used for testEnvironment configuration should export a class with methods "setup", "teardown", and "runScript".

- In the puppeteer_environment.js file , we can also define global variables with the "this.global" object. These variables will be present in all the tests as global variables. So in the "setup" method of this file, we have defined the global variables "URL", "CWD", "page", and "__BROWSER__". The URL variable holds url on which Prutor is running, CWD holds current working directory. The page variable holds the puppeteer page object and __BROWSER__ variable holds the puppeteer browser object. In this file We have also written the code to instantiate the code coverage.

- In the "teardown" method, we have the code to stop the code coverage. The "runScript" is same as the default method.

- The "globalSetup" configuration allows the use of a module that exports an async function. This function is triggered once before all the test suites. We have defined our globalSetup configuration in the *setup.js* file. In this file, we have instantiated the browser to start.

- The "globalTeardown" configuration allows the use of a module that exports an async function. This function is triggered once after all the test suites. We have defined our globalTeardown configuration in the *teardown.js* file. In this file, we have closed the browser instance.

4.2.4 Folder structure

Let us now discuss different folders and their use in the test suite one by one. These folder are namely *tests*, *scripts*, *pages*, *data*, *.nyc_output*, *coverage*. Figure 4.3 shows the files structure of the test suite.

- **tests folder**: This is the folder that contains all the tests. We run the files inside this folder to run all our tests. These tests are written for the files inside the "public/scripts" folder of the Webapp component. Let us now take the *admin.account.test.js* file as an example to understand the naming convention. The *account.js* file is present inside the *public/scripts/admin* folder. So, first, we write the file's folder name and then write the file name. Thus the convention is: $\langle folderName \rangle.\langle fileNameInsideThatFolder \rangle$.test.js.

- **scripts folder**: Many of our tests requires some pre-populated data in the database to run. The shell scripts present in this folder are used to pre-populate the database as well as delete the data after the tests are finished running.

- **pages folder**: Many of the tests have some code in common, like the login, logout code; almost all the tests use them. So to avoid the repetition of such common code, we have created some classes, and these classes contain methods that have code to handle these common functionalities. The pages folder contains the files of these classes.

- **data folder**: This folder contains C language programs used in the tests. These program files are not used directly in the tests; they are just for reference. It also contains the *test.csv* file; this file contains base64 encoded strings "3" and "Hello World!", it is used to add an automated test case for the test *admin.createProblem.test.js*. There is one more folder *accounts*, this folder contains the 'CSV' files that have info to create different account types. It is used in the test *admin.accounts.test.js*.

- **.nyc_output folder**: This is the folder where all the data is collected for generating the code coverage report while the test suite is running. We need to delete this folder before running the test suite so that a new .nyc_output folder is generated for the test suite we are currently running.

- **coverage folder**: This folder contains the actual HTML code coverage reports that we use to see the used/unused code. This folder is also generated for us after the test suite is complete. The HTML reports in this folder are generated from the data collected from the .nyc_output folder. We need to delete this folder before running the test suite to generate a new one for the currently running test suite.

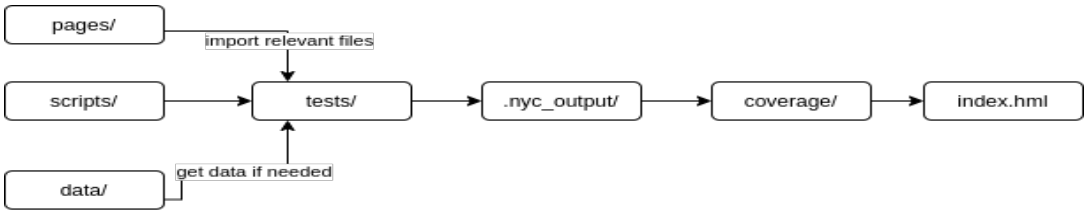

FIGURE 4.3: End-to-end tests flow.

4.2.5 A typical end-to-end test rundown

Figure 4.4 shows a typical end-to-end test suite file. Let us go through this file line by line to get more familiar with the end-to-and test writing.

```
1  import LoginPage from "../pages/LoginPage";
2  import BasePage from "../pages/BasePage";
3
4  describe("Test suite for admin/tasks page", () => {
5      const loginPage = new LoginPage(),
6          basePage = new BasePage();
7
8      beforeAll(async () => {
9          // Login to the Prutor.
10         await loginPage.visit(URL);
11         await loginPage.isLoginDisplayed(["input[type='login']", "input[type='password']", "#btn-submit"]);
12         await loginPage.loginCred(
13
14         // Goto admin tasks page.
15         await basePage.waitAndClick("#navbar-admin > ul > li:nth-child(2) > a");
16     });
17     afterAll(async () => {
18         await loginPage.logoutAdmin();
19     });
20     afterEach(async () => {
21         await page.waitForTimeout(500);
22     });
23     test("Click on events in menu bar", async () => {
24         await basePage.waitAndClick("#home");
25         await basePage.waitAndClick("#events");
26     });
27     test("Check the assigned task", async () => {
28         await basePage.waitAndClick("#menu > li:nth-child(2) > a");
29         await basePage.waitAndClick("#events:nth-child(1)");
30         await basePage.waitAndClick("#taskStatus > div:nth-child(2) > ul > li > i");
31         await page.waitForTimeout(500);
32         await basePage.waitAndClick("#taskStatus > div.tree.jstree > ul > li:nth-child(1) > ul > li:nth-child(1) > a");
33         const tabs = await global.__BROWSER__.pages();
34         await tabs[tabs.length - 1].close();
35         await page.waitForTimeout(1500);
36     });
37 });
```

FIGURE 4.4: A typical End-to-end test.

- In line 1 and 2, we import the classes *LoginPage* and *BasePage* from the *pages* folder.
- From line 4, we start our describe block. In this block, we create the objects of the classes we imported, then define the beforeAll, afterAll, afterEach functions, and finally write our tests one after another.

- In line 5 and 6, we create the objects of the classes we imported. In this example, the object of classes LoginPage and BasePage are created.
- In line 8 to 16, beforeAll function is defined. This function runs once before the first test runs. In this function, we first login to the Prutor and then go to the page where we want to run this test. In this example, the *admin tasks* page.
- In line 17 to 19, afterAll function is defined. This function runs once after the last test is finished. In this function, we generally perform the database cleanup and logout of the Prutor.
- In line 20 to 22, afterEach function is defined. This function runs after each test is finished. In this function, we wait for some milliseconds before starting the next test to synchronise the tests properly.
- Line 23 to 26 is our first test, and line 27 to 36 is our second test. In these tests, we have used the combination of methods of classes we have imported and the methods provided by the Puppeteer's *page* object.

