Thesis Project: Fixing Authentication-related Vulnerabilities With Custom ESLint Rules

Fabian Harlang (faha@itu.dk), Rares-Ionut Mocanu (rarm@itu.dk)

February 20, 2025

Enhancing Authentication Practices in Express Application Development

---

IT University Copenhagen: Applied Security Faculty

Supervisor: Willard Rafnsson (wilr@itu.dk)

1 Abstract

Recently, Rafnsson et al. demonstrated the feasibility of using ESLint - a linter - to find and automatically fix vulnerabilities in JavaScript programs [31]. Subsequent development of this work has culminated in a (recently released) tool *Thunderhorse* [2], which has been demonstrated to outperform existing open-source SAST (Static Analysis Security Tools) tools in terms of performance and F-score. *Thunderhorse* detects and fixes many common vulnerabilities, including XSS, SQL injection, unsafe path traversal, hardcoded credentials, vulnerable dependencies, and insecure ciphers. In this project, we will improve the ESLint plugin Thunderhorse even further by implementing additional customized rules that find and automatically fix vulnerabilities in web applications' code. Our rules will specifically target authentication-related bugs in the context of express/node-driven applications. To demonstrate the usefulness of our rules, we will conduct a performance evaluation using the ESLint RuleTester and assess the efficiency of our rules in regard to false positives/negatives. In addition, we will build a web page for demonstration purposes that is intentionally vulnerable to the exploits that our customized rule aims to mitigate. Our rules will become part of *Thunderhorse* and made available to JavaScript developers at large through *Node Package Manager*. Ultimately, our goal is thus to improve software security by helping developers adopt best practices for secure coding using ESLint.

Contents

1 Abstract **1**

2 Introduction **5**
2.1 Methodology 5
2.1.1 Implementation 5
2.1.2 Experimentation 5
2.2 Contributions 6

3 Background **6**
3.1 Finding and Fixing Vulnerabilities on the Web 6
3.1.1 Program Analysis 6
3.1.2 ESLint as a Static Analysis Security Tool 7
3.1.3 Thunderhorse ESlint-plugin 8
3.2 Authentication in Express/node 9
3.2.1 Passport js 10
3.2.2 Custom built authentication system 10
3.3 Authentication Bugs on the Web 11
3.3.1 Password related vulnerabilities 11
3.3.2 Brute force and DoS related vulnerabilities 12
3.3.3 OAuth Implicit Flow 12
3.3.4 Cookie related vulnerabilities 16
3.3.5 Cross-Site Request Forgery (CSRF) 18
3.4 Related Work 19

4 Implementation **20**
4.1 Cookie Safe Attributes 20
4.1.1 Examples of attacks due to missing cookie attributes 20
4.1.2 Finding the Vulnerability 20
4.1.3 Fixing the Vulnerability 23
4.2 OAuth 2.0 Implicit Flow 25
4.2.1 Possible dangers regarding the use of Implicit Flow 25
4.2.2 Finding the Vulnerability 25
4.2.3 Fixing the Vulnerability 27
4.3 Cross-Site Request Forgery (CSRF) 28
4.4 Describing the Vulnerability 28
4.4.1 Finding the Vulnerability 28
4.4.2 Fixing the Vulnerability 30
4.5 Check password hashing 34
4.5.1 Describing the vulnerability 34
4.5.2 Detecting the vulnerability 34
4.5.3 Fixing the vulnerability 35
4.6 Enforce Password Policy 39
4.6.1 Describing the vulnerability 39
4.6.2 Detecting the vulnerability 40
4.6.3 Fixing the vulnerability 40
4.7 Prevent brute force 43
4.7.1 Brute force attacks 43
4.7.2 Detecting vulnerability 44

4.7.3 Fixing the vulnerability . . . 45
4.8 Insecure express session . . . 48
4.8.1 Describing the vulnerability . . . 48
4.8.2 Detecting the vulnerability . . . 48
4.8.3 Fixing the vulnerability . . . 49
4.9 Express rate limit . . . 50
4.9.1 Describing the vulnerability . . . 50
4.9.2 Detecting the vulnerability . . . 51
4.9.3 Fixing the vulnerability . . . 52

5 Performance Evaluation 53
5.1 Evolution of Testing Tools . . . 53
5.1.1 ESlint RuleTester utility . . . 53
5.1.2 Combining ESLint's In-Built Testing Functionality with Mocha for Evaluating Linting Rules . . . 54
5.2 Evaluating Linting Rules Using Confusion Matrix . . . 55
5.3 Understanding the performance score . . . 56

6 Discussion 59
6.1 CSRF-Rule journey . . . 59
6.2 Deviating naming convention may cause false negative . . . 59
6.3 Comparison with other ESLint rules . . . 60
6.4 Attempting to improve Implicit flow rule further . . . 60

7 Conclusion 61
7.1 Future Work . . . 62
7.2 Improving metrices . . . 62

List of Figures

1 Login/register using node/express with passport js . . . 11
2 OAuth 2.0 flow . . . 13
3 STP Flow . . . 18
4 Finding Parameter in `res.cookie()` . . . 21
5 Checking Options . . . 22
6 Fixing Conditions (1) . . . 23
7 Fixing Conditions (2) . . . 24
8 Found Vulnerability `response_type=token` in String Literal . . . 26
9 Found Vulnerability `response_type=token` in Template Literal . . . 26
10 Finding Literal Node with String `response_type=token` . . . 26
11 Finding Template Literal Node with string `response_type=token` . . . 27
12 Fixing Literal Node into `response_type=code` . . . 27
13 CSRF Rule - Variable Declaration . . . 29
14 CSRF Rule - Finding Lack of CSRF token protection . . . 29
15 CSRF Rule - Nodelist . . . 30
16 CSRF Rule - Warning for Verification . . . 30
17 CSRF Rule - Adding Code . . . 31
18 MongoDB-object containing plain text password detected . . . 35
19 Server responding after registration with plain text password . . . 35

20 Code after applying the fix - secure handling of registration using hashing . 39
21 User object logged to the console with hashed password 39
22 Warning message in register form due to violation of password policy 43
23 Callback function bruteforce.prevent() passed to route 46
24 Potential Brute force attempt detected 47
25 Inspecting the cookie attributes in developer tools 50
26 Server response in the terminal for rate limit exceeded 53
27 Mocha Report on Secure-Cookie-Rule Test 54

2 Introduction

2.1 Methodology

Considering such a dynamic environment as application development, security to this day remains as prevalent as ever. A great deal of security vulnerabilities has their source from commonly used libraries and frameworks, as well as bad practices or lack of knowledge in terms of secure programming. The rapid pace of development, often driven by the necessity to keep up with market demands or customer needs, often leads to the appearance of such vulnerabilities or encourages developers to overlook them simply. In our Master's thesis, we explore the role of ESLint and its potential in reducing security risks within JavaScript applications. We will leverage the static analysis capabilities of ESLint, and we will propose a number of innovative security rules that deal with some of the most common vulnerabilities that one can come across in different applications. Our rules can be categorized as dealing with authentication vulnerabilities on the web or dealing with general vulnerabilities on the web. Our rules will not only detect these vulnerabilities, but they will also be able to provide automatic fixes in an attempt to create a secure development process for the developer. Our methodology involves two key components: *Implementation* (build) and *Experimentation* (test). In the Build phase, we will develop rules for ESLint to identify and rectify security vulnerabilities in JavaScript programs. In the Experiment phase, we will evaluate the efficacy of our rules by conducting several test cases and evaluating them on an intentionally vulnerable authentication application.

2.1.1 Implementation

Through this thesis, we aimed to create nine linting rules within ESLint, that can be used by developers for finding authentication-related vulnerabilities in their code, but also to fix them. The rules we have decided to create, tackle certain vulnerabilities that can be seen below and are identified by their Common Weakness Enumeration (CWE) number identifiers, most notably [21]:

- CWE-1004: Sensitive Cookie Without 'HttpOnly' Flag
- CWE-1275: Sensitive Cookie with Improper SameSite Attribute
- CWE-200: Exposure of Sensitive Information to an Unauthorized Actor - (Rule that attempts to tackle the implementation of the OAuth 2.0 Implicit Flow protocol)
- CWE-352: Cross-Site Request Forgery (CSRF)
- CWE-521: Weak Password Requirements
- CWE-614: Sensitive Cookie in HTTPS Session Without 'Secure' Attribute
- CWE-307: Improper Restriction of Excessive Authentication Attempts

2.1.2 Experimentation

The performance of our rules has been analysed by using the in-built testing environment of ESLint. The test units are custom-made and reflect the situations that we thought of while creating the rules. The tests are meant to determine certain situations in which true positives, false negatives and false positives can arise. We have used these parameters in order to calculate metrics like *recall*, *precision* and *F-score*, for each of our rules. Generally,

the *F-scores* for our rules shows that for the majority of occurrences, our rules can correctly identify vulnerabilities, however, there is room for improvement. Throughout the project, we outline different situations in which our rules have picked on false positives and false negatives and we argue how technical improvements can be made to the code, therefore improving the *F-scores*. We are aware that a larger and more diverse testing set is highly desirable for having reliable results and we aspire to add to the number of tests in the future, possibly even using more relevant benchmarking tools as well.

2.2 Contributions

The thesis project aims to enhance security for authentication-related vulnerabilities in the context of the express. Our main contribution is our *implementation*, which includes 9 distinct customized ESLint rules that specifically target security issues in the process of accessing a restricted application with login and register functionalities. The majority of the rules provide an option for automatically fixing the detected issues. Moreover, we built a customized authentication application using express and node that was made in order to discover relevant vulnerabilities and exploits within these frameworks. In our *experimentation* part, we conducted test cases for all distinct rules and gathered the result in a confusion matrix. In addition to this, we provide an in-depth insight into the potential vulnerabilities and exploit in the realm of authentication and provide insight into the technology commonly used for this purpose while discussing the benefits and shortcomings of customizing security rules to mitigate them.

Our plugin is currently named *eslint-plugin-secure-authentication* and can be downloaded via node packager manager:
`https://www.npmjs.com/package/eslint-plugin-secure-authentication`

The custom-built authentication application that we will be referring to throughout this paper can be forked from the below repository:
`https://github.com/faha92/passport_login_system`

3 Background

3.1 Finding and Fixing Vulnerabilities on the Web

3.1.1 Program Analysis

Program analysis refers to the analysis of programs that has as a focus the evaluation of different parameters such as correctness or safety but not only [11]. Some tools used for conducting and ensuring such evaluation are for example Static Application Security Testing (SAST) tools and linters like ESLint.

In order to define how accurate our rules are, we will refer to the following qualitative concepts: **soundness** and **completeness**. A program analysis tool is considered to be sound if, for every piece of code that it analyses, it is able to correctly identify a real issue - not generating any false positives. In terms of completeness, such a tool is considered complete, if it is able to find all the issues in a given code - not generating any false negatives [19].

We have approached the creation of our rules by trying to balance these two concepts mentioned above - however, it can be said that our rules have a bigger focus on soundness

since they also target very particular vulnerabilities that also tend to be quite complex. In this case, we are aware that our rules might miss some real issues and have less completeness, at the same time being able to identify vulnerabilities correctly. Even then, considering the complexity of certain applications, together with the different programming patterns and styles that developers use, it is quite difficult to benchmark the level of soundness that our rules achieve. We face the same challenge when it comes to completeness. Having perfect completeness would imply testing our tool against all possible variations of programming patterns and styles. Nevertheless, throughout the development of our rules, we aimed to refine them constantly while taking into consideration various reasons that would make our rules less complete or sound. We discuss more about the certain false positives and false negatives that we have identified for our rules, in the discussion section. In terms of quantitative analysis, in order to get a better understanding of how our program analysis rules perform, we will use the following metrics: true positives, true negatives, false positives and false negatives. Our approach will involve using a Confusion Matrix together with the mentioned metrics. From this, we will be able to derive statistics that will give us more information about our rules such as Precision, Recall and F1 Score.

We are aware that the size of the data-set plays an important role in the accuracy of the statistics that we can derive from such a matrix and we must outline that the results we have gathered are preliminary results that may not truly reflect the performance of our rules, especially given that the test cases are not necessarily representative of a more varied code style and pattern. In terms of future work, more test cases can be done and based on the outcomes, our rules could be refined.

3.1.2 ESLint as a Static Analysis Security Tool

Static Analysis Tools (SAT) are commonly used to check code in software for syntactical errors or violations of some convention or coding pattern that developers must adhere to. It may check the program for coding-related bugs like null pointer errors, division by zero, buffer overflows, or coding standard compliance that could enhance the overall readability and maintainability of the code. During the recent decade, multiple tools have emerged as coding has evolved and applications have become more complex. As a result, the demand for tools to keep large coding projects shared by numerous developers coherent and consistent has increased. Some of the most commonly used tools for the different programming languages include:

- **RuboCop**: This is a Ruby static code analyzer and linter, based on the community Ruby style guide.
- **FindBugs**: A static analysis tool designed to find bugs in *Java* code.
- **Pylint**: A tool used for *Python*, it checks for coding standards and has options to customize rules.
- **ESLint**: A most popular linting tool for JavaScript and TypeScript. It is easy to set up and configure and has an option for custom rules.

While most of the mentioned tools are useful ways of detecting violations of certain rules respectively to the programming language they were designed for, the latter, ESlint is the most suitable one for the web as it is specifically designed for JavaScript code. Besides that, it is highly configurable and offers the opportunity to create customized

rules. This opens up a variety of opportunities not limited to style guides or coding patterns but also the detection of code vulnerable, hence Static analysis Security Tool (SATS) to web exploits, which is a major concern for developers coding for the web in full-stack JavaScript.

ESLint makes it easier to identify and fix bugs and security issues in JavaScript code. It enables the customization of rules within the code base and provides instant feedback. After installing it, developers can set rules in a configuration file that flags violations as either warnings or errors. Alongside built-in rules, ESLint supports plugins of different natures to enforce specific coding styles or languages and offers automatic fixes for identified issues. It's widely used due to its compatibility with popular IDEs, and its user-friendly VS-code extension. Its highly customizable features prove invaluable in catching subtle bugs and vulnerabilities. For a more detailed description of the tool and how it works, we refer to the research project we conducted in autumn 2022. [22]

As much as ESlint stands out as an ideal linting tool for security-related bugs in JavaScript it shares a common problem with most other linting tools - namely, the dilemma of false positive and false negative and finding a sound ratio between the two. Detecting too many patterns in code that, in fact, are not issues *(false positives)* may result in the linting tool being disabled altogether while not detecting the bugs when actually needed, i.e. *false negatives* - may likewise cause the plugin to be discarded concluding that the linting simple don't effectively detect critical code patterns. For detecting more complex properties of the code, this problem may persist but should still outweigh the benefits of preventing insecure code. In other words, ESLint rules need to be "sensitive" enough to reduce *false negative* , making the application vulnerable to security while keeping *false positive* to a minimum.

3.1.3 Thunderhorse ESlint-plugin

Fixing security vulnerabilities using customized Eslint rules with automatically replacing critical code patterns is a fairly novel approach to information security. In the paper *Fixing Vulnerabilities Automatically with Linters* by Rafnsson et al *" (...) investigate the extent of which linters can be leveraged to help programmers write secure code".* [31]. As a proof of concept, they managed to build several substantial linting rules with options for automatically fixing the issues. Some of the rules aim to prevent injection attacks of different natures like *SQL injection* and *Cross-site scripting* by enforcing proper input sanitation. Their research has resulted in the publication of the node package *eslint-plugin-security* , a plugin of ESlint security rules available for the public to use in their JavaScript project using *npm install* command. The plugin has later transitioned into a larger package, thanks to other contributors from IT University Copenhagen, namely the predecessor *thunderhorse.*

This package is an expanded collection of ESLint rules aimed to enhance security assistance and development experience compared to existing ESLint security packages. It is maintained by the Applied Security faculty at the IT University in Copenhagen.

The underlying principle of this package is to consolidate all effective rules into a unified configuration while simultaneously replacing "rules that do not perform well with rules with higher-quality alternatives." [2] Currently, the plugin was released in July 2022 and can be downloaded as a node package and enabled in the ESlint configuration file. As of today, the *thunderhorse* plugin counts more than 40 different rules within the categories of cross-site scripting, secure protocols, and insecure data handling. While the plugin offers a variety of significant, high-quality linting rules, it seems there is a shortage of rules that specifically target authentication-related vulnerabilities. Observing that, we considered

potentially vulnerable patterns that could be found in the context of express/node-driven login/register applications. Before exploring the specific vulnerabilities we have targeted by the implemented rules, it is pertinent to provide additional context on the typical implementation of authentication using these technologies.