4.2.6 Code coverage report explanation

Finally, let us discuss the HTML code coverage report. Figure 4.5 shows the typical report file. This report is generated as the final step after our tests are finished running. The red highlights indicate that the lines not covered/used in our test suite and the non lighted line are covered in our test suite. There can be a situation where the lines are lighted as red i.e. uncovered even though they are covered, such type of situations are discussed in the drawbacks section 4.2.7 below. So we need to do a final manual inspections of this report to make sure we not end up removing the used lines of code.

4.2.7 Drawbacks of end-to-end test suite

There are some of downsides or drawbacks of this test suite, they are discussed below:

- The tests we run are flaky [64]; the end-to-end tests are flaky by nature because they depend on many components like frontend, backend, database, etc. The failure of any one of these components can cause the end-to-end test to fail. We have taken utmost care that our tests are not flaky, but there can still be a situation where this

```
define(['bootstrap'], function() {

  load_css('/styles/theme');

  $(document).ready(function() {
    $('#btn-pass').click(setPassword);
  });
});

function setPassword() {
  var pass1 = $('#pass1').val().trim();
  var pass2 = $('#pass2').val().trim();
  var path = window.location.pathname.split("/");
  var hash = path[path.length - 1];
  if(pass1 === '') {
    toastr.warning('Password cannot be empty!');
    return;
  } else if(pass1 !== pass2) {
    toastr.warning('Entered passwords do not match!');
    return;
  }
  $.post('/accounts/setnewpassword', {hash: hash, password: pass1}, function(response) {
    if(response === true) {
      toastr.success('Password has been set! Login to the ITS.');
      window.location.href = '/';
    } else
      toastr.error('An error occurred! Password could not be set.');
  });
}
```

FIGURE 4.5: A typical coverage file.

flakiness arises. The solution to such a situation is to run the test suite again, and most probably, this time around, the test suite will run without any problem.

- The coverage report shows ternary operator, single line if-else statements as uncovered even though they are covered. So by inspecting the code around these type of statements, we have to decide whether the statement is used or unused.

- The coverage report fails to cover the catch block in the try-catch statements. The reason being that the code inside the try block is based on the backend call and that backend call, in turn, depends on the database call. So to fail the try block and execute the catch block, the database call should fail or crash. However, with end-to-end tests, we cannot control the backend and database because this test suite does not have direct access to the backend codebase. Hence, the catch blocks remain uncovered.

Chapter 5

Vulnerability scan

This chapter starts by going through the basics of web application vulnerability scanning, like various vulnerabilities that can be exploited on a web application. Then it discusses vulnerability scan using the OWASP ZAP tool; we have talked about this tool in section 2.2 of chapter 2. Next, it defines exploring and scanning, the two core concepts of OWASP ZAP vulnerability detection. After that, it describes proxying through the OWASP ZAP server and some of the difficulties we faced with running our end-to-end tests through the OWASP ZAP server, and how did we overcome those difficulties. Finally, it ends with explaining report generation in OWASP ZAP.

5.1 Basics of website vulnerability scanning

Vulnerability scanning is a process of checking our web application against a huge database of known vulnerabilities and identify the potential security weakness in our web application. These known vulnerabilities can be categorized into the following types:

- **File upload vulnerability**: This type of vulnerability occurs when the web application allows users to upload files. It is divided into two types "local file upload vulnerability" and "remote file upload vulnerability". The local file upload vulnerability occurs when the web application allows users to upload a malicious file which is then executed by the web application. The remote file upload vulnerability occurs when the web application receives user input to fetch some malicious file from the internet. Hackers then execute these files on the web application.

- **Code execution vulnerability**: In this type of vulnerability, hackers first gather information about the system on which web application is running, like the server's operating system. Hackers then try to gain administrative access to this system. Once hackers manage to gain administrative access, they can remotely execute any malicious code.

- **Local file inclusion vulnerability**: This kind of vulnerability allows hackers to read any file within the same server. This vulnerability is critical because hackers can read any file, so if a web application stores any critical files like password files, hackers will be able to read them. Also, if there are many web applications on the same server, the hacker will access all those web application files.

- **Remote file inclusion vulnerability**: It is similar to the local file inclusion vulnerability. If the server is configured to allow certain functions like the *allow_url_fopen()* function in PHP to retrieve data from remote servers, then hackers can include any file from any computer into the web application. This allows hackers to run reverse shells [65], system commands and gain complete access to the server.

- **Injection vulnerability**: The web application is supplied with untrusted inputs to exploit such kind of vulnerability. The web application process the input as a part of a command or query. However, it is not a part of the command or query, leading to altering that command or query execution. Injection vulnerability is a widespread and dangerous attack. This type of attack is well understood; thus, many reliable and freely available tools allow even inexperienced attackers to abuse this type of vulnerability. The popular type of injection vulnerabilities are listed below:

  - Code injection [66]
  - SQL injection [67]
  - Cross-site scripting [68]
  - Command injection [69]
  - CRLF injection [70]
  - LDAP injection [71]
  - XPath injection [72]

5.2 Concepts of OWASP ZAP

In the section 2.2.4 of chapter 2, we have introduced the OWASP ZAP tool. In this section we will get more familiar with the some of the important concepts of OWASP ZAP:

- **Exploring applications**: There two main phases in OWASP ZAP scanning "exploring" the application and "attacking" the application, here we will discuss exploring. The OWASP ZAP gathers information about the web application in the exploring phase, like the URLs it visited. It intercepts all the requests and responses; with this information, a sites tree is generated. Which is then used in the attacking phase. There are multiple ways to explore web applications with OWASP ZAP, discussed below:
  - **Manual explore**: There is a manual explore tab present on the desktop interface. Clicking on this tab will start the standard Chrome or Firefox browser. In this browser, we need to manually explore the web application, i.e. click on all the buttons, links etc., present in the web application. All these requests and responses proxy through the OWASP ZAP server, and a sites tree is generated. The drawback of this is that human intervention is required to explore the complete application.
  - **Proxying end-to-end tests**: In this approach, we write a good end-to-end test suite, which covers all the parts of the web application. We configure the browser on which our end-to-end tests are running to proxy through the OWASP ZAP server. The OWASP ZAP server then catches all the requests and responses and generate a sites tree. This approach is considered best amongst all because it does not require human to explore through the web application, and it works on actual data, which is a problem in standard and AJAX spider. This is the approach we have used for exploring the Prutor.
  - **Standard spider**: The spiders are used when we do not have end-to-end tests for the web application, and we do not want to explore manually. The problem with these spiders is they do not provide relevant data, i.e. say we want to fill a form on web application, these spiders will fill all the input fields with "owasp zap" string or something similar. Having excellent data is good for the attacking phase. The *standard spider* is a standard web crawler that goes through all the web application URLs and generates a sites tree. It is swift, but the problem with a standard spider is that it does not handle modern web

applications that use lots of Javascript. That is why OWASP ZAP has the AJAX spider.

  - **AJAX spider**: As discussed above the standard spider is not good with web applications that use lots of Javascript and *single page applications*(SPAs), to deal with this problem the AJAX spider was introduced. It uses the headless browser to access the SPAs. So the AJAX spider launches the browser and the browser then makes the requests to generate the sites tree.