3.2 Authentication in Express/node

Using the run-time environment *node js* in combination with the web application framework *express*, it's possible to run JavaScript outside the browser and allow communication with a server, typically using middleware, restful API, and customized routes handling the request and responses. Authentication is typically accomplished through middleware libraries, which perform such tasks in the HTTP request-response cycle between the server and client. A standardized procedure is to create a signup form for the user to register, i.e. fields to input their credentials. This information will be handled as a post request that can be further processed by middleware and data validated by specific criteria. Once successfully submitted, the user is saved to a database (relational or object-oriented) to persist the data allowing users to later login into the system. Passwords may as well be using encryption algorithms such as bcrypt to hash the password before saving it to the database. Hashing turns the password into a unique, fixed-length string of characters. This a safety measure that ensures even if someone gains access to the database, they won't be able to reveal the original passwords. When the newly created user then prompts the credentials into the login form, another post request will execute; this time, the password will be hashed again and be compared to the one stored in the database of the current individual attempting to grant authorization to the system or resource usually identified by a username or email. If successfully authorized, the server initializes a session upon login. This will create either cookie or a token, depending on the implementation. In a session-based web application, the session ID is stored on a cookie in the browser. While the user remains logged in, the cookie will be sent along with every subsequent request. The server can then compare the session ID stored in the cookie against the session information stored in memory to verify the user's identity. It's also possible to enable the expiration of such cookies enforcing the use to re-authorize after a given duration.

In a token-based authentication like JWT (JSON Web Tokens), the server creates a JWT with a secret and sends the JWT to the client. The client stores the JWT (usually in local storage) and includes it in the header with every request. The server would then validate the JWT with every request from the client, and if the validation is successful, it will consider the user as authenticated. [3] In addition, authorization may also be handled by third-party services like Google or Facebook. The open-standard authorization protocol OAuth allows authentication by unrelated servers using an access/authentication-token exchange and is a common way of securing the resources of a website without the hassle of making a custom server authentication. In the context of OAuth, an access token can be seen as a 'key', granting permissions for specific actions, whereas an authentication token serves to verify the identity of the user. In combination, they make the basis for a secure, user-friendly authorization system that respects privacy and enhances usability.

This form of authentication strategy, while effective, is a complex topic that requires a more detailed examination. In later sections, we will elaborate further on its mechanisms, strengths, limitations, and potential vulnerabilities, thereby providing a more comprehensive understanding of OAuth and the risk associated with using it.

Regardless of the approach to identifying the user, all strategies are a way of protecting certain routes (URLs) or files (Path) of an application, ensuring only authorized and

legitimate users have access to the protected and perhaps sensitive information/content. Middleware functions can be used to assess if a user is authenticated (presence of a cookie or token) before the request handler function is executed. If the user isn't authenticated, the middleware can respond with an error or redirect the user to a login page.

Finally, the user should be able to terminate their session. This is done by *logging out*, which is the process of destroying the session and thus making the user no longer authenticated. This process is crucial for shared devices where multiple users need to access the system in succession, as is the case in many office environments or libraries.

3.2.1 Passport js

Passport js is a widely used authentication middleware that provides a comprehensive set of strategies that essentially streamlines the process of setting up authentication for common authentication methods, i.e. local strategy (username and password), OAuth (Google, Facebook, etc.), JWT, and many more. It's compatible with any Express-based web application and follows a middleware pattern for handling authentication, making it easily configurable. Passport.js has multiple built-in library functions that aid the developer in authenticating the user by handling the mentioned procedures like POST requests, validating credentials, and initializing sessions. It is thus up to the developer which part they want the middleware to manage and which part they want to handle themselves with customized code. This flexibility enables developers to maintain control over their authentication process while benefiting from the efficiency and reliability of Passport.js's proven strategies, but will also leave room for potential flaws and security bugs that could make the authentication process vulnerable to adversaries attempting to grant unauthorized access to the applications. To identify in which cases this could occur and the characteristics of these specific code patterns, we have experimented with these technologies building a customized authentication system allowing us to familiarize ourselves with the benefits and pitfalls of implementing authentication solely with JavaScript library code [30].

3.2.2 Custom built authentication system

In order to obtain a better understanding of the above-described authentication process within express and node applications, we manage to build a login application from scratch using these frameworks and middleware. Building this login system served as a process of ideation for identifying and detecting common authentication-related with relevancy for fixable and customized ESlint rules. Initially, we set up Express to listen on port 9000 and successfully configured and connected MongoDB. Express-session is used throughout the application to store user data and create cookies, while Passport.js handles user login and sign-up and ensures persistent authentication. The routes are configured in a separate file. They are responsible for handling routes related to user actions like *login*, *logout*, and *register*, ensuring that the user is directed to the correct places and post and get requests are executed accordingly for the authentication functionalities to work. Moreover, the app includes flash messages to display success and warning messages above the login and register forms and express-parser to interpret the requested data in JSON format adequately. We also styled the application using Bootstrap but kept this to a minimum as it's outside the scope of this project. To test if resources are protected from unauthorized users, we made a simple dashboard page that only is accessible to registered users. The first naive implementation of the authentication system was intentionally made only to include very generic functionalities that would allow users to register and log in, but from that point on, serve as a *tabula rasa* for exploring where in the process vulnerabilities could

emerge. Fiddling around with these frameworks and middleware, successfully making the authentication work as expected, we discovered that authentication-related bugs could roughly be categorized into a few essential types of vulnerabilities.

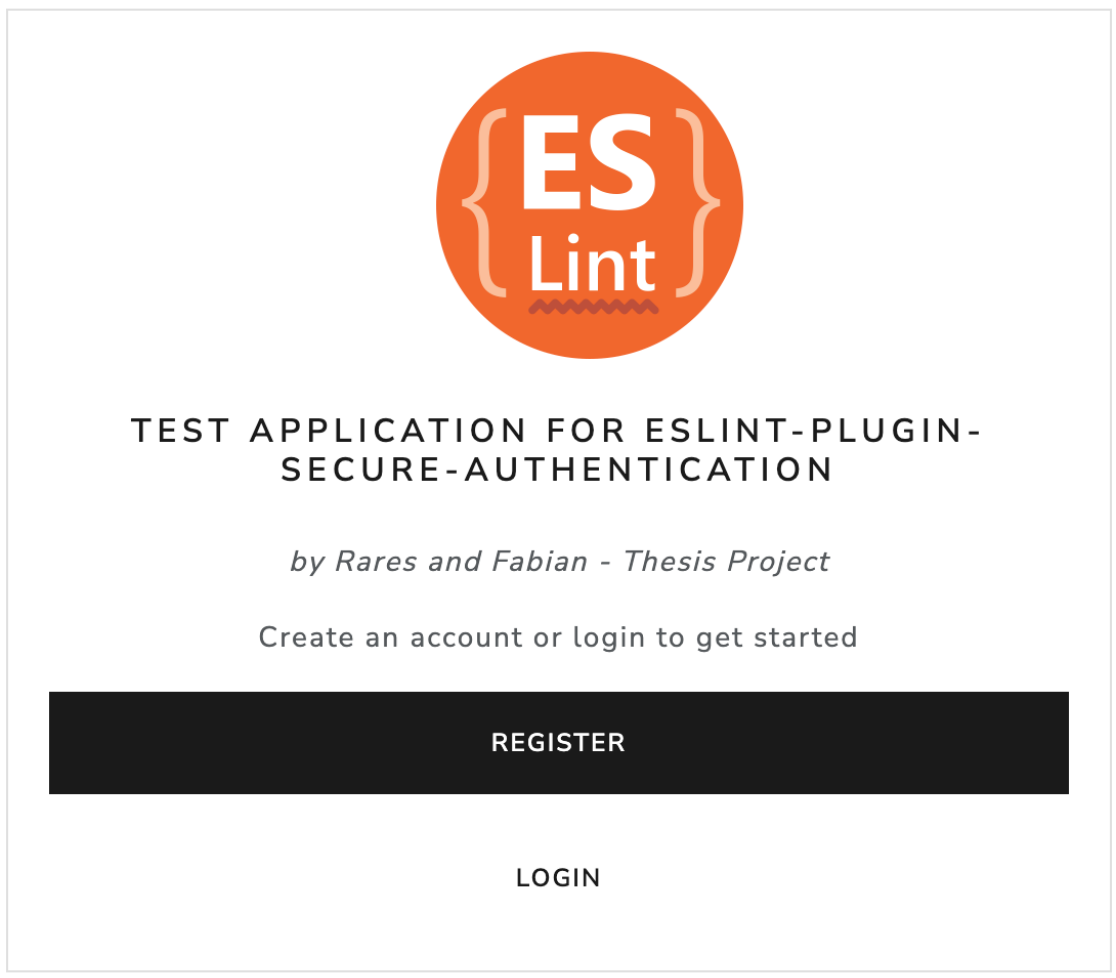


Figure 1: Login/register using node/express with passport js

3.3 Authentication Bugs on the Web

3.3.1 Password related vulnerabilities

User authentication in computing systems traditionally depends on three factors: something you have (e.g., a hardware token), something you are (e.g., a fingerprint), and something you know (e.g., a password). [6] The password remains the most common way to authenticate among the authentication options (knowledge, possession, location, biometric). Meanwhile, as Brainard et al. state in their paper: The problem of identifying yourself with "Something you know" is that it relies on users' behaviour and selection of information that can be unpredictable and insecure as most people may pick a password that is easy to remember, but also likely to be guessed by an attacker. Needless to say, the password, i.e. the most prevalent type of "key" to accessing computer systems, is a source of various vulnerabilities. The most common mitigation strategy that also will enhance security is the requirement of a *password policy* upon registration. The idea is that if

passwords are strong enough, i.e. a string that in combination has the properties: 8 $\geq$ characters, uses upper and lower case, and special characters, they become consequently a lot harder to break for hackers. After setting up the basic authentication application, this seemed a fairly obvious pattern for a rule to identify the presence of such validation and, as a fix, include it in the login route. Another observation we made is that in the context of saving the submitted password to MongoDB, it seems fairly unrestricted, and no requirement of how the password is stored is mandatory within any of the used frameworks. Therefore a rule detecting *plain text passwords* that are attempted saved to the database and hashed beforehand by an automatic fix seemed like a suitable condition for another customized rule to implement. Another type of safety measure and candidate for a rule we thought about was *MFA (multi-factor authentication)*, but despite the proven level of security it adds to authentication, it seems for this moment to be a bit too complex of a task for a custom ESlint rule to implement successfully, and maybe more of a recommended layer of security rather than an actual violation of best practice or bug.

3.3.2 Brute force and DoS related vulnerabilities

Brute force attacks use computational power to generate large amounts of illegitimate login requests. This will heighten the chances of eventually guessing the correct password/username combination, while Denial of service attacks would aim to disrupt the system with an abundance of HTTP requests causing the server to be unavailable for legitimate user requests. We discovered that both kinds of attacks could be conducted using standardized authentication in the express/node environment earlier accounted for. No built-in limiter of any kind is configured by default in any of the frameworks. However, it seems that numerous libraries like express-brute (brute force prevention) and express-rate-limit (DoS prevention) do provide a straightforward implementation to mitigate excessive requests of this nature. For that reason, we moved forward with these authentication bugs and aimed to detect the absence of such limiter implementation in our code and let the automatic fixes enforce the application to prevent such attacks by including a generic configuration and requiring it as a middleware callback function in critical routes. [17]

3.3.3 OAuth Implicit Flow

The OAuth 2.0 is a standard that deals with authorization. More specifically, it is a collection of different protocols that describe processes through which third-party applications can gain access to a user's resources without actually having to get the user's credentials. These protocols differ slightly in terms of the process flow, however, the idea is basically the same - clients request and subsequently manage access tokens in a supposedly secure manner. The main idea behind OAuth 2.0 is that the users, or the resource owners, are able to maintain their privacy intact while, at the same time, they are able to grant access to certain information. OAuth 2.0 was developed by Internet Engineering Task Force (IETF) and is described in RFC 6749.

As mentioned in [13], the four main roles involved in the process are:

- **Resource Server**: It represents the server that holds the resources owned by the resource owner. In the process flow, the clients send the resource server the access tokens, afterwards, the resource server is responsible for validating the token. If the validity is confirmed, then it provides the requested resources.

- **Client**: It represents the entity that has to get authorized for accessing the resources of the resource owner. The client can be a web application, mobile application, or

any other software that needs to access the resource owner's resources.

- **Authorization Server**: It represents the entity which deals with the authentication of the resource owner. The authorization server also issues the access token to the client.

- **Resource Owner**: It represents an entity that is the owner of a particular resource that is being protected. The Resource owner is usually the entity that provides permission to a given service A (the client) to access their resources at service B (the resource server)

We can observe the summarized interaction between the four roles below, as mentioned in [13]:

- **Authorization Request** In this step, the client asks for authorization from the resource owner. The authorization request can also be made indirectly through the Authorization Server.

- **Authorization Grant** The resource owner sends the client an authorization grant, which is a credential that is the authorization. Afterwards, the client sends the authorization grant to the authorization server to request an access token.

- **Access Token** After authenticating the client and validating the authorization grant, the authorization server issues an access token to the client. Afterwards, the client uses the access token to request the protected resource from the resource server.

- **Protected Resource** Finally, the resource checks to see if the access token is valid and, depending on the validity, sends the protected resource to the client.

The steps mentioned above are represented in the following figure 2:

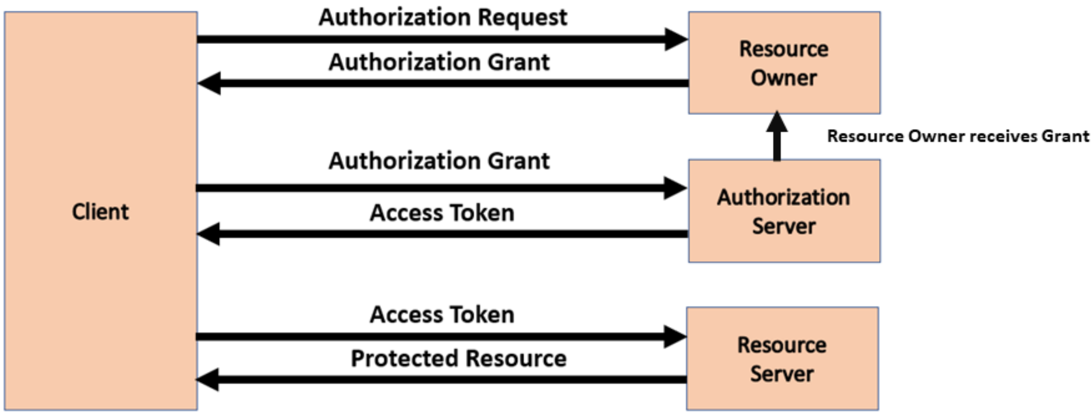


Figure 2: OAuth 2.0 flow

To give an example, let's assume that a user wants to log into a third-party website such as StackOverflow by using their Google credentials. In order to do this, the user will be redirected to Google's login page. In this context, Google is the Authorization Server that verifies the credentials. If the user is authenticated successfully, the Authorization Server will ask for confirmation in sharing certain data - for example, e-mail information and profile picture. Once the user/Resource Owner approves the sharing of information, the Authorization Server (Google) sends an access token to StackOverflow/the Client (the application that needs access to the protected resource). Once the Client has access to the token, it can use it in order to request specific data (e-mail and profile picture) from Google. In this case, Google is the Resource Server (as well as the Authorization Server). The Resource Server then validates the token received from the Client and eventually sends the data back to StackOverflow.

Some important notes are that the Authorization server doesn't always have to be the Resource Server. They can also be separate entities. It is also possible that one Authorization Server could create tokens that more than one Resource Server can accept.

The Implicit flow in OAuth 2.0 is designed for clients within browsers. It streamlines the process by directly issuing an access token without the need for intermediate credentials, such as an authorization code. Although this method enhances the responsiveness and efficiency of certain clients, it is important to be aware of potential security risks that might arise, especially when the authorization code grant type is available and which we discuss more in our discussion section. This flow does not involve client authentication, and the access token might be exposed to the resource owner or other applications accessing the resource owner's user-agent [13].

The implicit flow can be summarized in the following steps taken from the example in [13] :

- **Client Identifier and Redirection URI** The first step of the flow starts with the client directing the user-agent (browser) to the Authorization Server. When the client does this, several parameters are included in the process - the client identifier, the requested scope, the local state and the redirection URI. If the authorization server grants access, it will redirect the user agent to the specified redirection URI.

- **User Authenticates** In this step, it is established if the client receives an access request or not from the authorization server.

- **Redirection URI with Access Token in Fragment** In case the authorization server authorizes the resource owner successfully, the user-agent will be redirected to the client by the Authorization Server. This is done by using the redirection URI which now includes the access token.

- **Redirection URI without Fragment** When the authorization server issues a redirect response, it provides a URL to which the user-agent (in most cases, a web browser) is intended to navigate to. This URL includes a fragment part, which contains the access token. However, when the user-agent makes a request to this redirected URL, it does not include the fragment part in the request sent to the server. What it does instead is that it keeps the fragment that holds the access token stored locally - usually in its memory. The user-agent is then able to use this locally-stored access token for subsequent interactions with the server.

- **Script** When the browser navigates to the redirect URL - that doesn't have the fragment, the server at that URL sends back a web page. This web page typically

includes some script embedded in it. This script is written in such a way that it can access the full URL of the page, including the fragment that was retained by the browser but not sent to the server. By accessing this full URL, the script can extract the access token and any other parameters that were included in the fragment.