- **Scanning/Attacking applications**: In this phase we actually find vulnerabilities in the web application. There are two types of scanning in OWASP ZAP "passive scanning" and "active scanning". Let us now discuss them one by one:

  - **Passive Scanning**: In passive scanning, ZAP looks at requests and responses and do not modify them. Just by looking at the requests and responses, if ZAP finds some vulnerabilities, it reports them. As it only looks at the requests and responses and does not modify them, it is safe to use on any website. ZAP runs the passive scan by default. It runs alongside the exploration phase in the background thread to not slow down the exploration of the web application.
  - **Active Scanning**: In active scanning, ZAP actually attacks the web application, so we should use it only on the web applications with permission to attack. It tries to find potential vulnerabilities by using known attacks against the web application. Automated scanning tools can only find certain types of vulnerabilities, so these scans should follow manual scanning to discover all possible types of vulnerabilities.

5.3 Steps OWASP ZAP scanning

The first step is to configure the browser to proxy the request and responses through the OWASP ZAP server. The next step is to start our end-to-end test suite this will start the web application exploration phase and simultaneously the passive scanner will also start in the background thread. Once our end-to-end test suite finishes the exploration phase is completed, the next step is the active scanning phase. In the active scanning phase OWASP ZAP will attack our web application. Once the active scan is done we can generate the full scan report from the reports tab. In the appendices A we have mentioned these steps in detail.

Chapter 6

End Results

In this chapter, we examine the final results that we got from both backend and frontend code coverage reports and the vulnerability scan report. First we examine the backend code coverage report in the section 6.1, then in the section 6.2 we examine the frontend code coverage report, finally in the section 6.3 we examine the vulnerability scan report.

6.1 Backend code coverage report analysis

Table 6.1 shows the backend code coverage statistics. Looking at the stats, we can say that there is not a lot of dead code present in the backend codebase. But, as we have discussed in section 3.1.1 of chapter 3, the "app_modules" folder is only accessed through the "routes" folder. So if the routes folder does not use any function from the app_modules folder, that function becomes unreachable, thus making it unused. We wrote scripts to check the reachability of functions of the app_modules folder from the routes folder. We found that there are some functions that are unreachable and thus unused. Table 6.2 gives the name of such functions. Figure 6.1 shows the bar graph of the original size and reduced size of the "app_modules" folder in the backend codebase after removing unused functions. The number of lines removed from the codebase is around **490** lines which are about a **25.73%** reduction in size.

Function name	statements covered	percentage statements covered
accounts.js	209/231	90.48
analytics.js	80/80	100
assignments.js	291/298	97.65

assignta.js	70/85	82.35
counts.js	26/26	100
dataviz.js	40/40	100
engine.js	210/210	100
events.js	171/178	96.07
grading.js	55/55	100
notifications.js	50/50	100
pager.js	72/72	100
problems.js	194/232	83.62
reportgen.js	45/45	100
scratchpad.js	50/50	100
statistics.js	79/79	100
tasks.js	27/27	100
telemetry.js	42/42	100
tests.js	20/20	100

TABLE 6.1: Backend code coverage statistics

analyticsjs/getCodeInstance()	analytics.js/addStar()
analytics.js/getCodeClusters()	analytics.js/getCodeWithErrors()
analytics.js/getSyntacticErrorTypes()	analytics.js/setSeen()
analytics.js/getInstancesHavingError()	analytics.js/removeStar()
events.js/getCurrentEvent()	feedback.js/getFeedback()
feedback.js/deleteSpec()	feedback.js/toggleSpec()
feedback.js/updateSpecInput()	feedback.js/getSpecs()
feedback.js/getRegisteredFeedbacks()	feedback.js/updateSpecCode()
feedback.js/updateFeedback()	feedback.js/addSpec()
mailer.js/send()	notifications.js/markSeen()
notifications.js/notify()	problems.js/getProblem()
problems.js/getPartialSolution()	reportgen.js/getStudentUsers()
statistics.js/getPerformanceStatistics()	

TABLE 6.2: Unused functions in the backend codebase

6.2 Frontend code coverage report analysis

All code related to DOM manipulation, route handling, etc., is present in the "public/scripts" folder of the webapp codebase. There are entire functionalities like debugging

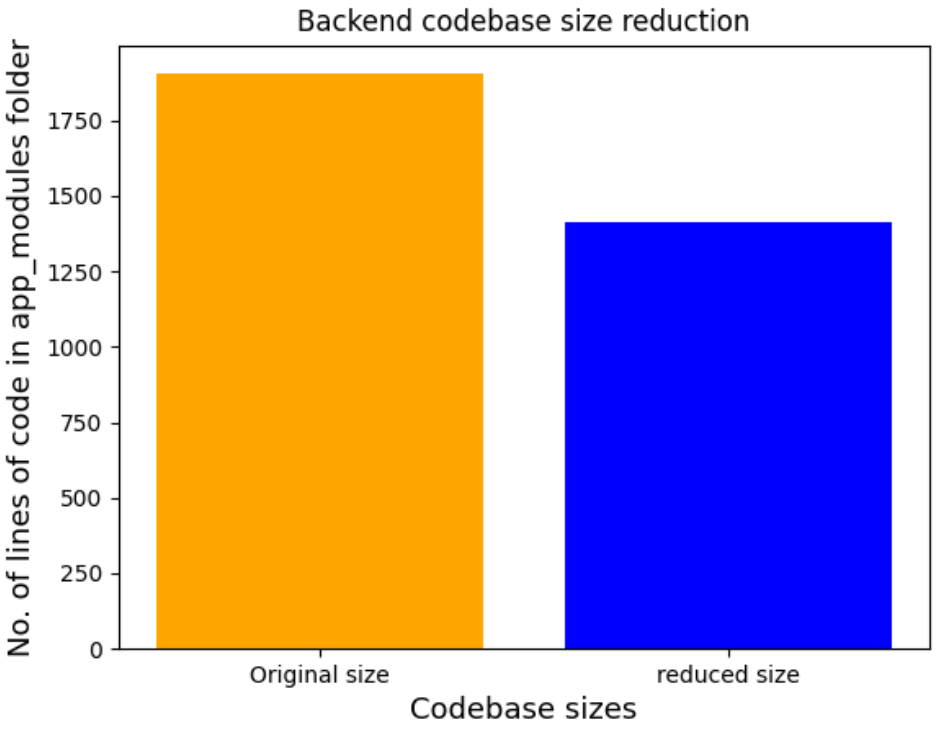

FIGURE 6.1: Reduction in backend codebase size

system and feedback system, which are deprecated, so large pieces of unused code are present in the frontend codebase. Table 6.3 shows statistics of files in the "public/scripts" folder that has unused code. As we can see, some files have 0% coverage, which signifies that the code inside those files is not at all used. The reason being functionalities provided by those files are depreciated. Figure 6.2 shows the top five files with the most significant codebase reduction. Figure 6.3 shows the overall codebase reduction in the "public/scripts" folder. The number of lines removed from the "public/scripts" folder is around **1378** lines which is about **15.11%** reduction in size. Figure 6.4 shows combined backend and frontend codebase reduction; it considers the files inside the "app_modules" in the backend and files inside the "public/scripts" folder in the frontend. Thus, figure 6.4 does not reflect the exact sizes but shows a majority codebase of backend and frontend. The combined reduction in the codebase size comes out to be **16.94%**.