- **User-agent** The browser runs the script that was included in the web-hosted client resource it received. This code is run locally, on the user's machine, and its job is to extract the access token from the full URL (including the fragment).
- **Access token** The user-agent further passes the access token to the client.

Let's assume another made-up example:

A user tries to log into a music application with Google. The application sends a request to the authorization server (Google) and it includes in it its client ID, the scope of access it wants (such as the user's email), and the redirection URI (a URI in the music application). Thereafter the user is redirected to Google's login page. The user logs in and then Google asks consent to grant the music application permission in order to access details from the user's Google account. Since, in this example, Google is the authorization server, it generates an access token and then appends it to the redirection URI as a fragment, and then redirects the user's browser back to the music application. The user's browser makes a request to the redirection URI without the fragment. The browser retains the access token in the fragment locally. The music application returns an HTML page that includes code that is capable of accessing the full redirected URI, including the fragment. The user's browser executes the code embedded in the HTML page and extracts the access token from the fully redirected URI. Finally, the browser passes the access token to the music application, which can use the token to make requests on behalf of the user to Google's API's.

The way we decided to fix the security issue regarding the Implicit Grant Flow is by warning the user that the use of this particular grant has been detected. In OAuth 2.0, the `"response_type"` parameter is indeed crucial in determining the kind of OAuth flow to be implemented. For our rule, it is easy to come up with a fixed solution given that a different grant type (flow) - Authorization Code, uses the same parameter but with a different value `response_type=code`. Given that changing the value of the `"response_type"` parameter to "code" seems to be the easiest approach in trying to help the developer to use a safer grant - Authorization Code Grant - our rule attempts to do just that.

The reason why Authorization Code Grant is a better alternative to the Implicit Grant Flow is that the client's credentials are never exposed to the user or the user's browser. The access token is returned from the authorization server to the client directly, not through a redirect, reducing the risk of it being intercepted in transit.

There is more that needs to be done for implementing Authorization Code Grant than just changing the value to `response_type=code`. The task is hardly trivial and is possibly unfeasible.

According to RFC6749, certain additional steps need to be taken in order to follow the Authorization Code Grant. For example, Client Authentication has to be done, which would imply the exchange of an authorization code that the client has received after the user has authorized with the actual access token. Making such a change usually also requires the use of client secret value that must not be exposed to client-side code [13]. Basically, the Authorization Code flow would require additional code that would handle the exchange of the authorization code with the access token. Additionally, there would be other changes required in the given application - handling of redirection with the authorization code, having to make an HTTP POST request to exchange the code for the

token and afterwards having to handle the token itself. Having a rule that would be able to fix these issues automatically could introduce more bugs than solutions. Nevertheless, warning the developer that switching from the Implicit flow to the Authorization Code flow could be a good step for improving security is mainly what our rule handles.

We will now give an overview of how the Authorization Code Grant is assumed to work and then provide an example of what our ESLint rule could ideally implement as a fix. The typical Authorization Code Grant flow starts with the client redirecting the user-agent (typically the browser) to an authorization endpoint. This will make the user (resource owner) authenticate and authorize access. There are several elements that the client has to include throughout this process:

- A unique identifier called the "client identifier" which the authorization server assigns to the given client when it is initially registered.
- A "requested Scope" which represents the permissions that the given application is asking for. It could be that the application is asking for permission to have access to a user's private picture album.
- A redirection URI which will be used by the authorization server to redirect the user after they granted/denied the access.

Secondly, the flow continues with the authorization server authenticating the resource owner and having to establish if they grant or deny the client's access. Next, the user-agent is going to be redirected by the authorization server, back to the client through the redirection URI, and it will include an authorization code as a query parameter. Next in the process, an exchange will take place between the client and the authorization server. In order to receive the access token, the client will make a request to the authorization server's endpoint. By including the access code and the redirect URI in the request, they will be verified against their originality/authenticity. Lastly, the authorization server will ensure the validity of the authorization code and that of the redirect URI and eventually responds back with the access token.

3.3.4 Cookie related vulnerabilities

Cookies are pieces of data that a user's browser receives when visiting certain websites. Their purpose is to ease the user experience by remembering certain information. Doing so can make visiting and interacting with certain websites, a smoother experience [12]. The user's browser can store a certain cookie such that it can be used later on for the same server. According to [24], the general purpose of cookies can fit into three categories:

- **Session management**: Keeping track of log-in information; keeping track of purchases; server load balancing;
- **Personalization**: User preferences; language settings, themes, etc.
- **Tracking**: Behavioral advertising; analytics; social media.

According to the Expressjs documentation, the 'res.cookie()' method is used to set cookies on the client-side (user's browser) when handling HTTP responses. The method sets a cookie name to a value. The value parameter may be a string or object converted to JSON. The options parameter is an object that can have certain properties [35]. The method within the Express framework has the following format:

```
res.cookie(name, value [, options])
```

The options parameter is an object that can have multiple properties with different effects on the cookie. According to Expressjs, one of the best practices for Express applications is to use cookies securely.

We will discuss these three properties, and we will expand on the necessity of having them present in the method, from a security perspective:

1. `maxAge`: The Max-Age attribute determines the maximum duration of a cookie's existence. The duration is measured in milliseconds and after the time limit passes, the cookie itself expires [4, p. 20]. If the property isn't specified, the browser will view it as a session cookie and delete it when the browser closes. Generally, the cookie's purpose determines the time-out property's duration. From a security standpoint, having a lengthy max-age value can present risks when using devices shared among multiple users. For instance, if a user logs into a website on a shared computer and leaves without logging out or closing the browser, it means the cookie stays on the device. If another user uses the same computer, they could potentially access the website with the previous user's session, possibly leading to unauthorized access to sensitive information, account breaches, or other security problems. Assigning a suitable value for this property helps to limit the cookie's lifespan, thus minimizing the chances of potential attacks. [16].

2. `httpOnly`: A security concern involving cookies is CWE-1004, which refers to sensitive cookies that are missing the 'HttpOnly' attribute. A developer could include the HttpOnly attribute in the Set-Cookie HTTP response header. This would help with reducing the risk associated with Cross-Site Scripting. In XSS attacks, an attacker could create a malicious script that could have the potential of extracting important data from a cookie and thus create a security breach [8]. The HttpOnly attribute ensures that cookies will not be accessed through JavaScript client code. The attribute ensures that cookies will be transmitted only through HTTP or HTTPS requests. This is therefore a safeguard against XSS attacks. [5]. In simpler terms, the HttpOnly attribute ensures that browsers only include the cookies when making requests over HTTP or HTTPS, and prevent access to the cookies through non-HTTP APIs, such as those used by client-side JavaScript [16].

3. `sameSite`: This property controls if a cookie is shared with requests between various websites, providing some protection against cross-site request forgery (CSRF) attacks. CSRF attacks exploit the fact that browsers automatically send all relevant cookies when making requests, including authentication cookies. When a user is logged into a website, their authentication cookie is included with their requests. A CSRF attack takes advantage of this by tricking the user into making a request to the target website from another site while still including the authentication cookie. Because the authentication cookie is included in the forged request, the target website cannot easily differentiate between a legitimate request made by the user and a fraudulent one made by an attacker [28]. The sameSite property can have different values, however, our rule adds the missing property with the value "Strict". This particular value implies that the browser sends the cookie only for the same site that has set the cookie in the first place. In case the request is made from a different scheme or domain (this also applies in case the domain has the same name), then

the cookies that have the SameSite attribute set as Strict won't be sent. [33]. For example, on a given website, if a user is logged in and the sameSite attribute is enabled if they attempt to follow a link to a private GitHub repository that has, by chance, been listed somewhere in a discussion forum, then Github will not receive that specific session cookie and, therefore, the given user won't be able to access that particular repository. Having the attribute enabled could also be very useful if, for example, in the context of a bank website, it wouldn't be desired that certain information or pages are linked to external sites [28].

3.3.5 Cross-Site Request Forgery (CSRF)

One of the most popular ways to combat CSRF attacks is by implementing CSRF tokens. This is also known as Synchronizer Token Pattern (STP) [20]. The diagram below 3 shows how the flow for this pattern is supposed to behave like. Due to the complexity of implementing such a rule, we were able to create a rule that mainly tackles the second step of the flow. More will be discussed on this regard in the 4.4.2 section that deals with the fixing part of our rule.

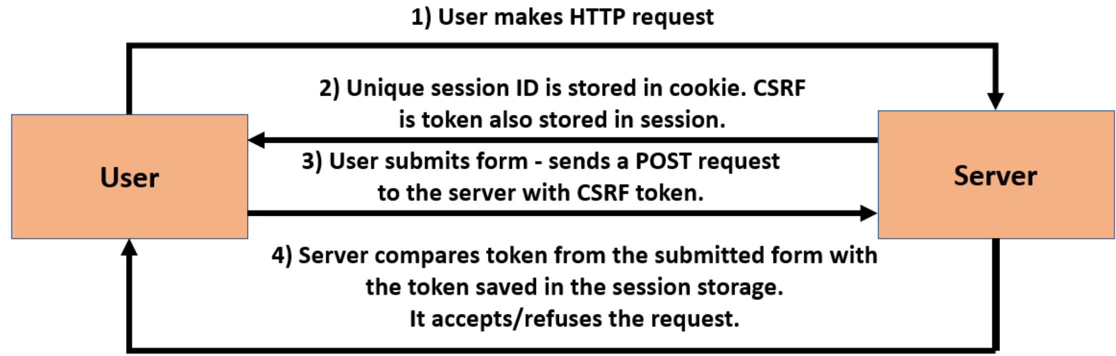


Figure 3: STP Flow

Even though CSRF tokens have been built for different frameworks in different ways, including express.js (which we use for our ESlint rule), the systems behind the CSRF tokens can be described in a very straightforward manner. Such tokens should usually be created on the server side. They can be created per request or per session, the latter being less secure given that a potential attacker would have a much longer time to exploit a stolen token (our rule creates a per-request token). When the user makes an HTTP request through the client side, the server side will verify if the CSRF token that is present in the request matches the token present in the user session. The server will deny the HTTP request if a token is not present in it or if it doesn't match the user session. In short, the server sends the client a token, the user submits a form or request with the token, and then the server approves or rejects the incoming request if the token is invalid [28].

3.4 Related Work

In their research, "Fixing Vulnerabilities Automatically with Linters" [31] Rafnsson, Giustolisi, Kragerup, and Høyrup from the IT University of Copenhagen explored the potential of linters as static analysis tools for detecting and fixing vulnerabilities in code. In their research, they focus on ESLint as a linting tool, and they also create new rules that detect and fix vulnerabilities prone to attacks - more specifically, cross-site scripting (XSS), SQL injections and configuration vulnerabilities. In their research, through their rules, they have demonstrated the potential that linters can have in contributing to more secure code, as new security rules could be created. Additionally, they also outline that analysing dependencies for vulnerabilities are rather important since it could make the code within an application more prone to attacks. They also outline some important limitations in relation to using ESLint for security purposes. Firstly, the programmers need to trust the rules that they are getting are not harmful. Secondly, ESLint is able to scan only one file at a time, meaning that ESLint will not be able to scan the library dependencies. They also make a case that if each file was scanned independently and found to be without vulnerabilities, there would still be a potential risk since in itself, the way the modules that make up the application interact with each other might create vulnerabilities. They also recognise the importance of having clear guidelines when it comes to creating new rules and also the importance of having the linting rules maintained by the community itself.

For our research, Rafnsson et al.'s work truly provide an important framework. Similar to their aim in creating ESLint rules, our research focuses on the same aspect - creating rules that are able to identify and fix security vulnerabilities. Their findings on linters in relation to using rules for mitigating security vulnerabilities provide helpful insight for our own research and how to approach the potential arising challenges.

In their Master Thesis "Maximal security with minimal effort" [1], Agersten and Madsen made further advances in the field of automated vulnerability detection and fixing through ESLint. Through their work, they have introduced 11 new rules, out of which 7 are able to provide automatic fixes - further proving the findings of Rafnsson et al. and also suggesting that the existing rules could be further optimized. In their research, they also point out that further evaluation of SAST tools needs to be made and that the Open Source Security Foundation's (OpenSSF) test suite could actually aid in this direction, while it is also extended further with this purpose.

Agersten and Madsen's work offers additional insight into the potential of linters in managing security vulnerabilities. This evidently puts emphasis on the importance of further developing and refining linter rules.

4 Implementation

4.1 Cookie Safe Attributes

4.1.1 Examples of attacks due to missing cookie attributes

The code below is an example taken exactly from CWE - 1004 [8], and describes a successfully made XSS attack. A new cookie is created and it misses the `httpOnly` attribute which would generally prevent the attacker from using client-side scripts to access them:

```
1 String sessionID = generateSessionId();
2 Cookie c = new Cookie("session_id", sessionID);
3 response.addCookie(c);
```

The attacker can just create a malicious script that would freely access the `document.cookie`.

```
1 document.write('<img src="http://attacker.example.com/collect-cookies?
      cookie=' + document.cookie . '">'
```

The code below is an example taken exactly from CWE - 1275 [9], and describes a successfully made CSRF attack.

A new `sessionId` is created and is being sent to the client browser.

```
1 let sessionId = generateSessionId()
2 let cookieOptions = { domain: 'example.com' }
3 response.cookie('sessionid', sessionId, cookieOptions)
```

Since the `sameSite` attribute was missing, the cookie will be included in every request made to the server, including cross-site requests. So in the example below, the user that received the cookie, could navigate to an ”evil” website and submit a form and send its cookie in the request to the official website that will not see anything wrong with it.

```
1 <html>
2     <form id=evil action="http://local:3002/setEmail" method="POST">
3     <input type="hidden" name="newEmail" value="abc@example.com" />
4     </form>
5     <script>evil.submit()</script>
6 </html>
```

4.1.2 Finding the Vulnerability

Our rule goes through all the function calls `CallExpression` if the function that is called is named `res.cookie` - this being a typical function to set a cookie in Express.js - then our rule will check to see if the third argument is passed into the function. The third argument represents the options object and it can have multiple properties. Our rule will find the functions that don't have or `maxAge`, `httpOnly`, and `sameSite` properties. Let's consider the code below that is missing a third parameter. It only contains the name and the value of the cookie. In this particular case, our rule will identify that the parameter with the options is missing and it will trigger a warning.

```
1 app.post("/login", (req, res) => {
2   res.cookie("session", "dummySession");
3   res.send("Logged in");
4 });
```

In figure 4, it can be seen that our rule runs on the premise that the node must have at least three arguments (the name of the cookie, the value and the options parameter). The rule also checks to see if the type of the `options` attributes is that of `ObjectExpression`. If both conditions are met, we will save `optionsIndex` with the value `2` such that we will later use the index value to store the right attributes within the actual `options` parameter. In order to clarify any confusion, in reality, we could have other objects as parameters that are `ObjectExpression` - we don't know for sure if we will get the `options` attribute. A developer could add any other parameter for whatever given reason, however, it is expected to be the `options` object, according to the API provided by the Express.js library.

```
CallExpression(node) {
  if (
    node.callee.type === "MemberExpression" &&
    node.callee.property.type === "Identifier" &&
    node.callee.property.name === "cookie"
  ) {
    const optionsIndex = node.arguments.length >= 3 ? 2 : -1;
    const options =
      optionsIndex !== -1 && node.arguments[optionsIndex].type === "ObjectExpression"
        ? node.arguments[optionsIndex]
        : null;

    let hasExpiresOrMaxAge = false;
    let hasHttpOnly = false;
    let hasSameSite = false;
```

Figure 4: Finding Parameter in `res.cookie()`

Next, in our rule 5, we check that the expected options are in place, and we flag them as `true`. If any of them are missing, then our rule will warn that they have to be implemented.

```
if (options) {
  options.properties.forEach((property) => {
    if (
      property.type === "Property" &&
      property.key.type === "Identifier" &&
      (property.key.name === "Expires" || property.key.name === "maxAge")
    ) {
      hasExpiresOrMaxAge = true;
    }

    if (
      property.type === "Property" &&
      property.key.type === "Identifier" &&
      property.key.name === "httpOnly"
    ) {
      hasHttpOnly = true;
    }

    if (
      property.type === "Property" &&
      property.key.type === "Identifier" &&
      property.key.name === "sameSite"
    ) {
      hasSameSite = true;
    }
  });
```

Figure 5: Checking Options

4.1.3 Fixing the Vulnerability

In terms of fixing the vulnerabilities, our rule takes into consideration three possible variants. The first two conditions can be seen in figure 6.

In the first condition, if the `option` value is `null`, then we will add the attributes at the end of the arguments list. Once again, we are running the rule on the premise that the developer will add three attributes to `res.cookie()`. In the second condition, we are just covering for the possibility that the options argument could exist, but it could be empty. If that is the case, then we add the missing attributes.

```
if (!hasExpiresOrMaxAge || !hasHttpOnly || !hasSameSite) {
  context.report({
    node,
    message: "Missing secure attributes in res.cookie options",
    fix(fixer) {
      const fixes = [];

      if (!options) {
        fixes.push(
          fixer.insertTextAfter(
            node.arguments[node.arguments.length - 1],
            ', { maxAge: 15 * 60 * 1000, httpOnly: true, sameSite: "Strict" }'
          )
        );
      } else if (options.properties.length === 0) {
        fixes.push(
          fixer.replaceText(
            options,
            '{ maxAge: 15 * 60 * 1000, httpOnly: true, sameSite: "Strict" }'
          )
        );
```

Figure 6: Fixing Conditions (1)

In the third condition 7, if a specific property in the `option` parameter is, missing - if it has been flagged as `false`, then it will be added at the end of the list.

```
} else {
  const lastOption = options.properties[options.properties.length - 1];
  if (!hasExpiresOrMaxAge) {
    fixes.push(
      fixer.insertTextAfter(
        lastOption,
        ', maxAge: 15 * 60 * 1000'
      )
    );
  }

  if (!hasHttpOnly) {
    fixes.push(
      fixer.insertTextAfter(
        lastOption,
        ', httpOnly: true'
      )
    );
  }

  if (!hasSameSite) {
    fixes.push(
      fixer.insertTextAfter(
        lastOption,
        ', sameSite: "Strict"'
      )
    );
  }
}
```

Figure 7: Fixing Conditions (2)

4.2 OAuth 2.0 Implicit Flow

4.2.1 Possible dangers regarding the use of Implicit Flow

For describing what vulnerabilities the use of Implicit Flow brings into an application, we must outline that two important steps are omitted from the protocol. In the implicit flow, there is no separate authorization grant step, such as the issuance of an authorization code. Instead, the access token is issued directly by the authorization server in the response after the resource owner authorizes the client. As there is no authorization grant step in the implicit flow, the client doesn't need to make a separate request to the authorization server to exchange the authorization grant for an access token.

These changes in the protocol may result in certain security issues. According to [13], it is an absolute necessity for access token credentials to be kept confidential both in terms of storage but also transit. The access tokens should be shared only throughout the relevant parties, these being the ones described above as well - the authorization server, the resource servers as well as the client for which the access token was generated in the first place. The transit of the token should be done using TLS(Transport Layer Security) with server authentication. It is important to note, however, that the main security risk with the Implicit flow is not the transmission of the access token over the network (since it should be over TLS), but rather the exposure of the access token in the URL and in the browser history, and the potential for access tokens to be leaked to unauthorized parties.

Added to this, the lack of client authentication in the implicit flow means that there would be no way for the authorization server to verify that the client receiving the access token is the one for which it was meant to be given in the first place. In Implicit Flow, the access token is returned directly to the URL. If the redirection process is compromised by a potential attacker, the URL could be intercepted and therefore the access token could be obtained and used with malicious intent.

4.2.2 Finding the Vulnerability

This ESLint rule is designed to warn developers when their JavaScript code may be using the Implicit OAuth 2.0 flow. It does this by checking for the string `response_type=token` in the code. Since the `token` value is used in the Implicit flow, then the rule will know for sure that a vulnerability indeed exists.

Since in JavaScript, we have two ways of defining strings - string literals and template literals, we had to create two functions for finding the vulnerabilities. These two types of strings are represented in the abstract syntax tree (AST) differently. String literals are represented through Literal nodes and template literals are represented through TemplateLiteralNodes.

In the following figure 8, when the user navigates to the `/login` endpoint on the server, they will be redirected to Spotify's account authorization page where they should log in and consent for the necessary credentials to be used on the main application. In the picture, it can be seen that the type of grant has been found in a string literal and it is Implicit Grant Flow, since `response_type=token` is a specific parameter. The code with the vulnerability has been underlined in red.

```
32  app.get('/login', (req, res) => {
33    // Redirect the user to the authorization page
34    const scope = 'user-read-private user-read-email';
35    res.redirect('https://accounts.spotify.com/authorize' +
36      '?response_type=token' +
37      '&client_id=' + clientId +
38      (scope ? '&scope=' + encodeURIComponent(scope) : '') +
39      '&redirect_uri=' + encodeURIComponent(redirectUri));
40  });
41
```

Figure 8: Found Vulnerability `response_type=token` in String Literal

In this example 9, the same vulnerability `response_type=token` is found in a template literal.

```
52  app.get('/login', (req, res) => {
53    // Redirect the user to the authorization page
54    const scope = 'user-read-private user-read-email';
55    res.redirect(`https://accounts.spotify.com/authorize?response_type=token&client_id=${clientId}${scope
56  });
57
```

Figure 9: Found Vulnerability `response_type=token` in Template Literal

The code in the figures fulfils the same role the only difference is that the first vulnerability has been found in an example that uses concatenation and normal strings and the second one uses template literals.

In order for our rule to find the vulnerability, we use the `checkLiteral` function to see if there exists a literal node in the code that contains the string `response_type=token`. The `checkLiteral` function only checks for string literals, not template literals.

The following figure 10 shows the process of searching for the mentioned nodes.

```
create: function (context) {
  function checkLiteral(node) {
    if (typeof node.value === 'string') {
      const tokenIndex = node.value.indexOf('response_type=token');
      if (tokenIndex !== -1) {
        context.report({
          node,
          message: 'response_type=token detected in the code, which indicates the use of the Implicit OAuth 2.0 flow.',
```

Figure 10: Finding Literal Node with String `response_type=token`

For finding the template literals, we used the `checkTemplateLiteral` function. More specifically, for finding the presence of strings in template literals.

The following figure 11 shows the process of searching for the mentioned nodes.

```
create: function (context) {
  function checkLiteral(node) {
    if (typeof node.value === 'string') {
      const tokenIndex = node.value.indexOf('response_type=token');
      if (tokenIndex !== -1) {
        context.report({
          node,
          message: 'response_type=token detected in the code, which indicates the use of the Implicit OAuth 2.0 flow.',
```

Figure 11: Finding Template Literal Node with string `response_type=token`

4.2.3 Fixing the Vulnerability

After finding the vulnerability, the developer can fix the issue. More specifically, our rule simply changes the string value from `token` to `code`, encouraging the developer to use/implement the Authorization Code Grant.

Our fix is used inside the `checkLiteral` function and can be seen in the following figure 12. What it does is that the function constructs a new string that is identical to the original string `(node.value)`, except that `response_type=token` is replaced with `response_type=code`. This new string, `fixed`, is created by slicing the original string into two parts: one before `response_type=token` and one after it, and then concatenating `response_type=code` in between - that would be so that the original part of the string is kept intact with the exception of the new value of the parameter.

The code for this is:

```
1   node.value.slice(0, tokenIndex) + 'response_type=code' +
2   node.value.slice(tokenIndex + 'response_type=token'.length).
```

Finally, the function calls `fixer.replaceText(node, '$fixed')` to replace the original string literal in the source code with the new string.

```
fix(fixer) {
  const fixed = node.value.slice(0, tokenIndex) + 'response_type=code' + node.value.slice(tokenIndex + 'response_type=token'.length);
  return fixer.replaceText(node, `'${fixed}'`);
},
```

Figure 12: Fixing Literal Node into `response_type=code`

A complete fix for our ESLint rule would mean adding the remaining implementation of the flow, which deals with handling the exchange between the access code and the actual token.

4.3 Cross-Site Request Forgery (CSRF)

4.4 Describing the Vulnerability

Cross-Site Request Forgery (CSRF) is a type of attack through which a website user can be manipulated to perform malicious actions unwillingly and unknowingly. The victim can be tricked into performing such actions by accessing malicious links that may have been sent through e-mail or chat. If it happens that the victim is logged into a certain application, the CSRF attack could lead to actions or requests that modify the data or configuration of a user's account or the web application itself. These actions lead to a change in the "state" of the application, which means that the conditions or settings of the application are altered - cases in which the entire web application could be compromised. Other than phishing methods, it is also possible that certain CSRF attacks are hidden on vulnerable websites (stored CSRF flaws). These links can be stored in "img" or "iframe" tags, but not only. These elements will be rendered on the client side. An unsuspecting victim will be even more trustworthy of certain redirecting links, given that they were accessing a website that they were familiar with in the first place. Added to this, the attack can be even more problematic since the user victim is most likely also already authenticated into the website - therefore, when clicking the malicious link, no further authentication would be required [27]. A more concrete example would be that a form that submits certain information is executed as soon as the web page is rendered. This can be done when malicious code is added to the website (as seen in the example below).

```
1 <form method="post" action="transferMoney">
2   <input name="totalSum" type="hidden" value="999" />
3   <input name="receiver" type="hidden" value="hacker" />
4 </form>
5 <script>document.forms[0].submit()</script>
```

The main idea behind a CSRF attack is that a forged request is made on the client side. In the example above, this is a POST request, however, it can be any type of HTTP request, depending on what is intended to be achieved by the attack. The attacker will trick the victim's browser into sending a request to the server side and therefore succeed with the attack. Since an HTTP cookie is required to authorize certain requests or to tell that the requests come from the same browser - the attacker would also need to have the victim authorized when navigating a session [25].

4.4.1 Finding the Vulnerability

Our code defines an ESLint rule that identifies if CSRF protection is being used or not on the server side of an application and then suggests a fix if appropriate. To be more precise, it looks through the code and figures out if the `csrf` string is being imported through the `require()` function.

As we can see in figure 13, in the first part of the rule we use the two variables:

```
1     let csrfImported = false;
2     let httpRequestNodes = [];
```

We use them in order to keep track of whether the CSRF package is imported or not as well as to store the nodes that represent the HTTP request methods.

```
13  create(context) {
14    let csrfImported = false;
15    let httpRequestNodes = [];
16
17    return {
18
19      VariableDeclaration(node) {
20        node.declarations.forEach((declaration) => {
21          if (
22            declaration.init &&
23            declaration.init.type === "CallExpression" &&
24            declaration.init.callee.name === "require" &&
25            declaration.init.arguments.length > 0 &&
26            declaration.init.arguments[0].value === "csrf"
27          ) {
28            csrfImported = true;
29          }
30        });
31      },
```

Figure 13: CSRF Rule - Variable Declaration

In the second part of the rule, we use a listener function `VariableDeclaration` that checks each variable declaration in the source code, in order to detect if the CSRF package is imported using the required function. When the package is imported, the `csrfImported` variable is set to true.

In the following figure 14 we can see that since the CSRF module has not been found as imported with a statement such as `const csrf = require('csrf');`, our rule fires up a warning that HTTP requests are being made without having the middleware imported.

```
82  CSRF protection (csrf) is not imported but HTTP requests are being made
83  const app: Express
84
85  View Problem (Alt+F8)    Quick Fix... (Ctrl+.)
86  app.post("/dummy-post", (req, res) => {
87    res.send("Dummy POST request");
88  });
```

Figure 14: CSRF Rule - Finding Lack of CSRF token protection

In the third part of the rule 15, the `CallExpression` listener collects all HTTP request method nodes (GET, POST, PUT, and DELETE) by checking if the callee is a `MemberExpression` with an object named `app` and a property name that matches one of the HTTP request methods. The matched nodes are then stored in `httpRequestsNodes`.

The nodes will be analysed further on so as to determine whether there are any issues that should be reported. The nodes that represent the HTTP requests and have been saved in `httpRequestNodes` will be used so as to trigger the warnings that `csrf` is not being imported.

```
32        CallExpression(node) {
33          if (
34            node.callee.type === "MemberExpression" &&
35            node.callee.object.name === "app" &&
36            ["get", "post", "put", "delete"].includes(node.callee.property.name)
37          ) {
38            httpRequestNodes.push(node);
39          }
40        },
```

Figure 15: CSRF Rule - Nodelist

If CSRF protection is imported, the rule reports a warning for each HTTP request node to remind the developer to use the `tokens.verify()` method in the HTTP requests so as to validate the CSRF token as seen in figure 16.

```
if (csrfImported) {
  httpRequestNodes.forEach((httpRequestNode) => {
    context.report({
      node: httpRequestNode,
      message: "Warning: CSRF tokens are being generated but you still need to use "
      + " tokens.verify() in order to validate the token for each HTTP request.",
    });
  });
}
```

Figure 16: CSRF Rule - Warning for Verification

4.4.2 Fixing the Vulnerability

The fix within our rule happens in the last part of the rule. `Program:exit` listener is triggered when the ESLint traversal is complete. It checks if CSRF protection is imported and performs two different actions based on this value.

If CSRF protection is not imported, the rule reports a warning for each HTTP request node and provides a fix that inserts the necessary code to set up CSRF protection as seen in figure 17. If the middleware is imported, then it will trigger a warning that `tokens.verify()` still needs to be used in order to validate the token for each HTTP request.

```
"Program:exit": function (node) {
  if (!csrfImported) {
    httpRequestNodes.forEach((httpRequestNode) => {
      context.report({
        node: httpRequestNode,
        message: "CSRF protection (csrf) is not imported but HTTP requests a
        fix(fixer) {
          const csrfLines = `const express = require('express');
          const session = require('express-session');
          const csrf = require('csrf');
          const bodyParser = require('body-parser');
          const crypto = require('crypto');

          const app = express();

          const secret = crypto.randomBytes(16).toString('hex');
          const tokens = new csrf({ secret });

          app.use(session({
            secret: 'session secret',
            resave: false,
            saveUninitialized: true,
            cookie: { secure: true } //
          }));

          app.use(bodyParser.urlencoded({ extended: false }));

          app.use((req, res, next) => {
            if (!req.session.csrfToken) {
              req.session.csrfToken = tokens.create(secret);
            }
            res.locals.csrfToken = req.session.csrfToken;
            next();
          });

          const firstNode = context.getSourceCode().ast.body[0];

          return fixer.insertTextBefore(firstNode, csrfLines);
        },
```

Figure 17: CSRF Rule - Adding Code

The reason why the rule only warns about the need to use `tokens.verify()` in the HTTP request and doesn't actually provide a fix is because the full fix implementation of the csrf middleware (verification/validation) is quite non-trivial and it is highly dependent on the unpredictable coding patterns that developers might use in their applications.

Even then, we were able to create a rule that when it finds the potential vulnerability (not using CSRF token on HTTP requests), it generates the token. It would still be assumed that the developer will adapt their client-side/server-side code such that they can access the token within the cookie that we have created in the first place and then be

able to pass it as a value that will eventually be sent back to the server for validation.

Below we explain the code that our rule adds and that is also seen in figure 17. For csrf middleware, a token needs to be manually created by instantiating the middleware with a secret key. The secret key is created using a core module built into Node.js called `crpyto` which generates encrypted pseudo-random data. The argument represents the number of bytes to be generated [26].

In short, the secret is used to create and verify CSRF tokens and to ensure that the requests made are legitimate. This is achieved by our rule as seen in the code below:

```

const secret = crypto.randomBytes(16).toString("hex");
const tokens = new csrf({ secret });
```

Next [29], as it can be seen in the below code, our rule will set up the session middleware. This will help with storing the CSRF token. When the server will send the CSRF token to the client, it will also store the CSRF token in the user's session. Afterwards, when a particular form will be submitted by the user, the server will have to compare the CSRF token that was initially sent through the form with the one that was stored in the user's session.

Our other rule **express-session-cookie** could be used for adding additional attributes to the cookie session. In this context, `httpOnly` will prevent the cookie from being accessed by client-side JavaScript Code which prevents to some extent the session id from being stolen. `sameSite` attribute will allow the cookie to be sent only with a request from our site. And finally, `maxAge` will give an expiration timer that would limit the time of an attacker to get a hold of the session id. Of course, these attributes could affect the user experience, and therefore consideration from the side of the developer needs to be taken.

```
app.use(session({
  secret: 'session secret',
  resave: false,
  saveUninitialized: true,
  cookie: {
    secure: true,
    maxAge: 15 * 60 * 1000,
    httpOnly: true,
    sameSite: 'strict'
  }
}));
```

Secondly, in the code below our rule implements a custom middleware function that checks whether a CSRF token exists for the current session `req.session.csrfToken`. If not, it creates one and stores it in the session. It then assigns the CSRF token from the session to `res.locals.csrfToken`. The purpose of `req.session` is to store the corresponding session data from server side when requests come in. These requests are tracked by the `express-session` middleware, which looks into the request cookies for specific session IDs. `res.locals.csrfToken` will be adding `csrfToken` property to the locals object for the current response.

```
app.use((req, res, next) => {
  if (!req.session.csrfToken) {
    req.session.csrfToken = tokens.create(secret);
  }
  res.locals.csrfToken = req.session.csrfToken;
  next();
});
```

Since our rule focuses only on the server-side implementation of the CSRF protection, it would also be expected that on the client side, the developer is able to submit a particular form with the respective `csrfToken` that will be validated on the server side.

In the example below, the CSRF token that was created in the middleware and was saved in the session is embedded in the form. When the form will be submitted, the token will be included in the POST request. The `input type` should be `hidden`, such that a particular attacker would not be able to see it.

```
app.get(’/form’, (req, res) => {
  const formHTML = ‘
    <!DOCTYPE html>
    <html>
    <head>
      <title>Form</title>
    </head>
    <body>
      <form action="/form" method="post">
        <input type="hidden" name="_csrf"
        value="\${req.session.csrfToken}">
        <input type="submit" value="Submit">
      </form>
    </body>
    </html>
  ‘;
  res.send(formHTML);
});
```

Evidently, trying to create a rule that accommodates for particular client-side form is complicated as it could be that the developer wants to have a form that functions/behaves differently.