Function name	statements covered	percentage statements covered
editor.js	379/415	91.32
main.js	190/202	94.05
admin/dataviz/main.js	17/20	85
admin/dataviz/progress.js	0/167	0

admin/problems/manage.js	468/485	96.49
admin/problems/upload.js	172/180	95.55
editor/debug.js	126/402	31.34
editor/scratch.js	655/660	99.24
editor/event.js	349/361	96.67
editor/display.js	337/531	63.46
admin/events.js	667/678	98.37
admin/viewer.js	478/487	98.15
admin/data.js	0/159	0
admin/jobs.js	0/176	0
admin/jobstatus.js	0/135	0
modules/Analyzer.js	0/158	0

TABLE 6.3: Frontend code coverage statistics

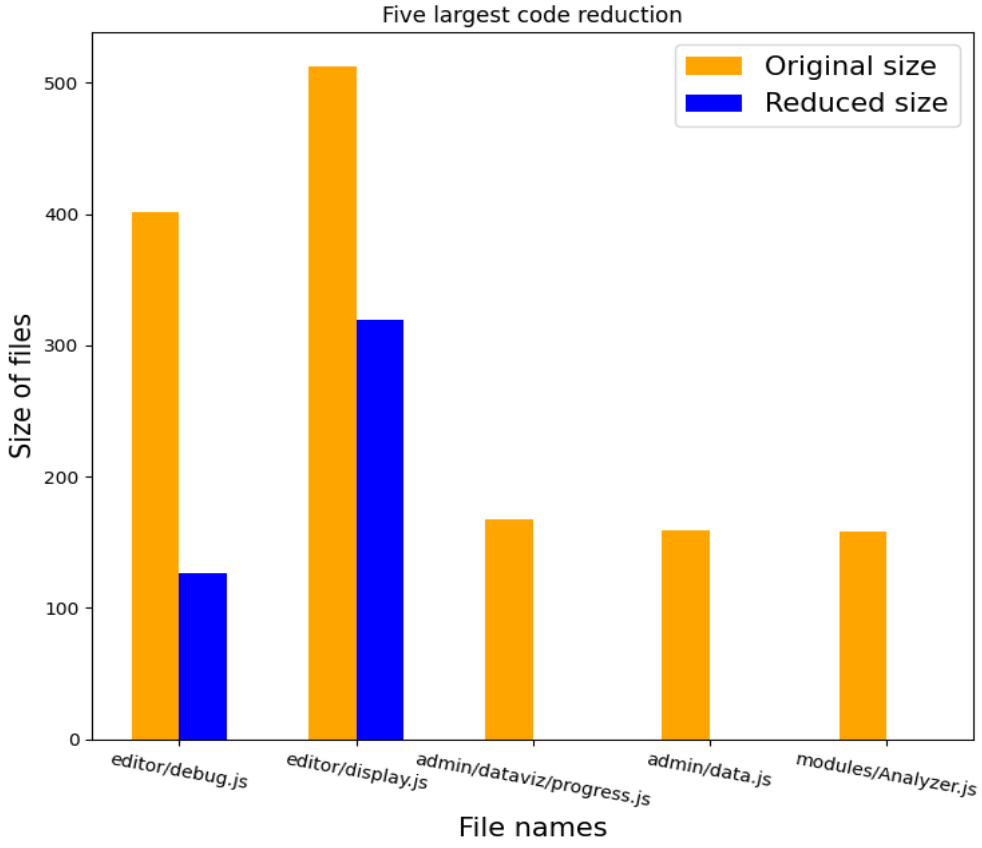

FIGURE 6.2: Top five files with largest code reduction in frontend codebase

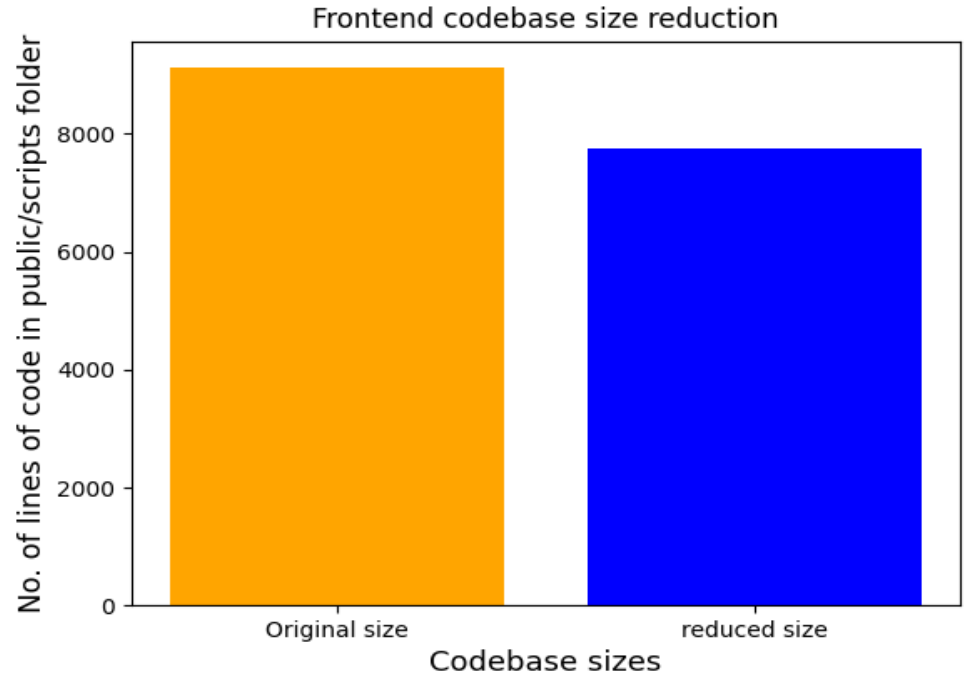

FIGURE 6.3: Reduction in frontend codebase size

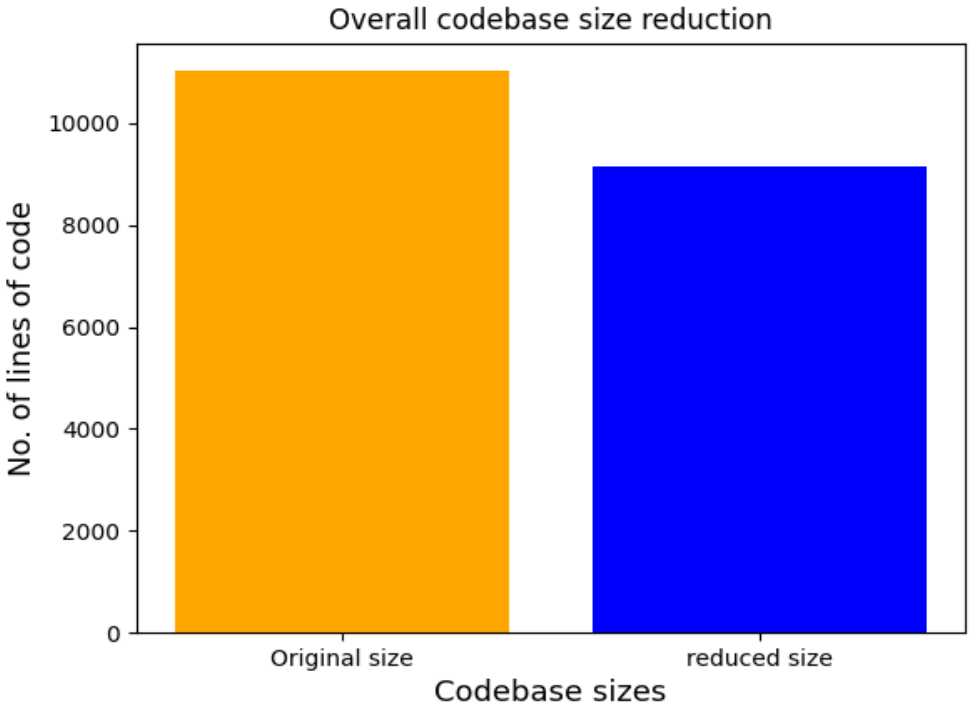

FIGURE 6.4: Combined backend and frontend codebase reduction

6.3 Vulnerability scan report analysis

The OWASP ZAP vulnerability scan report has four categories of alerts based on vulnerability severity; they are High, Medium, Low and Informational alerts. The high, medium and low alerts are actual vulnerabilities with descending order of severity, but informational alerts are like warnings and best practices. Figures 6.5, 6.6 and 6.7 show plots indicating the number of medium, low and informational alerts present in different account types of Prutor. There is no plot for the high severity alerts because, according to OWASP ZAP scan reports, there are no high severity vulnerabilities present in Prutor. In all these plots, we can see a common trend that as we move from instructor to student account types, the number of vulnerabilities decreases. It is expected behaviour because account types of Prutor follow the trend, $instructor \supset Tutor \supset TA \supset student$ ($\supset$ is superset symbol) for access privileges, so access to the number of functionalities also follow the same trend. Thus, there are more chances of finding vulnerabilities in instructor account than in student account.

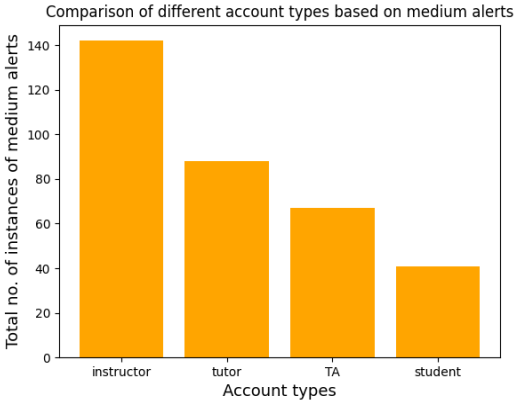

FIGURE 6.5: Medium severity alerts

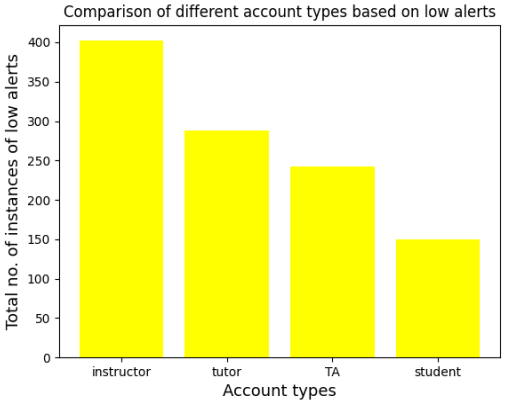

FIGURE 6.6: Low severity alerts

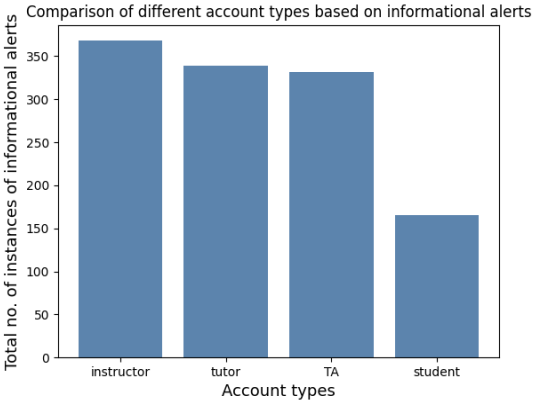

FIGURE 6.7: Informational alerts