Next, when the form is submitted, the browser sends the CSRF token back to the server. When the server receives the form information, it compares the token sent by the browser with the token it sent when the form was loaded initially. If the tokens match, the server knows that the form submission is legitimate.

```
app.post(’/form’, (req, res) => {
  const submittedToken = req.body._csrf;
  if (tokens.verify(req.session.csrfToken, submittedToken)) {
    res.send(’Form submission successful
```

```
5            and CSRF token validated!');
6   } else {
7     res.status(403).send('Invalid CSRF token');
8   }
9 });
```

4.5 Check password hashing

4.5.1 Describing the vulnerability

Storing passwords as plain text in a database could potentially be a single point of failure vulnerability if the system gets compromised by hackers. As referenced in the common weakness enumeration definition list with id *CWE-256*. This "*password management issues occur when a password is stored in plain text in an application's properties, configuration file, or memory* (...)" [7] Therefore, In the context of authentication storing hashed passwords has become an absolute standard and best practice approach when building login systems. Although many websites and open-source CMS systems (WordPress, Drupal) already adhere to this convention by default, usually by storing the password in a relational database system using an MD5 encryption algorithm [32], it has to be done manually when building applications in a node/express environment as express servers nor the commonly used object-orientated MongoDB enforce the developer to store sensitive information in an encrypted format. Despite the many strong hashing libraries like bcryptjs and argon2, if the developer is a novice and doesn't utilize them properly applications built with these popular JavaScript stacks could be a target of this type of vulnerability.

While hashing passwords can greatly improve security, it is like many other vulnerabilities, not a foolproof solution. One potential concern that can arise when using password hashes is a "rainbow table" attack, where an attacker generates a table of possible password hashes and then uses it to look up the password corresponding to a given hash quickly. To prevent this type of attack, it is important to use a unique salt for each password, which makes it much more difficult for an attacker to generate hashes.

Although both mentioned libraries do provide an option to salt the password, another challenge of performance may occur. While more secure hashing algorithms like Argon2 can provide stronger protection against attacks, they may also be more computationally intensive and slower to compute, which can impact the performance of the application. To balance the concern of security and performance, we have decided to reproduce our hashing detection rule using the bcryptjs library, as this encryption algorithm is considered to outperform Argon2 in both security and performance slightly. [39]

4.5.2 Detecting the vulnerability

The custom ESLint rule, *check-password-hashing*, is designed to mitigate the vulnerability associated with storing passwords in plain text, as described earlier. While there are various methods to save a newly registered user's data to a database, such as using relational or object-oriented databases and different types of transactions, this rule specifically focuses on Express-based applications that use MongoDB for data storage.

Since MongoDB is an object-oriented data storage, the rule is designed to look for patterns that instantiate objects - in particular, objects with properties that contain "passwords" in their keys.

```
const newUser = new User({
  name,
  email,
  // password // <--- attribute is not using hashed password x
  password,
});

newUser
  .save()
  .then((user) => {
    // success
    errors.push({ msg: "you are registered" });
    console.log(newUser);
```

Figure 18: MongoDB-object containing plain text password detected

The rule will, however, only target objects where the .save() method, later on, will be invoked. This is because, in MongoDB, the .save() method is used to persist data to the database. Therefore, the rule's design is to scrutinize these instances, ensuring that any object containing a "password" key undergoes secure handling before the .save() operation is performed, while regular objects will be ignored.

This ensures that any operation involving password handling adheres to secure practices, such as hashing, before the data is persisted in the database. This mitigates the risk of storing plain text passwords, enhancing the overall security of user data.

If the object was saved to the database, as shown in the figure, the password would appear as plain text and would thus violate the best practice of storing passwords in a hashed format, making the application a target for the vulnerability described in *3.3.1.* As seen in the figure below, whenever someone submits the registration form via the route "users/register", a post request is sent to the server that would subsequently respond with the content of the object.

```
{
  name: 'John Doe',
  email: 'demo_user@itu.dk',
  password: 'Demo9000',
  _id: new ObjectId("645ce5a30ccf7464d5a0a03c"),
  date: 2023-05-11T12:54:59.253Z,
  __v: 0
}
```

Figure 19: Server responding after registration with plain text password

4.5.3 Fixing the vulnerability

As implied, this scenario is exactly what should be avoided. Our rule *check-password-hashing* will loop through all objects of the file and look for password properties and

trigger if a match is found, returning a warning message, i.e.

```
1  function checkPasswordUsage(node, properties, objectNode) {
2    for (const property of properties) {
3      if (
4        property.key.name === "password" &&
5        property.value.type === "Identifier" &&
6        property.value.name === "password"
7      ) {
8        context.report({
9          node: property,
10         message:
11           "Do not use plain text passwords. Use hashed passwords
                 instead.",
12       });
13     }
14   }
15 }
```

Since we are going to utilize the bcryptjs library for hashing the password the rule will check if this is already required in the target file using a *ast.body.filter* import the library at the ending range of imports and add it at that location later on in the fix.

```
1      if (!bcryptImported) {
2            const importNodes = context
3              .getSourceCode()
4              .ast.body.filter(
5                (n) =>
6                  n.type === "ImportDeclaration" ||
7                  n.type === "VariableDeclaration"
8              );
9            if (
10             !importNodes.some(
11               (n) =>
12                 n.type === "VariableDeclaration" &&
13                 n.declarations.some((d) => d.id.name === "bcrypt")
14             )
15           ) {
16             const lastImportNode = importNodes[importNodes.length -
                   1];
17             const lastImportEnd = lastImportNode.range[1];
```

Before the fixer object is declared for the automatic fix, a specific condition needs to be satisfied. The rule should only target objects that are relevant for saving to MongoDB, specifically those on which the .save() method is subsequently invoked. To ensure this, the rule initiates another traversal of the code to find objects that meet this criterion.

The checkPasswordUsage function, created earlier, is then invoked on the identified objects. This step ensures that only relevant objects, i.e., those that are pertinent for

database storage, are considered. For example, consider a scenario where an object named `newUserTest`, containing the password property, is declared but no `.save()` method is invoked on it. In this case, the rule is expected to ignore `newUserTest` and instead focus on correctly identifying objects like `newUser`, whose properties are intended to be saved to the database:

```
1       return {
2
3         "CallExpression:exit"(node) {
4           if (
5             node.callee.type === "MemberExpression" &&
6             node.callee.property.name === "save"
7           ) {
8             const objectName = node.callee.object.name;
9
10            const variable = context
11              .getScope()
12              .variables.find((v) => v.name === objectName);
13            if (variable && variable.defs[0].node.init.type === "
                  NewExpression") {
14              const properties =
15                variable.defs[0].node.init.arguments[0].properties;
16              checkPasswordUsage(node, properties, variable.defs[0].node.
                    parent);
17            }
```

Finally, the fixer object is created to automate the solution to the issue at hand. Code is inserted at three distinct locations in the Abstract Syntax Tree (AST). Firstly, an import statement for the 'bcryptjs' library is added to the existing list of imports. Next, the password is hashed using the `bcrypt.hashSync` method with a salt of 10 rounds, and this operation is assigned to the `hashedPasswordDeclaration` constant. This code block is inserted just before the object in which the plain text password was found, effectively replacing it with: `hashedPassword = bcrypt.hashSync(`*property.value.name*, 10)`;`. The choice to use the 'bcryptjs' library and the `hashSync` method was made considering the balance between performance and security. Additionally, the implementation of the hash function is succinct, making it ideal for an ESLint rule that can automate the fix. The salt rounds are set to 10 by default, denoting the number of times the hashing algorithm is applied. In this context, a salt round of 10 means the data will be processed 2^{10} (i.e., 1024) times, providing a significant diffusion to the original string. The higher the number of salt rounds, the longer the process takes, adding an extra layer of security in case of database compromise or "rainbow table attacks." At the last location, the actual critical password properties that contain the plain text password will be replaced by the `const hashedPassword` we declared above the object containing the hashed version returned by the `hashSync` method. Full fix implementation is thus as follows:

```
      fixes.push(
              fixer.insertTextAfterRange(
                [lastImportEnd, lastImportEnd],
                `\n // importing bcrypt \nconst bcrypt = require("
                    bcryptjs");`
              )
            );
          }
          bcryptImported = true;
        }


        const hashedPasswordDeclaration = `// Hashing the password
            using bcrypt with a salt \nconst hashedPassword = bcrypt.
            hashSync(${property.value.name}, 10);`;

        fixes.push(
          fixer.insertTextBefore(
            objectNode,
            `${hashedPasswordDeclaration}\n`
          ),
          fixer.replaceText(property, `password: hashedPassword`)
        );

        return fixes;
```

By implementing a specific fix to the critical code pattern of storing plain text passwords in MongoDB, we can ensure the passwords are hashed correctly using the bcryptjs library. This modification takes place at three distinct code locations: the import list, above the object declaration, and within the password property.

To begin with, the bcryptjs library must be imported. Then, the hashing method is invoked, and the returned value is assigned to a constant variable named `hashedPassword`. This hashed password replaces the original plain text password within the password property of the user object.

The outcome of this fix is the transformation of the initial code into a pattern that adheres to best practices for user registration. Consequently, this mitigates the security vulnerability associated with storing passwords in plain text.

To confirm that the password was hashed correctly, we can check the server response in the console. This is demonstrated in figure 21. This modification not only secures the application but also educates on the importance of securely handling sensitive user data.

```
//  Hashing the password using bcrypt with a salt
const hashedPassword = bcrypt.hashSync(password, 10);
const newUser = new User({
  name,
  email,
  password: hashedPassword, // <--- attribute is using hashed password gen by bcrypt √
});

newUser
  .save()
  .then((user) => {
    // success
    errors.push({ msg: "you are registered" });
    console.log(newUser);
```

Figure 20: Code after applying the fix - secure handling of registration using hashing

```
{
  name: 'John Doe',
  email: 'demo_user3@itu.dk',
  password: '$2a$10$Ob2MA4IKSLWzSdVfi/xLHeYid9ZUXcl2b.f3UML1SopdLtxo.7ZU2',
  _id: new ObjectId("6462160c6259418bb10c1129"),
  date: 2023-05-15T11:22:52.667Z,
  __v: 0
}
```

Figure 21: User object logged to the console with hashed password

4.6 Enforce Password Policy

4.6.1 Describing the vulnerability

As mentioned in the previous vulnerability, password hashing is not a fail-proof solution if passwords are weak and guessable. Even if salt is added to the hashing and a ”rainbow table attack” could be mitigated like so, an adversary may still simply try to input commonly used passwords from a large collection found online to grant unauthorized access. Therefore, it’s important for the developer to encourage the user to create a strong password in the first place during the registration process. According to a recent report by Verizon, 81% of data breaches are caused by weak or compromised passwords. [37]. Different recognizable properties define weak passwords:

1. **Length and complexity:** Short passwords with a repeated or logical pattern that are guessable for humans or computers with little or no complexity added to them, like a combination of upper and lowercase characters or special symbols.

2. **Password with personal info:** These kinds of passwords are too easy to guess since they contain predictable personal information like birthdays, phone numbers, names, or a combination of them.

3. **Deprecated passwords**: Passwords that have not been updated for a while and potentially could have been a part of one of the major data breaches that have occurred in recent years. [15]

To ensure strong passwords, it's therefore important to enforce a password policy that will avoid any of the above properties. This can be a decision the developer makes by conducting data validation of the password input field upon signing up that rejects registration to proceed if these requirements are not fulfilled. One way to encourage users to create strong passwords is to provide feedback on the strength of the password they are entering. This can be done in real time as the user is typing their password or after they have submitted it. This feedback can include information about the length, complexity, and uniqueness of the password, as well as suggestions for how to improve it.

However, as this security layer is entirely voluntary and no generic code pattern to set up a password policy is mandatory within a node and express environment, we consider this a great case for a static analysis tool like ESlint to detect the absence of such validation.

4.6.2 Detecting the vulnerability

As underscored in the previous section, the importance of employing strong passwords as a primary defence against unauthorized access and potential data breaches cannot be overstated. A significant portion of data breaches can be traced back to the acquisition of compromised passwords, either by exploiting weak credentials or through brute force attacks. This highlights the role of strong passwords as a high priority in securing sensitive data.
A straightforward and commonly used strategy to encourage the creation of robust passwords involves enforcing stringent requirements during the user registration process. This might include stipulations such as minimum length, the inclusion of both uppercase and lowercase letters, numbers, and special characters. Additionally, discouraging the use of common or predictable password patterns can further enhance password security. On that note, we have created a custom rule `enforce-password-policy` that aims to detect if such validations exist within a given code pattern. This rule in specific scrutinizes the code for the usage of a registration route of some sort e.g. `"router.post(/register) (...)"`. This is because data validation usually is done within this route function, allowing the user to register if all conditions are met or halt the registration and throw an error message, displayed in the registration form, if not. These validations can be a way of checking if a user already exists if the email has the wrong format and of course the complexity of the inputted password. The absence of the latter conditions is exactly what this rule is trying to detect. It does so by checking the usage of the library `password-validator` and its powerful validation schema. This library is flexible and customizable password validation for Node.js that helps in ensuring that users create secure and strong passwords, which can help protect their accounts from being breached.

4.6.3 Fixing the vulnerability

The password-validator library's key functionality lies in its comprehensive validation schema, which allows for the creation of complex validation rules with relative ease. This schema is essentially a set of rules defined to meet the specific password policy requirements. By leveraging its chainable methods, the library allows developers to string together multiple validation rules in a clear and concise manner. Each method in the chain provides some sort of validation in a very semantic syntax using relational and conditional keywords like "is", "has" "not" etc. This enables a succinct and efficient validation of password length, usage of special characters, usage of upper and lower case, and even blacklisting of predefined passwords - that saves the developer time and effort writing numerous excessive if-statements. A chain of methods within the schema will then subsequently be

invoked by passing the content of the user-typed password field to the `schema.valdate` method. Violation of any of the conditions can thus be checked by negating this validation in an if-statement. The validation will then only pass if all conditions are met in the schema.

```
1 var passwordValidator = require('password-validator');
2
3 // Create a schema
4 var schema = new passwordValidator();
5
6
7 schema
8 .is().min(8) // Minimum length 8
9 .is().max(100) // Maximum length 100
10 .has().uppercase() // Must have uppercase letters
11 .has().lowercase() // Must have lowercase letters
12 .has().digits(2) // Must have at least 2 digits
13 .has().not().spaces() // Should not have spaces
14 .is().not().oneOf(['Passw0rd', 'Password123']); // Blacklist these
     values
15
16 (...)
17
18   if (!schema.validate(password)) {
19     errors.push({ msg: "Password must be at least 8 characters long
         and include uppercase and lowercase letters, as well as at
         least one special character" });
20   }
```

To utilize this validation boilerplate in our rule we initially detect if the library is imported:

```
1
2      VariableDeclarator(node) {
3         if (
4           node.init &&
5           node.init.callee &&
6           node.init.callee.name === "require" &&
7           node.init.arguments[0].value === "password-validator"
8         ) {
9           hasPasswordValidatorImport = true;
10        }
11      },
```

Furthermore, our approach involves pinpointing the exact location of a specific code pattern, namely `router.post`, that includes the "register" path. This method is essential as it allows us to identify the section of the code where user registration functionality is implemented.

```
    CallExpression(node) {
      if (
        node.callee.type === "MemberExpression" &&
        node.callee.object.name === "router" &&
        node.callee.property.name === "post" &&
        node.arguments.length > 0 &&
        node.arguments[0].value === "/register"
      ) {
        registerRouteNode = node;
```

If we detect the specified code pattern—namely, the presence of a registration route without the use of the password-validator library—then an automated fix can be initiated. This fix involves generating a schema that utilizes a sequence of data validation methods, as illustrated in the previous section. Importantly, this schema is strategically inserted at the beginning of the file. This approach ensures that robust password validation measures are in place prior to the registration process, thereby making the `schema.validate()` method readily available for invocation within the route function. The fixer object is therefore written as follows:

```
          fix(fixer) {
            const importText =
              "const passwordValidator = require('password-validator
                  ');\n\n";
            const schemaText =
              "const schema = new passwordValidator();\n\nschema\n .
                  is().min(8)\n .has().uppercase()\n .has().lowercase
                  ()\n .has().symbols()\n .has().digits()\n .not().
                  spaces();\n\n";

            const insertImportFix = fixer.insertTextBefore(
              context.getSourceCode().ast.body[0],
              importText + schemaText
            );

            // Insert password validation code after the password
                length check
            const insertValidationFix = fixer.insertTextAfter(
              passwordLengthCheckNode,
              `
    if (!schema.validate(password)) {
      errors.push({ msg: "Password must be at least 8 characters long
          and include uppercase and lowercase letters, as well as at
          least one special character" });
    }
              `
            );

            return [insertImportFix, insertValidationFix];
```

From the perspective of the user, during the registration process, if the password they input contravenes the validations defined in the schema provided by the fixable rule, an error message will promptly be displayed above the registration form. This instant feedback not only helps the user understand the specific password requirements that were not met but also encourages them to adhere to the established password policy. By clearly communicating these requirements, users can create secure passwords on their first attempt, thereby enhancing the overall user experience while mitigating the risk of their password being guessed or compromised using brute force attacks.

Figure 22: Warning message in register form due to violation of password policy

4.7 Prevent brute force

4.7.1 Brute force attacks

Although a proper password policy may reduce the likelihood of passwords being guessed by brute-force attacks, there are other ways that attackers can attempt to gain access to an application. For example, attackers can use malicious software that is capable of simulating millions of login requests per second, thus making it possible to guess even fairly sophisticated passwords. To defend against these types of attacks, it's important to limit the number of allowed login attempts for a given user or IP address.

This can be done by implementing a lockout policy that temporarily blocks a user's

account or IP address after a certain number of failed login attempts. The duration of the lockout period and the number of allowed attempts should be carefully considered to balance security with user convenience.

In addition to limiting login attempts, it's also important to monitor login activity for unusual behaviour that may indicate a brute-force attack is in progress. This can include monitoring for repeated failed login attempts, login attempts from unusual IP addresses or locations, or patterns of login attempts that indicate automated attacks. To reproduce this vulnerability several open source express libraries were developed. A widely used one for this purpose is *express-brute* - this library is specifically designed to protect against brute-force attacks by implementing rate-limiting and temporarily locking accounts after a configurable number of failed login attempts. In the next section, we will demonstrate a rule detecting the absence of brute force protection for the login route and, as an automatic fix, conducts a simple implementation of the mentioned library to avoid this kind of vulnerability.

In the previous rule, we demonstrated how a custom rule could aid the developer and user in adhering to a strong password policy. This could reduce the likelihood of the password being guessed and may also reduce the possibility of brute force attacks being successful. Brute force attacks are as already described a type of cyberattack in which an attacker attempts to gain access to a system by systematically trying all possible combinations of passwords or encryption keys until the correct one is found. This method essentially relies on the "brute force" of computational power to overcome security barriers, hence the name.

The time it takes for a brute force attack to succeed depends on the complexity and length of the password, which is why strong password policies are crucial e.g. Eight-character password composed only of lower-case letters offers 208 billion combinations, while an eight-character password that follows the policy of our previous rule will offer over six quadrillion combinations. [38]

While it's true that increasing the number of possible password combinations can make a brute force attack exponentially more time-consuming and thus less likely to succeed, we must also consider the rapidly evolving landscape of computing technology. Particularly with advancements in Artificial Intelligence (AI) and Machine Learning (ML), computers are predicted to become significantly more powerful and efficient in processing vast amounts of data. This improvement in computational power also implies an enhanced capability to predict and compute potential password combinations.

In this context, the threat of brute force attacks remains relevant and possibly even increases, as technological advancements could enable these attacks to become more sophisticated and efficient. [36]

Therefore, it is a relevant mitigation strategy to implement a brute force prevention library that will limit the amount of failed login attempts and disallow users to grant access to the application for a specific time frame.

4.7.2 Detecting vulnerability

In the rule *prevent-brute-force*, we have taken additional steps to enhance our code pattern detection. In addition to detecting the absence of the express-brute-library and the usage of a route path containing the keyword 'login,' we also want to detect the presence of a potential login system in the code.

To accomplish this, we will invoke a regex expression on the post routes to localize the position of the route dealing with the login request. By analyzing the code pattern using regex, we can identify specific routes that handle the login functionality. This allows us

to further refine our rule and assume that the application may be relevant to brute force protection if these conditions are met.

```
1     CallExpression(node) {
2       if (
3         node.callee.type === "MemberExpression" &&
4         node.callee.object.name === "router" &&
5         node.callee.property.name === "post" &&
6         node.arguments.length > 0 &&
7         /\/login/i.test(node.arguments[0].value)
8         // detects all routes containing the keyword "login" using
            regex.
9       ) {
10        // Login route detected
11        loginRouteNode = node;
```

Before resuming with the automatic fix, we look for brute force prevention that potentially has already been implemented. This is achieved by traversing the code for usage of *brute-force* library and the method `bruteforce.prevent()`. If the rule thus succeeds in finding a login route but no implementation of a brute force prevention, i.e. the application allows unlimited failed login attempts, the desired code pattern is found, a fix can subsequently be applied.

4.7.3 Fixing the vulnerability

To brute force protect the application the brute-force middleware should be imported and configured. Our automatic fix is doing exactly that. At the beginning of the file *express-brute* is required and the different flags are set to some initial values.

freeRetries: This flag determines the number of allowed requests before the rate-limiting comes into effect. Once the specified number of free retries is exhausted, subsequent requests will be subject to rate limiting. The default value is 2.

maxWait: It represents the maximum amount of time (in milliseconds) a user must wait before making another request after being rate-limited. This flag helps in preventing excessive retries by enforcing a delay between subsequent requests. The default value is 500 milliseconds.

store: It specifies the storage adapter to be used for persisting rate-limiting data. Express-Brute supports multiple storage adapters such as MemoryStore,

```
1  const bruteforce = new ExpressBrute(store, {
2  freeRetries: 5,
3  minWait: 600 * 1000, // <-- set to 60 minutes (600000 milliseconds)
       per default
4  failCallback: function (req, res, next, nextValidRequestDate) {
5    console.log('Too many failed login attempts. Rate limiting request
         .');
6    req.flash(
7      'error',
8      'Too many failed login attempts. Please try again later.'
9    );
```

```
10      res.redirect('/users/login');
11    },
12  });
```

Our fixer object will, as demonstrated in the code above, set the `freeTries` flag to 5, which means that users will be allowed 5 failed login re-attempts before the rate-limiting kicks in. This value provides a certain level of flexibility for users to make mistakes while entering their credentials without being immediately restricted.

Additionally, the `maxWait` time is set to 600*1000, equivalent to 1 hour (6M milliseconds). This means that if a user exceeds the allowed number of failed attempts, they will be forced to wait for a maximum of 1 hour before being able to retry. This delay helps mitigate brute-force attacks by introducing a time-based restriction on consecutive login attempts.

It's important to note that these values are just examples and should be adjusted according to the specific requirements and security considerations of the application. In a live application, the configuration may differ depending on factors such as the sensitivity of the data being protected, the user base, and the desired level of protection against malicious activities. By configuring the `freeTries` and `maxWait` flags appropriately, developers can strike a balance between usability and security, allowing a reasonable number of retries while also preventing abuse and unauthorized access. To enable this middleware to handle login requests, the callback method `bruteforce.prevent()` callback method needs to be incorporated into the POST route (as seen in figure 19) responsible for handling the login functionality, i.e. the node that we detected and localized earlier. Once the fix is applied, potential brute-force attacks will be unsuccessful as the stream of illegitimate login requests will be halted. Users will be alerted about the failed attempts above the login form and asked to try again later (figure 21).

```
// Login
router.post("/login", bruteforce.prevent, (req, res, next) => {
  passport.authenticate("local", {
    successRedirect: "/dashboard",
    failureRedirect: "/users/login",
    failureFlash: true,
  })(req, res, next);
});
```

Figure 23: Callback function bruteforce.prevent() passed to route

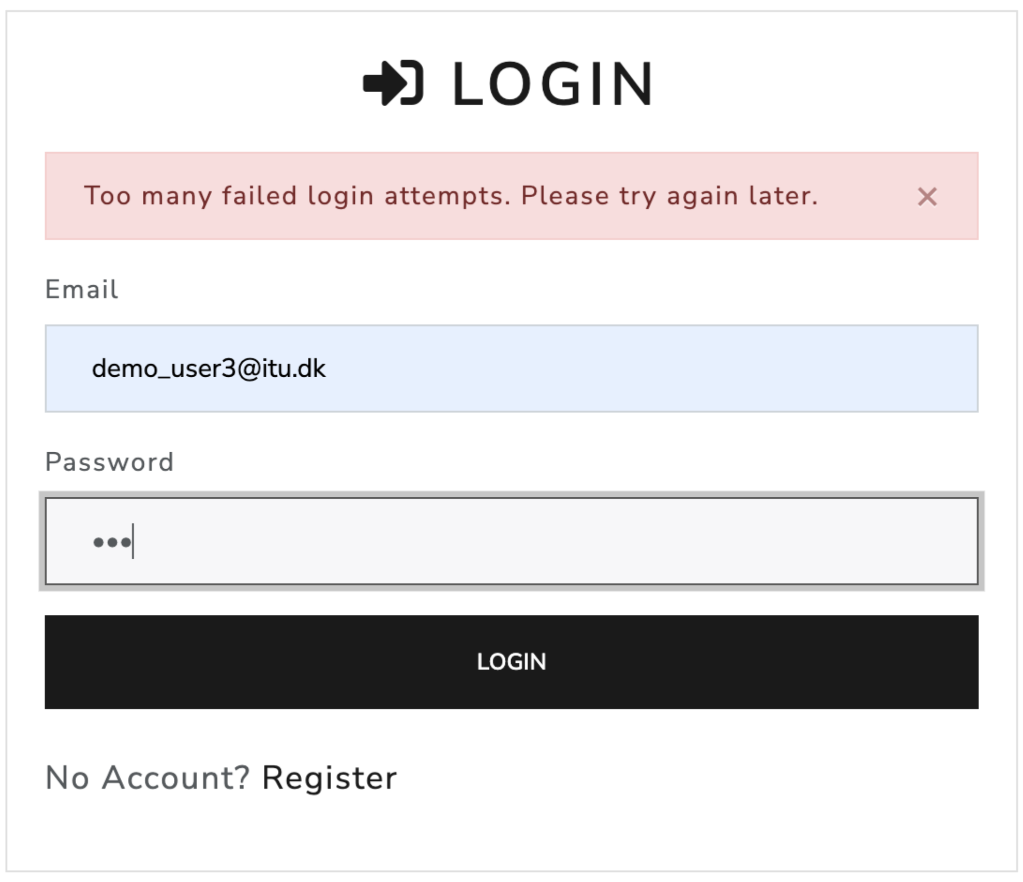


Figure 24: Potential Brute force attempt detected

4.8 Insecure express session

4.8.1 Describing the vulnerability

Sessions in Express are a way to persist data across requests. Each user that visits a website has a unique session, which means that the variable data isn't shared between users. Sessions are commonly used to store information about the user, allowing you to personalize the website to them. It is also a common way to keep track of a user's authentication state and, therefore, mandatory when working with authentication and access control.

Sessions are not configured per default in express, however, and it's necessary to require the library **express-session** and use it as middleware and set some properties and flags. This will enable sessions in the application, but the generic implementation of cookies in express sessions is not configured to be secure automatically. It is thus up to the developer to properly configure cookies to be secure. If not done properly, the application could be vulnerable to interception during transmission by an adversary. Sessions provide different flags that, if set correctly, potentially could avoid common attacks respectively:

Man-in-the-middle: Setting the `Secure` flag on a cookie is critical to prevent man-in-the-middle attacks. Without it, cookies can be transmitted over insecure HTTP connections. This can lead to potential interception and misuse of sensitive data, like session identifiers, especially over public WiFi networks.

Cross-Site Scripting (XSS): Cookies without the `HttpOnly flag` can be accessed via client-side JavaScript. This exposes them to potential XSS attacks, where malicious scripts can steal these cookies, leading to user impersonation.

Cross-Site Request Forgery (CSRF): Absence of the `SameSite flag` in cookies allows them to be included in cross-site requests. This enables CSRF attacks, where users can be tricked into executing unintended actions on the attacker's behalf.

There are obviously numerous potential interception-based attacks that could be avoided simply by setting these flags correctly. After that realization, we developed an EsLint rule that would detect code patterns with the presence and configuration of express sessions that do not correctly set these flags.

4.8.2 Detecting the vulnerability

This ESLint rule *express-insecure-sessions* checks for the properties as described above: `Secure`, `httpOnly` and `sameSite`.

The rule first determines if a piece of code is calling a function named '`session`' by looking at the structure of the code. This is done in the `isSessionCall` function. If the function being called is not 'session', it does not process further.

If it is a 'session' function, the rule then checks whether the object passed as an argument to the function call has the properties `secure`, `httpOnly`, and `sameSite` in its '`cookie`' property. This is done in the `isMissingCookieProperties` function.

If the '`cookie`' property is not an object or if it doesn't include all three properties (`secure`, `httpOnly`, `sameSite`), the rule will consider the session insecure and raise a warning or error.

It's important to note that this rule will only apply to code that's following the convention of naming the function that set up session cookies '`session`' and it's assuming that the configuration for the session cookie is passed in an object to this function. This is common in Express.js applications, for example, but may not apply to every Node.js application. For that reason, this rule is named with the prefix `express` to underscore this

point. Our more general-purpose insecure cookie detection rule `Cookie Safe Attribute` aims, as described in section **6.1.1** to look for this vulnerability regardless of the frameworks being used.

The principles of enforcing `secure`, `httpOnly`, and `sameSite` attributes on session cookies, however, remain relevant across different architectures and frameworks.

4.8.3 Fixing the vulnerability

The ESLint rule 'express-insecure-sessions' not only identifies problems but also proposes solutions. It uses the 'fixer' object, provided by ESLint, to modify the code automatically. Here's how it works:

If the '`cookie`' object is found to be missing any of the required properties ('`secure, httpOnly, sameSite`'), the rule suggests a fix inside the '`context.report`' method:

```
1     fix(fixer) {
2   const missingProps = [
3     'secure: true',
4     'httpOnly: true',
5     'sameSite: "lax"',
6   ];
7
8   // code for fixing
9 }
```

The '`fix`' method receives a '`fixer`' object which provides several methods for modifying code. In this case, '`fixer.insertTextAfter`' is used, which inserts new text after a specified AST node.

The missing properties are first compiled into a string '`missingProps`', which includes each missing property and its recommended value.

Then there are two different strategies for fixing the code:

1. If a 'cookie' object already exists but is missing properties, the rule adds the missing properties directly into the existing 'cookie' object:

   ```
   1     const lastProperty = cookieProp.value.properties[cookieProp.
             value.properties.length - 1];
   2 return fixer.insertTextAfter(lastProperty, ', ' + insertText);
   ```

   Here, '`insertText`' is the string of missing properties. '`fixer.insertTextAfter`' adds this string after the last property of the 'cookie' object.

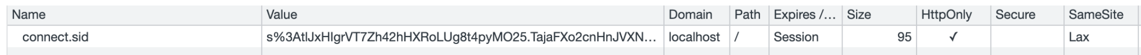
2. If the 'cookie' object does not exist, the rule adds a new 'cookie' object with the missing properties:

```
1    const insertText = ', \n cookie: { ' + missingProps.join(', ') + '
         }';
2 const lastProperty = optionsNode.properties[optionsNode.properties.
     length - 1];
3 return fixer.insertTextAfter(lastProperty, insertText);
4 ```
```

Here, ‘`insertText`‘ is a string that includes the new ’cookie’ object and its properties. ‘`fixer.insertTextAfter`‘ adds this string after the last property of the options object that’s passed to the ’session’ function.

This way, the ‘express-insecure-sessions‘ rule not only detects insecure configurations but also attempts to fix them automatically, making it easier for developers to adhere to best security practices.

Inspecting the web page using the developer tool will allow us to view the session cookies after the fixes are applied. By doing so, we can confirm that these properties are indeed set correctly (figure 22). Setting the secure flag, however, will only work for web applications with a valid SSL certificate. In a development environment using where the site is previewed from localhost without HTTPS protocol, the redirection after login will fail as this will, in this case, create a post request from an insecure protocol. Therefore, it’s recommended to disable this flag while testing and re-enabling it for live applications.



Name	Value	Domain	Path	Expires /...	Size	HttpOnly	Secure	SameSite
connect.sid	s%3AtlJxHlgrVT7Zh42hHXRoLUg8t4pyMO25.TajaFXo2cnHnJVXN...	localhost	/	Session	95	✓		Lax

Figure 25: Inspecting the cookie attributes in developer tools

4.9 Express rate limit

4.9.1 Describing the vulnerability

Denial of service (DoS) attacks is a very common cyberattack that aims to disrupt the normal functioning of a system or website by flooding the victim’s server with an overwhelming amount of illegitimate traffic. The objective is to break the network infrastructure to such an extent that the actual request from legitimate users is halted, resulting in the user experiencing downtime of the system, preventing them from doing their job, i.e. logging into the intranet, communicating, or accessing information. Victims could also be online retailers that could lose revenue from customers unable to complete purchases or even governmental institutions where the attacks perhaps have a more political motif. In some cases, cybercriminals may use DoS attacks as a smokescreen or distraction while carrying out other malicious activities, such as stealing sensitive data or injecting malware. Since the website’s resources are focused on mitigating the DoS attack, these secondary attacks may go unnoticed for an extended period. Regardless of the motivations of the attacks, DoS comes in many variations and can from a more technical perspective, be broadly categorized into four types based on their methods and targets. Some of the most common types of DoS attacks include:

1. Volume-based attacks: These attacks aim to consume the bandwidth of the target network by generating a large volume of traffic. Examples include:

   UDP flood: In this attack, the attacker sends a large number of User Datagram Protocol (UDP) packets to the target, overwhelming its capacity to process them. ICMP flood (Ping flood): The attacker sends a large number of Internet Control Message Protocol (ICMP) echo request packets (pings) to the target, causing it to become unresponsive.