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The main aim of this thesis was to get the lean and mean version of Prutor. After writing the unit test suite for generating the backend code coverage report using the Jest framework and writing the end-to-end test suite for generating the frontend code coverage report using Puppeteer, Chrome code coverage, Jest. We have successfully removed about **490 lines** from the backend codebase and about **1378 lines** of code from the frontend codebase. The benefits of removing these lines of code are that the codebase has become more maintainable. We have a high degree of confidence that the Prutor codebase does not have any unused code, and the latency of Prutor will also be slightly improved.

We even touched upon the security aspect of Prutor by generating a detailed report on security vulnerabilities present in the Prutor using the tool OWASP ZAP. This report gives a detailed description of severity, URL, how to recreating the vulnerability etc. Using this information, we can patch those vulnerability issues to avoid any security breaches.

We have demonstrated the use of tools like Puppeteer and OWASP ZAP. These tools are potent and can do various tasks other than those mentioned in this thesis, like Puppeteer is used for web scraping, generating screenshots etc. Thus practical demonstration of these tools will enhance the knowledge of the reader.

7.2 Future Work

- The vulnerability scan report may contain false positives, so an alert in the report must be manually checked to ensure that it is a real vulnerability. Also, there may be vulnerabilities that OWASP ZAP does not discover. So a scan by a vulnerability scanner like OWASP ZAP should be followed by manual scanning to find every possible security breach. Further, the manual scanning step should be followed by patching these vulnerabilities to avoid any potential security breaches.
- The test suites for vulnerability scanning need refactoring because currently, there are four test suites for each account type instructor, tutor, TA and student. These four test suites can be shrunk down to a single test suite to avoid code repetition.
- We have tried our best to cover all the use cases and scenarios while writing the test suites, but some situations may arise for which tests are not written. So, we always need to be vigilant about them and write tests for them.
- A code coverage testing is not done for the codebase of the Engine component of Prutor. We can do similar code coverage analysis for this codebase and eliminate all the unused PHP code.

References

[1] "Dead code." https://en.wikipedia.org/wiki/Dead_code (accessed June 08, 2021).

[2] "Prutor–a tutoring system for introductory programming." https://www.cse.iitk.ac.in/users/karkare/prutor (accessed June 08, 2021).

[3] "Web applications." https://en.wikipedia.org/wiki/Web_application (accessed June 08, 2021).

[4] "Docker." https://www.docker.com/ (accessed June 08, 2021).

[5] R. Das, "A platform for data analysis and tutoring for introductory programming," Master's thesis, Indian Institute of Technology Kanpur, June 2015.

[6] A. Jindal, "Scaling a web-based tutoring system from classrooms to moocs," Master's thesis, Indian Institute of Technology Kanpur, July 2018.

[7] S. Rajoriya, "Orchestration of cloud-based intelligent tutoring system," Master's thesis, Indian Institute of Technology Kanpur, June 2020.

[8] "Testing tools." https://www.lambdatest.com/blog/top-javascript-automation-testing-framework/ (accessed June 08, 2021).

[9] "Code coverage tools." https://betterprogramming.pub/how-to-measure-your-javascript-code-coverage-with-istanbul-8d53b6c9e711 (accessed June 08, 2021).

[10] "Automation tools." https://developers.google.com/web/tools/puppeteer (accessed June 08, 2021).

[11] "Jest." https://jestjs.io/ (accessed June 08, 2021).

[12] "Headless chrome." https://developers.google.com/web/updates/2017/04/headless-chrome (accessed June 08, 2021).

[13] "Devtools." https://chromedevtools.github.io/devtools-protocol/ (accessed June 08, 2021).

[14] "Puppeteer." https://pptr.dev/ (accessed June 08, 2021).

[15] "OWASP ZAP vulnerability scanning tool." https://www.zaproxy.org/ (accessed June 08, 2021).

[16] "website routes." https://divpusher.com/glossary/routing/ (accessed June 08, 2021).

[17] "Database query." https://www.educative.io/blog/what-is-database-query-sql-nosql (accessed June 08, 2021).

[18] "Docker documentation." https://docs.docker.com/get-started/ (accessed June 08, 2021).

[19] "Docker container." https://www.docker.com/resources/what-container (accessed June 08, 2021).

[20] "MySQL." https://www.mysql.com/ (accessed June 08, 2021).

[21] "Mongodb." https://www.mongodb.com/ (accessed June 08, 2021).

[22] "Memcached." https://github.com/memcached/memcached (accessed June 08, 2021).

[23] "Node.js." https://nodejs.org/en/ (accessed June 08, 2021).

[24] "Express." https://expressjs.com/ (accessed June 08, 2021).

[25] "doT templating library." https://olado.github.io/doT/index.html (accessed June 08, 2021).

[26] "jQuery." https://jquery.com/ (accessed June 08, 2021).

[27] "PHP." https://www.php.net/manual/en/ (accessed June 08, 2021).

[28] "Sandbox." https://en.wikipedia.org/wiki/Sandbox_(computer_security) (accessed June 08, 2021).

[29] "NPM package manager." https://www.w3schools.com/whatis/whatis_npm.asp (accessed June 08, 2021).

[30] "React." https://www.w3schools.com/react/ (accessed June 08, 2021).

[31] "Unit testing." https://www.tutorialspoint.com/software_testing_dictionary/unit_testing.htm (accessed June 08, 2021).

[32] "Integration testing." https://www.tutorialspoint.com/software_testing_dictionary/integration_testing.htm (accessed June 08, 2021).

[33] "END-to-END testing." https://www.guru99.com/end-to-end-testing.html (accessed June 08, 2021).

[34] "Chrome code coverage." https://developer.chrome.com/docs/devtools/coverage/ (accessed June 08, 2021).

[35] "Single-page application." https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58 (accessed June 08, 2021).

[36] "Server-side rendering." https://www.educative.io/edpresso/what-is-server-side-rendering (accessed June 08, 2021).

[37] "Penetration testing." https://www.imperva.com/learn/application-security/penetration-testing/ (accessed June 08, 2021).

[38] "Java programming language." https://en.wikipedia.org/wiki/Java_(programming_language) (accessed June 08, 2021).

[39] "XML format." https://www.w3schools.com/xml/default.ASP (accessed June 08, 2021).

[40] "markdown format." https://www.markdownguide.org/ (accessed June 08, 2021).

[41] "JSON format." https://www.w3schools.com/js/js_json_intro.asp (accessed June 08, 2021).

[42] "Kubernetes." https://kubernetes.io/ (accessed June 08, 2021).

[43] "Bare metal server." https://en.wikipedia.org/wiki/Bare-metal_server (accessed June 08, 2021).

[44] "DOM (document objest model)." https://www.w3schools.com/js/js_htmldom.asp (accessed June 08, 2021).

[45] "Event listners." https://developer.mozilla.org/en-US/docs/Web/API/EventListener (accessed June 08, 2021).

[46] "its-engine repository." https://bitbucket.org/karkare/its-engine/src/master/ (accessed June 08, 2021).

[47] "$_SERVER variable in php." https://www.w3schools.com/php/php_superglobals_server.asp (accessed June 08, 2021).

[48] "spl_autoload_register function." https://tutorials.supunkavinda.blog/php/oop-autoloading (accessed June 08, 2021).

[49] "Uniform resource identifier[URI]." https://en.wikipedia.org/wiki/Uniform_Resource_Identifier (accessed June 08, 2021).

[50] "PHP proc_open function." https://www.php.net/manual/en/function.proc-open.php (accessed June 08, 2021).

[51] "PHP proc_close function." https://www.php.net/manual/en/function.proc-close.php (accessed June 08, 2021).

[52] "Facebook." https://developers.facebook.com/ (accessed June 08, 2021).

[53] "Jest matchers documentation." https://jestjs.io/docs/expect#expectvalue (accessed June 08, 2021).

[54] "Jalangi." https://people.eecs.berkeley.edu/~ksen/slides/jalangi-soap15.pdf (accessed June 08, 2021).

[55] "Tree Shaking." https://webpack.js.org/guides/tree-shaking/ (accessed June 08, 2021).

[56] "Google Closure Compiler." https://developers.google.com/closure/compiler (accessed June 08, 2021).

[57] "Dependency management (we don't have to do anything with gradle, but the definition of dependency is apt here)." https://docs.gradle.org/current/userguide/dependency_management.html (accessed June 08, 2021).

[58] "JSDoc comments." https://jsdoc.app/about-getting-started.html (accessed June 08, 2021).

[59] "Puppeteer browser options complete list." https://devdocs.io/puppeteer/ (accessed June 08, 2021).