2. Protocol-based attacks: These attacks exploit vulnerabilities in the target’s network protocols to consume its resources. Examples include:

   SYN flood: The attacker sends a series of TCP SYN (synchronization) packets to initiate multiple incomplete connections, exhausting the target’s resources for

managing new connections. Smurf attack: The attacker sends a large number of ICMP echo request packets with a spoofed source IP address (the target's IP) to a broadcast network, causing all devices on that network to send replies to the target and overwhelm it with traffic.

3. Application-layer attacks: These attacks target the application layer (Layer 7) of the OSI model, aiming to exhaust the resources of the target application or service. Examples include:

   HTTP flood: The attacker sends a large number of seemingly legitimate HTTP requests to the target web server, causing it to become overwhelmed and unresponsive. Slowloris: The attacker opens multiple connections to the target server and sends partial HTTP requests, keeping the connections open for as long as possible. This prevents the server from serving other legitimate requests.

   [34]

4. Distributed Denial of Service (DDoS) attacks: These attacks involve multiple systems (often compromised and controlled by a botnet) that work together to generate a massive amount of traffic or requests, overwhelming the target. DDoS attacks can employ any of the previously mentioned attack types but are executed from multiple sources, making them harder to detect and mitigate.

Our rule *prevent-brute-force*, as demonstrated in the next section of the paper, will be focusing on HTTP flood type (3) as our ESLint plugin will be designed with the objective of detecting and preventing authentication-based vulnerabilities in node/express-driven applications. Our rule uses the **express-rate-limit** middleware as a dependency. This library provides an easy way to set up DoS attack prevention tracking the number of requests coming from each client's IP address and enforcing a specified limit over a defined period.

If a client exceeds the allowed number of requests within that period, the library responds with an HTTP 429 'Too Many Requests' status, and the request is rejected.

4.9.2 Detecting the vulnerability

The ESLint rule *express-rate-limit* checks an Express.js application for whether it has appropriately implemented the express-rate-limit middleware to limit the number of requests made to the API, thus mitigating the risk of Denial of Service (DoS) attacks. This would usually be detected in the file where server configurations are made. In our login application, we named this file *app.js*. The rule will determine if the application is protected against this vulnerability as follows:

1. Import Declaration Check: The rule begins by checking if the 'express-rate-limit' library is imported. If it finds a declaration with this import, it sets `expressRateLimitImported` to true. This is done in the `ImportDeclaration(node)` function.

2. Application Middleware Check: Then, the rule checks each app.use invocation. It is looking for instances where middleware is added to routes (indicated by the '`app.use`' method where the last argument is a call to require a file in the './routes/' directory). This check is done in the `CallExpression(node)` function. If it finds such an instance, it checks whether 'limiter' (the assumed variable name for the express-rate-limit instance) is an argument in the function call. If 'limiter' is not found, it adds the node to the `appUseNodes` array for later reporting.

3. Final Check and Reporting: On `Program:exit` (when ESLint is about to finish linting this file), it checks if the express-rate-limit library was imported (`expressRateLimitImported` is true) and if there were any nodes where the 'limiter' was missing. If the express-rate-limit library wasn't imported and there were `app.use` nodes that should have had the limiter middleware, it reports an error.

It is worth mentioning that, similar to other rules we have made, detecting vulnerability requires, in many cases, that the developers use standard naming conventions for node/express-driven applications. In the rate limit rule code pattern, using the variable "`app`" as the primary express declaration will benefit from this rule, while others deviating from that convention won't. This will be subject to further discussion in the next section regarding the assessment of false positives/negatives.

4.9.3 Fixing the vulnerability

The report includes a message about the problem and a fix function that would add the express-rate-limit library and its instance 'limiter' into the file and insert 'limiter' as a middleware into all app.use calls that need it. The configuration will be made at the beginning of the file and set the properties `WindowsMS`: The duration of rate limit in milliseconds, i.e. the amount of time where additional requests from the same IP address are rejected. Max: The maximally allowed request per IP address.

```
1                 const rateLimit = require('express-rate-limit');
2                 (...)
3                  const limiter = rateLimit({
4                 windowMs: 15 * 60 * 1000, // 15 minutes,
5                  max: 100, // limit each IP to 100 requests per
                      windowMs,
6                  message: "Too many requests from this IP, please try
                      again later"
7                  });
```

To enable this middleware, the limiter constant, as declared above, will be applied to all requests of the application by using `app.use(limiter);` Our fix is very generic in that sense - if the developer preferred only to activate this rate limit for certain routes, this behaviour could be achieved by specifying the custom route as the first argument like so: *app.use('/api/custom-route', limiter);* To verify that the limiter middleware was correctly implemented, HTTP requests to the application can be executed from the terminal intentionally exceeding the allowed request set in the "max" property. Using a for-loop and curl to request the page will return the HTML content in the terminal 100 times but (as seen in Figure 25) responds with an error the 101st time:

```
1 % for i in {1..101}; do
2    curl http://localhost:9000
3 done
```

```
      src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.6/umd/popper.mi
n.js"
      integrity="sha384-wHAiFfRlMFy6i5SRaxvfOCifBUQy1xHdJ/yoi7FRNXMRBu5WHdZYu1hA
6ZOblgut"
      crossorigin="anonymous"
    ></script>
    <script
      src="https://stackpath.bootstrapcdn.com/bootstrap/4.2.1/js/bootstrap.min.j
s"
      integrity="sha384-B0UglyR+jN6CkvvICOB2joaf5I4l3gm9GU6Hc1og6Ls7i6U/mkkaduKa
BhlAXv9k"
      crossorigin="anonymous"
    ></script>
  </body>
</html>Too many requests from this IP, please try again later%
fabianharlang@Fabians-MacBook-Air ~ %
```

Figure 26: Server response in the terminal for rate limit exceeded

5 Performance Evaluation

5.1 Evolution of Testing Tools

At the beginning of our project, we attempted to benchmark our ESLint rules together with other tools. We attempted to do this by using the **OSSF-CVE-Benchmark** tool (`https://github.com/ossf-cve-benchmark/ossf-cve-benchmark`) provided by Open Source Security Foundation (OpenSSF). This tool was meant to help us compare our own custom rules to other existing tools in terms of how well they perform in identifying vulnerabilities in different applications. These applications are known to contain vulnerabilities, which are identified as (CVE's) - Common Vulnerabilities and Exposures.

The reason why we did not pursue making use of the OpenSSF tool was because we were not able to integrate our own rules within the environment - this and other preceding issues of technical nature put a lot of time-constraint on our project.

While OpenSSF is a robust tool that could definitely be used for benchmarking our rules for future work, we have eventually decided to make use of ESLint's in-built testing environment together with the Mocha framework. This particular methodology also allowed us to create confusion matrices for our rules, from which we were able to get preliminary information on the performance of our rules.

5.1.1 ESlint RuleTester utility

ESlint provides a built-in rule tester that helps write test cases for customized ESlint rules. *RuleTester* utility compares the expected output of the rule custom rule that it requires in the test with a given input (that may or may not trigger the rule). Test cases can either be *valid*, i.e. the rule should not detect the pattern as an error or *invalid*, the rule should detect the pattern as an error. If there is a match between the test cases and the actual output, the rule should pass the test, and on the contrary, if the output is not matching the expected result, the test will fail. Using the RuleTester tool will improve the developer's understanding of how the rules perform and their behaviour, ensuring that it works as intended. By simply counting the number of false positives/negatives and true positives/negatives, we have an adequate sample space of data to calculate the performance of the rules using different insightful metrics like *recall*, *precision*, and *F1-*

score and accuracy that in combination make up the entities of a *confusion matrix*: a contingency table that visualizes performances of algorithms. The approach is inherited from *statistical classification* and is often used in the field of Machine Learning [10]. It seems, however, suitable for this performance evaluation as well since rules by nature are binary and can be proven either false or negative, i.e. in the context of coding patterns, they are either violated or not.

5.1.2 Combining ESLint's In-Built Testing Functionality with Mocha for Evaluating Linting Rules

For testing our rules and for computing the outcomes in the confusion matrices, we have used the in-built testing functionality that ESLint provides, however, the `RuleTester` function doesn't really provide any information on the tested rules in the form of reporting, therefore it is a bit difficult to visualise the tests that pass or fail, therefore we decided to use the Mocha framework for testing. Mocha helped the evaluation process of our rules by providing a solution for generating understandable reports of the results that we further used in building our confusion matrices. In the figure below 27, we see an example of the report generated by Mocha. The passed tests in the `invalid` array represent the true positives, while the failed tests represent false negatives. The failed tests within the `valid` array represent the false positives, and the passed tests represent true negatives which we did not take into account for the performance evaluation.

```
cookie-security
  valid
    ✓ res.cookie('name', 'value', { maxAge: 15 * 60 * 1000, httpOnly: true, sameSite: "Strict" });
    1) res.cookie('name', 'value', {  maxAge: 15 * 60 * 1000, httponly: true, sameSite: "Strict" });
    2)  res.cookie('name', {  httpOnly: true, maxAge: 15 * 60 * 1000, sameSite: "Strict", })
  invalid
    3) app.use(session({
          secret: 'session secret',
          resave: false,
          saveUninitialized: true,
          cookie: {  httpOnly: true, sameSite: "Strict" } //
        }));
    4) res.writeHead(200, {
        'Set-Cookie': 'myCookie=myValue; Max-Age=900; HttpOnly; SameSite=Strict',
        'Content-Type': 'text/plain'
      });
    ✓ res.cookie('name', 'value');
    ✓ res.cookie('name', 'value', {  httpOnly: true, sameSite: "Strict" });
    ✓ res.cookie('name', 'value', {  httpOnly: true, });
    ✓ res.cookie('name', 'value', {  });


  6 passing (79ms)
  4 failing
```

Figure 27: Mocha Report on Secure-Cookie-Rule Test

5.2 Evaluating Linting Rules Using Confusion Matrix

In order to see how accurate our ESLint rules are, we have gathered data from the tests we ran on our rules, and we have used it in confusion matrices for each of them [18, pp. 86-87]. Through testing the rules, it will be predicted if certain code is `valid` or `invalid` - meaning that if a test passes as valid, then it is assumed that our particular rule for that particular code case can either reflect a true negative or a false positive. If the test passes as invalid, then it can either reflect a true positive or a false negative. In the context of ESLint's `RuleTester`, the `valid` and `invalid` arrays are used to write tests for ESLint rules to ensure they are working as expected. For the `valid` arrays - containing our code snippets for testing - the ESLint rule should flag any errors. If this happens, the rule picking on the snippet will be considered a false negative. The `invalid` arrays contain code snippets that would be considered invalid according to our rule - more specifically, our rule will flag vulnerabilities in these snippets. As a false negative, our rule should have found a vulnerability, and it did not, while for the true positive, our rule correctly picked on the vulnerability and also fixed it. Within the `invalid` array, there also exists the `output` array, which can include the corrected code snippet version. Basically, we have placed in the `output` arrays the expected code after our rule fixed it, and subsequently, our tests passed as true positives if they were aligned.

Based on the information gathered through the Mocha framework, we constructed confusion matrices for each of our rules. These confusion matrices take the following three parameters listed below. The reason why we excluded true negative as a parameter is that we are primarily interested in how well our rules are able to detect given vulnerabilities rather than how well it performs in terms of finding non-vulnerabilities.

- True Positives (TP) which represents the number of invalid test cases in which code was correctly identified as invalid.
- False Positives (FP) which represents the number of valid test cases in which code was incorrectly identified as invalid.
- False Negatives (FN) which represents the number of invalid test cases in which code was incorrectly identified as valid.

Using the confusion matrices gave us preliminary quantitative information with regard to how well our rules perform. We then calculate the following performance metrics from the information gathered within the confusion matrix:

- The **recall** metric (also known as 'sensitivty' or 'soundnesss') represents the fraction of real flaws reported by a tool. Recall is defined as the number of real flaws reported (True Positives), divided by the total number of real flaws - reported or unreported - that exist in the code (True Positives plus False Negatives) [23, p. 8]

$$\frac{TP}{TP + FN}$$

- "The **precision** (also known as "positive predictive value") is the ratio of weaknesses reported by a tool to the set of actual weaknesses in the code analyzed. It is defined as the number of weaknesses correctly reported (True Positives) divided by the total number of weaknesses actually reported (True Positives plus False Positives)." [23, p. 8]

$$\frac{TP}{TP + FP}$$

- "The **F-Score** provides weighted guidance in identifying a good static analysis tool by capturing how many of the weaknesses are found (True Positives) and how much noise (False Positives) is produced. An F-Score is computed using the following formula:" [23, p. 9]

$$\frac{2 * Precision * Recall}{Precision + Recall}$$

We must outline, however, that this particular methodology would generally assume that the tests that our rules would encounter, would be numerous enough but also be representative of a big range of code variation that can accurately reflect how our rules truly perform. We argue that finding publicly available code bases that also contain vulnerabilities within diverse coding patterns is quite challenging. Added to this, given that some of our rules deal with very specific cases, fulfilling "soundness" at the cost of "completeness". For example, missing a syntax (`require('csrf')`) in the code would be the only factor that fires up our rule, identifying a potential lack of csrf protection. This situation automatically implies that there could be a lot of false positives since csrf protection could already be implemented. Data subtracted from a confusion matrix that revolves around such a specific rule might not fully reflect its performance.

Indeed, since some of our rules rely on controlled conditions when it comes to testing, we argue that by having our own test cases, we actually have control over the tests, making them more precise and relevant.

5.3 Understanding the performance score

In the below table, we made a complete and extended performance score inspired by the *confusion matrix* with the test results of our rules using the metrics we explained earlier to assess the performances of our rules. The tests are conducted using the *RuleTester*, and test cases were written ourselves. As the rules get updated to newer versions, the testing tool can serve as an iterative evaluation of how the rules perform in practice and determine exactly which property that may be subject to improvement in the future, e.g. accuracy, sensitivity, and the balance between, i.e. F1-score. The complete collection of test cases for each rule kind be found here:
`https://github.com/faha92/eslint-plugin-secure-authentication/tree/main/tests/lib/rules`

No.	Rule	Invalid	Valid	TP	FP	FN	Precision	Recall	F1	Fix
1	*Check-password-hashing*	5	2	2	2	3	50%	40%	44%	✓
2	*Detect-plaintext-password*	8	0	6	0	2	100%	75%	85,7%	✗
3	*Enforce-password-policy*	5	3	3	3	2	50%	60%	54.5%	✓
4	*Prevent-brute-force*	6	3	4	3	2	57.14%	66.67%	61.54% -	✓
5	*Express-rate-limit*	8	2	3	2	5	60%	37.5%	45.45% -	✓
6	*Express-session-cookie*	7	4	3	4	4	42.86%	42.86%	42.86%	✓
7	*Implicit-flow-rule*	5	1	3	1	2	75%	60%	64,2%	✓
8	*csrf-rule*	5	2	4	2	1	66%	80%	72,3%	✓
9	*Secure-cookie-rule*	6	2	4	2	2	66%	66%	66%	✓

Table 1: ESLint Custom Security Rules Performance

1. **Check-password-hashing:** The rule successfully determined only an object that later on would be saved to the database. It does, however, fail a bunch of invalid cases where the password property is assigned a hard-coded plain string. This is a violation indeed, but also an unusual pattern as developers would rarely make the mistake of giving all users the same fixed password regardless of the one they've picked using registration. The rule may, however, be improved also to catch this behaviour in future updates. Some precision of the rule failed, as it seems in some cases it doesn't correctly identify the constant holding the actual hashed password did indeed replace the plain text counterpart.

2. **Detect-plaintext-password:** *The detection of plain text passwords had a high perfect precision of 100% rate and a high recall rate of 75% all in all.* We observed that the few cases of false negatives mostly was due to unexpected naming of the password property and if the app follows a different database scheme in which credentials are not saved to objects but are included in SQL queries instead. A pattern like that could be relevant for further potential rule creation. It's worth noting, however, that despite better performance, the rule does not provide an automatic fix as the previous rule.

3. **Enforce-password-policy:** The main reason for *false negative* for detecting if a

password policy was configured is the confusion around localizing the login route if it uses another naming convention.

4. **Prevent-brute-force:** As expected, the usage of *regular expression* improved the recall metrics and made the rule detect more cases as it looks for all routes that contain the "login" keyword, thus allowing partial matches to login to be considered as critical routes that should have brute force protection as middleware.

5. **Express-rate-limit:** This rule performed at 60% precision and 37.5% recall. The recall drop was due to some false negatives where the rule did not flag instances where the express server instantiating is not begin declared using the "app" variable, which essentially is standard. The rule may be improved to utilize a more reliable search term while traversing the AST.

6. **Express-session-cookie:** Express-session-cookie: This rule flagged both valid and invalid instances at the same rate, leading to a 42.86% precision and recall rate. The rule was unable to differentiate valid cases from invalid cases effectively due to the complexities in the session cookie setup. The way we used the *express-session* in our custom-built application proves the rule very helpful in aiding us in setting up safe cookies.