[60] "CSS selectors." https://www.w3schools.com/cssref/css_selectors.asp (accessed June 08, 2021).

[61] "page object methods." https://devdocs.io/puppeteer/index#pagegotourl-options (accessed June 08, 2021).

[62] "Jest configuration options." https://jestjs.io/docs/configuration (accessed June 08, 2021).

[63] "Jest testenvironment configuration option." https://jestjs.io/docs/configuration#testenvironment-string (accessed June 08, 2021).

[64] "Test Flakiness." https://engineering.atspotify.com/2019/11/18/test-flakiness-methods-for-identifying-and-dealing-with-flaky-tests/ (accessed June 08, 2021).

[65] "reverse shell." https://stackoverflow.com/questions/35271850/what-is-a-reverse-shell (accessed June 08, 2021).

[66] "Code injection vulnerability." https://owasp.org/www-community/attacks/Code_Injection (accessed June 08, 2021).

[67] "SQL injection vulnerability." https://owasp.org/www-community/attacks/SQL_Injection (accessed June 08, 2021).

[68] "Cross-site scripting (XSS) vulnerability." https://owasp.org/www-community/attacks/xss/ (accessed June 08, 2021).

[69] "Command injection vulnerability." https://owasp.org/www-community/attacks/Command_Injection (accessed June 08, 2021).

[70] "CRLF injection vulnerability." https://owasp.org/www-community/vulnerabilities/CRLF_Injection (accessed June 08, 2021).

[71] "LDAP injection." https://owasp.org/www-community/attacks/LDAP_Injection (accessed June 08, 2021).

[72] "XPATH injection." https://owasp.org/www-community/attacks/XPATH_Injection (accessed June 08, 2021).

[73] "Code coverage." https://en.wikipedia.org/wiki/Code_coverage (accessed June 08, 2021).

[74] "Prutor repository." https://bitbucket.org/karkare/prutor/src/master/ (accessed June 08, 2021).

[75] "allow_url_fopen() function in PHP." https://www.php.net/manual/en/filesystem.configuration.php (accessed June 08, 2021).

Appendices

Appendix A

OWASP ZAP vulnerability Scan instructions

A.1 OWASP ZAP proxy configuration

This section assumes that the OWASP ZAP desktop application is already installed on the system. The application can be easily downloaded from the OWASP ZAP website [15] and installed on the system. The proxy configuration instructions shown here are carried out on Ubuntu 18.04 operating system. The other operating systems may have a slightly different configuration. All we have to do is change the network proxy to manual and change "HTTP proxy", "HTTPS proxy", "FTP proxy" and "Socks hosts" to IP "localhost" and port "8080". By default, the OWASP ZAP proxy server runs at "localhost:8080". The step by step configuration instructions are given below.

1. Start the OWASP ZAP desktop application; this will also start the proxy server by default at "localhost:8080". We can verify that our proxy server has started or not by going to "localhost:8080" from our browser. We should see a webpage similar to figure A.1.

2. Go to the network proxy settings of the system and change "HTTP proxy", "HTTPS proxy", "FTP proxy" and "Socks hosts" to IP "localhost" and port "8080". Figure A.2 shows the setup for Ubuntu 18.04.

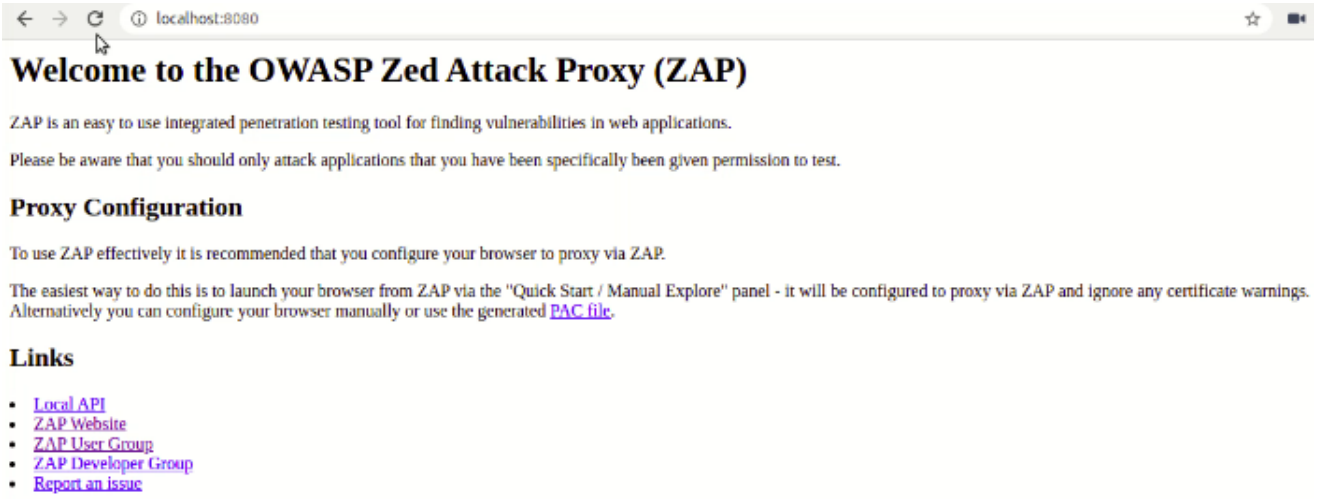

FIGURE A.1: OWASP ZAP proxy server homepage

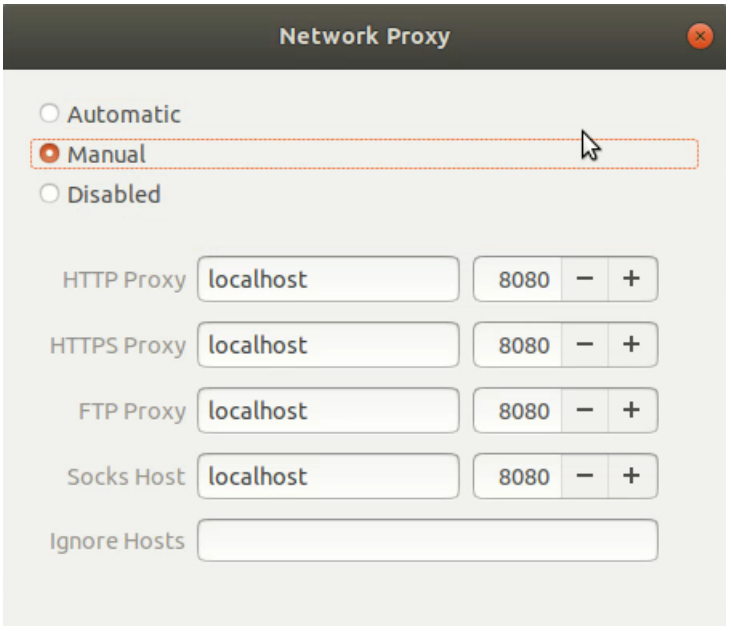

FIGURE A.2: Network proxy settings

A.2 Initiate active scan

After the proxy server configuration is done, we need to run the program **start.js**. The command to run this program is `node start.js`. This program will first ensure that the proxy server is set and then choose an account type and run the tests. While running the tests, the OWASP ZAP will create a site tree. After the tests are finished running, we need to perform an active scan. The instructions to perform an active scan are given below:

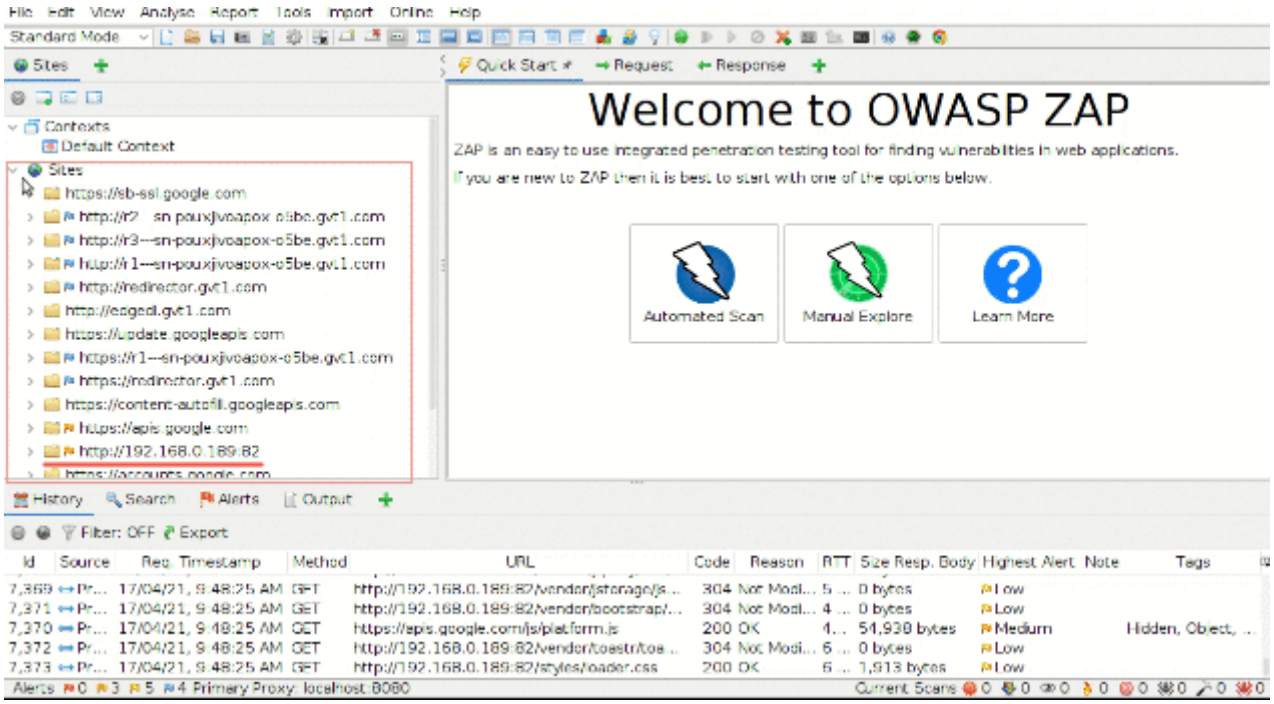

FIGURE A.3: OWASP ZAP site tree

1. In figure A.3, the red rectangle shows the site tree. In this site tree, there will be a folder "http://⟨*IPWherePrutorIsRunning*⟩:82"; in this case, it is "http://192.168.0.189:82". This folder contains all the URL, requests and responses etc., collected from the proxy server for Prutor. We need to create a new context for this folder by right-clicking on that folder and selecting "Include in Context", and then selecting "New Context". We create a new context because the active scan will run only in the new context, i.e. only on the URL, requests and responses etc., of this folder.

2. After including the folder in a new context, all we have to do is right-click on that folder and select "Attack", then select "Active Scan...". The active scanner will start, and it will take some time for the scanning to finish.

A.3 Generating scan report

Once the scanning is finished, the final step is to generate the report. In the top right corner of the OWASP ZAP desktop application, there are various options "File", "Edit", "View", etc. One such option is "Report" click on that option; various options to generate a report will appear like "Generate HTML report", "Generate XML report", etc. click

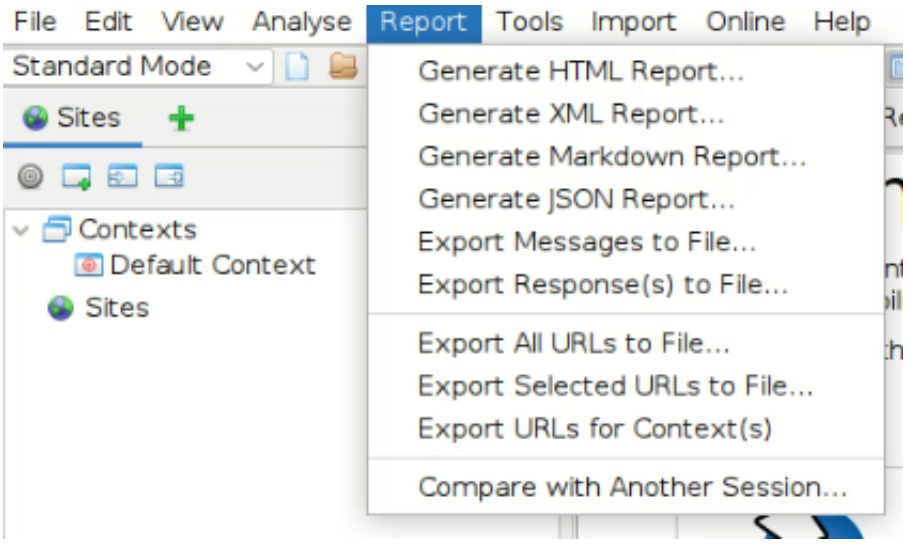

FIGURE A.4: Report generation in OWASP ZAP

on the suitable option, and after giving name and location to save, the report will be generated. Figure A.4 shows the report generation step.

Appendix B

Projects Links

Here we have shared the links to the repositories of projects and some associated links.

- Unit test suite: `https://bitbucket.org/prutor-optimisation-and-vulnerability-scan/prutor_unit_testsuite/src/master/`
- End-to-end test suite: `https://bitbucket.org/prutor-optimisation-and-vulnerability-scan/prutor_end-to-end_testsuite/src/master/`
- Vulnerability Scanning: `https://bitbucket.org/prutor-optimisation-and-vulnerability-scan/prutor_vulnerability_scan/src/master/`
- Related videos: `https://drive.google.com/drive/folders/1TW97hWUlgCve4Y6ixgHEs5qh_PXv4oTz?usp=sharing`