7. **Implicit-flow-rule:** In terms of precision 75%, our rule scores quite high, and it can be therefore inferred that for the majority of the instances, our rule is able to identify the vulnerability and avoid false alarms correctly. On the other hand, a recall of 60% implies that there is significant room for our rule to develop as it missed 40% of the problematic cases. Finally, the F1 score (64,2%) implies that our rule performs fairly well, however, it is brought down by the recall value, which could indeed be improved. All in all, soundness is lacking a bit for this rule, but it compensates in terms of completeness.

8. **Csrf-rule:** Our rule has a precision score of 66%, which is an agreeable score, but there is still room for improvement. This could mean that there are a few situations in which our rule is picking up on false alarms as being real vulnerabilities. Nevertheless, in terms of recall, the rule score is quite high, with 80%, meaning that it is quite strong in being able to pick up correctly on potential vulnerabilities - this means a high degree of soundness. Finally, the F1 score (72,3%) shows a good result but still with potential to be better by increasing the precision.

9. **Secure-cookie-rule:** Both recall and precision for this rule are 66%. This shows that in most cases, the rule can identify vulnerabilities correctly but also not fall for false alarms. The F1 score is 66% and reveals that while the rule performs well, there is still room for improvement. The rule performs similarly when it comes to completeness and soundness.

6 Discussion

6.1 CSRF-Rule journey

Initially, we have created an ESlint rule that makes use of the "csurf" middleware. The middleware offered CSRF protection in express applications and is available as an npm package. In principle, it helped with protecting applications from Cross-Site Request Forgery (CSRF) attacks by generating and validating CSRF tokens. The advantage of using and implementing this middleware, from a rule perspective, was that the code that would have to be written would be quite standard and predictable as the developer wouldn't have had to manually create and manage the CSRF tokens - at least on the server-side. The csurf middleware handled token generation, validation and error handling automatically, offering perspectives on creating a very simple ESlint rule that would simply add the necessary lines of code for offering CSRF protection - as we will show below. Unfortunately, we found out that on the npm website, the package was being reported as deprecated due to the high amount of security issues reported. Even then, the package has almost half a million weekly downloads and seems to be one of the most popular packages for express in terms of csrf token protection. Since we discovered that the package was deprecated, we have made a separate rule that uses another middleware called "csrf" with even more weekly downloads, specifically around seven hundred thousand weekly downloads. This package has not been reported as deprecated; however, creating a rule in ESlint that is able to fully implement the csrf middleware and its functionalities is quite impossible as the developer could use different naming conventions and coding patterns.

For token generation, the csurf middleware needed to be instantiated and used in the Express application. Once that was done, the middleware would automatically generate a new CSRF token for each identified request. The token will then be stored in a cookie as defined in the configuration. In terms of token validation, for the csurf middleware, there was no need to manually validate the token as csurf handled it automatically.

There are indeed certain benefits for using this particular middleware when it comes to creating a more simple ESLint rule, however, we have decided not to continue with pursuing work on a rule that makes use of the csurf middleware since the middleware is deprecated.

6.2 Deviating naming convention may cause false negative

One size *does not* fit all when it comes to a naming convention. Determining what properties are passwords and which routes are related to login or register can be challenging if developers deviate from standard naming conventions. One solution to perhaps improve *recall metrics* would be to use the *regular expression* to help the rules detect more patterns and names of using a similar convention. We did exactly this in one of our rules to determine the position of the login rules (e.g. rule will trigger on "userLogin") and could probably have implemented this on more rules. While it may help increase the accuracy slightly, shortcomings may still occur if other languages are used, or simply other preferences or guidelines are used. Variable and property names as keys for searching for a specific pattern in the code may, therefore, not altogether be a very accurate strategy for detection, but it can be a way to localize complex patterns that may not otherwise have been possible to identify. To reproduce this concern, rules of this nature should either not be using names as search terms at all and only detect syntactical patterns or include a glossary of commonly used conventions, perhaps collected from analyzing a large amount of GitHub repositories for humanized conventions, that deviate from strict naming

conventions, using natural language processing and machine learning technology. This is obviously outside the scope of our project and could remain unexplored for future work.

6.3 Comparison with other ESLint rules

Due to some technical complications with the Open SSF tool, we were unable to conduct proper benchmarking. This, of course, is a drawback for our project as it could have provided insightful information about the relative performance of our ESLint rules. On the bright side, we did manage to test the rules. Since the tests are included in the publicly available plugin, they can be improved and extended in quality and quantity with more cases written in the future. This will not only enhance the expected behaviour of the rule but also provide a more accurate score. This is because test cases contributed by the community are likely to be less arbitrary than those we initially developed.
In addition, the rules were created during an ideation process of working with an authentication system using the express framework, therefore, they are designed around real-life examples and libraries that are downloaded by millions of users. The experiential approach with our custom-built application gives our rule a targeted and practical orientation but may also increase the changing of their under-performing in comparison with other more general-purpose rules that aim to target coding patterns that are not library-specific like ours. The objective, however, was also to succeed in bringing some novelty to our rules that wouldn't overlap with existing contributions of the *thunderhorse* plugin. Finally, we aimed to target the majority of the vulnerability we discovered in our identification process. As a result, this led to the implementation of multiple security-related rules, potentially favouring a bit quantity over quality. While pursuing a broader range of vulnerabilities, we acknowledge that focusing on fewer rules perhaps could have refined the accuracy and overall performance of our plugin.

6.4 Attempting to improve Implicit flow rule further

Our fix only deals with literal nodes, and this is because trying to change the string inside a template literal proved to be quite a difficult task. The difficulty revolved around the manner in which the `fixer.replaceText` function works.

Let's consider this piece of code that we would try to fix:

```
1 ‘https://accounts.spotify.com/authorize?
2 response_type=token&client_id=\${clientId}‘
```

A potential fixer could look like this:

```
1   fix(fixer) {
2     const fixed = quasi.value.raw.slice(0, tokenIndex) +
3     'response_type=code' + quasi.value.raw.slice(tokenIndex
4     +'response_type=token'.length);
5     return fixer.replaceText(quasi, fixed);}
```

In AST the `quasi` value or the `TemplateElement` node for the piece of code showed above, is represented as the following block:

```
{
    "type": "TemplateElement",
    "start": 3,
    "end": 84,
    "value": {
        "raw": "https://accounts.spotify.com/authorize?
        response_type=token&client_id=\${clientId}",
        "cooked": "https://accounts.spotify.com/authorize?
        response_type=token&client_id=\${clientId}"
    },
    "tail": true
}
```

In the potential fix, we are trying to replace the `quasi` node with the new version. The problem that arises is that the entire text of the `TemplateElement` node, including the ${...} syntax with any expressions it might contain, is replaced. The ESLint fixer object, together with its method `replaceText` is not able to replace elements at a granularity finer than an AST node. So in our case, wanting to replace the string `response_type=token` is not possible with this particular method. There could be other manners in which this could be done, but this needs to be further investigated.

7 Conclusion

The recent release of *thunderhorse* plugin and the research we conducted in autumn 2022 [14] regarding ESLint as a security tool served as a breeding ground for further exploration of the potential of this approach to mitigating vulnerable JavaScript code. Indeed, as pointed out citerafnsson2020fixing, there is real potential in using linters with the purpose of preventing application vulnerabilities. Upon scrutinizing the already existing rules of the plugin, we realized there was a shortage of authenticated-related rules. On that note, we built an Express-based application with login and register functionalities, intentionally very generic to begin with. Subsequently, we started identifying potential vulnerabilities within this context of authentication. Our findings led to the implementation of *9 custom ESLint rules*, specifically targeted for these matters. Interestingly enough, we discovered that there are certain libraries that, when used as dependencies within our plugin, could actually come in handy for preventing certain vulnerabilities, as some of them had simple and concise method invocations that would tackle the vulnerability directly. Throughout the implementation of the rules, we aimed to balance both *completeness* and *soundness*. We did this by considering different edge cases that might arise in the different coding patterns that programmers might use. Some of the cases also became evident throughout the testing phase of our rules.

To estimate how our rules were performing in regards to *false positive/negative*, we conducted a performance analysis using the testing tool RuleTester built into ESLint. The final average F1 score for our entire rule set scored an average of 59,7%. This number should, however, be seen in the context of the very initial and perhaps arbitrary testing we have conducted. As earlier accounted for, we would have preferred using more advanced testing tools as well as benchmarking (such as the OSSF tool that we actually tried to utilise), also to consider the relative performance to other ESlint plugins. A general

observation regarding a majority of our rules is the concern of naming convention, it turned out that more sophisticated vulnerabilities like determining password hashing and the presence of brute force protection would be easier to find if a specific search term, i.e. naming convention, is consistently used among developers. However, this is not the case and maybe a source of an abundance of false negative results. We argued that regex and a data collection of commonly used naming conventions could reproduce this concern. Another observation we made was that Express libraries, although very effective indeed as auxiliary tools for our custom rules, may eventually become deprecated or incompatible. We experienced exactly this while implementing our CSRF rule, here, we had to change the library in the midst of the implementation due to deprecation.
In conclusion, the potential of ESLint as an effective tool remains promising, and it has proven itself to also be relevant in the context of finding authentication-related bugs. The context and area of what kind of vulnerabilities ESLint could detect in code patterns are indeed diverse; the work yet to be done is enhancing the performance and qualities of rules, making them trustworthy and accurate and thus more widely used throughout the major and ever-expanding community of JavaScript developers.

7.1 Future Work

It is definitely the case that ESLint is a tool with a lot of potentials when it comes to it being used for detecting vulnerabilities and fixing them. Our rules do detect certain vulnerabilities and have certain fixes, however, improvement can be made, especially when it comes to diminishing the chance of false positives and false negatives. This can be done by looking further at various coding patterns that developers have and customising these rules for that particular coding pattern - essentially tackling multiple cases. We believe that if a common coding pattern is discovered for certain vulnerabilities, then even our fixes can be worked on such that they provide a full fix implementation. Additionally, it is obvious that more rules could be created for more vulnerabilities.

7.2 Improving metrices

For the rule that we have created, we have identified several false positives and false negatives that could be taken into consideration when a developer wishes to put them into practice. This is highly relevant for perspectives on future work, as, in most cases, improving the rule would be trivial. Treating these edge cases within the rules would indeed improve the recall and precision scores for the specified rules.

CSRF Rule

False Positives:

- If the `csrf` module is not imported, our rule automatically assumes that the vulnerability exists even though there could be a chance that other kinds of methods or libraries are being used.
- Another false positive could be when strictly server-to-server routes are being used. If they don't interact with the browser, then the risk for CSRF shouldn't be possible in this context. In this case, our rule would trigger a false warning.

- Lastly, once the `csrf` middleware has been imported, our rule will still warn that `token.verify()` needs to be further implemented. The warning will remain even if it is implemented.

False Negatives:

- Even though we would assume that the only reason for the `csrf` module to be imported would be to be used, it could still be the case that it is never used. This would lead to a false negative.
- Our rule triggers only when a specific type of HTTP request is being made. Particularly those that are part of the Express framework in Node and follow this function pattern `app.get(); app.post()` and so on, where `app` is a generally agreed naming convention for instantiating `const app = express();` the Express application. For any other kind of call format, or different naming convention, our rule would fail to spot the calls.

Cookie-safe-attributes Rule

False Positives:

- The following attributes `httpOnly, sameSite, maxAge`, are not case-sensitive, so if they are written by the developer in lower cases, our rule will unfairly pick this as a vulnerability. The code in the rule that could handle this edge case would be trivial, and it is something that we are planning to improve.

False Negatives:

- There could be other ways or other libraries that use different attributes/ methods for setting up cookies. One example is the `cookie-session` module that we use for our rule that fixes the CSRF vulnerability. Our rule would not be able to pick up on the vulnerability if the specific attributes `httpOnly, sameSite, maxAge`, are not set in the cookie session.
- Our rule also doesn't verify the given values of the properties. For example, the `httpOnly` attribute could actually be set to `false` and `sameSite` could be set to `None`. In this case, our rule will not find this as being an issue. The implementation of this solution in terms of code is trivial.

OAuth Implicit Flow Rule

False Positives:

- There could be cases in which `response_type=token` could be used outside the context of the implicit flow. Our rule would pick up on it and identify it as an issue.

False Negatives:

- If the request URL is created dynamically or if `response_type=token` is built by concatenation, our rule will not be able the find the vulnerability.
- Given that `response_type=token` is case insensitive if the code would have the string written differently than in lower cases, our rule will not identify the vulnerability.

References

[1] Lasse Felskov Agersten and Oliver Magnus Madsen. “Maximal security with minimal effort”. MA thesis. IT University of Copenhagen, 2022.

[2] Applied Security Department, IT University Copenhagen. *eslint-plugin-thunderhorse*. npm package. Accessed: 2023-05-30. 2022. URL: `https://www.npmjs.com/package/eslint-plugin-thunderhorse`.

[3] Dan Arias. *Create a Simple and Secure Node Express App*. Last Updated On: October 07, 2021. Auth0. 2021. URL: `https://auth0.com/blog/create-a-simple-and-secure-node-express-app/` (visited on 10/07/2021).

[4] Adam Barth. *HTTP State Management Mechanism*. RFC 6265. Apr. 2011. DOI: `10.17487/RFC6265`. URL: `https://www.rfc-editor.org/info/rfc6265`.

[5] *Best Practice Security: Use Cookies Securely*. Accessed: 2023-05-03. Express.js. n.d. URL: `https://expressjs.com/en/advanced/best-practice-security.html#use-cookies-securely`.

[6] John Brainard et al. *Fourth-factor authentication: Somebody you know*. 2006. URL: `https://www.researchgate.net/publication/221609543`.

[7] MITRE Corporation. *CWE-256: Unprotected Storage of Credentials*. [Online; accessed 04-May-2023]. 2021.

[8] *CWE-1004: Sensitive Cookie Without 'HttpOnly' Flag*. `https://cwe.mitre.org/data/definitions/1004.html`. (Accessed on: 31-05-2023). Mitre, 2023.

[9] *CWE-1275: Sensitive Cookie with Improper SameSite Attribute*. `https://cwe.mitre.org/data/definitions/1275.html`. (Accessed on: 31-05-2023). Mitre, 2023.

[10] Chandramouli Das and Chittaranjan Pradhan. “Multicriteria recommender system using different approaches”. In: *Cognitive Big Data Intelligence with a Metaheuristic Approach*. Academic Press, 2022. Chap. 5.2.

[11] Nielson F., Nielson H., and Hankin C. *Principles of Program Analysis*. Springer, 2015.

[12] Google. *Types of cookies used by Google*. Accessed: 2023-05-24. 2023. URL: `https://policies.google.com/technologies/cookies?hl=en-US`.

[13] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. 2012. URL: `https://www.rfc-editor.org/info/rfc6749`.

[14] Fabian Harlang and Rares-Ionut Mocanu. *Research Project: ESLint for enhanced detection and fixing of vulnerabilities in JavaScript code*. Dec. 2022.

[15] Troy Hunt. *Have I Been Pwned: Pwned Websites*. Accessed: 2023-05-05. n.d. URL: `https://haveibeenpwned.com/PwnedWebsites`.

[16] Invicti. *Secure Cookie Attributes: A guide to cookie security*. Accessed: 2023-05-02. 2021. URL: `https://www.invicti.com/white-papers/security-cookies-whitepaper/`.

[17] Kaspersky. *Brute Force Attack*. Kaspersky. 2018. URL: https://www.kaspersky.com/resource-center/definitions/brute-force-attack (visited on 06/01/2023).

[18] Ajay Kulkarni, Deri Chong, and Feras A. Batarseh. "5 - Foundations of data imbalance and solutions for a data democracy". In: *Data Democracy*. Ed. by Feras A. Batarseh and Ruixin Yang. Academic Press, 2020, pp. 83–106. ISBN: 978-0-12-818366-3. DOI: https://doi.org/10.1016/B978-0-12-818366-3.00005-8. URL: https://www.sciencedirect.com/science/article/pii/B9780128183663000058.

[19] Bertrand Meyer. *Soundness and Completeness: Defined With Precision*. CACM Blog. Apr. 2019. URL: https://cacm.acm.org/blogs/blog-cacm/236068-soundness-and-completeness-defined-with-precision/fulltext.

[20] Kaviru Mihisara. *Synchronizer Token Pattern*. [Online; accessed 19-05-2023]. 2022. URL: https://medium.com/@kaviru.mihisara/synchronizer-token-pattern-e6b23f53518e.

[21] Mitre. *Common Weakness Enumeration*. Accessed: 2023-05-31. 2023. URL: https://cwe.mitre.org/index.html.

[22] Rares Ionut Mocanu and Fabian Harlang. *Research Project: ESLint for Enhanced Detection and Fixing of Vulnerabilities in JavaScript Code*. Accessed: 2023-05-30. 2022. URL: https://github.com/faha92/eslint-plugin-secure-authentication/blob/main/docs/Research_Project%20_ESlint.pdf.

[23] National Security Agency Center for Assured Software. *CAS Static Analysis Tool Study - Methodology*. Accessed: 2023-05-29. 2012. URL: https://www.nist.gov/system/files/documents/2021/03/24/CAS%202012%20Static%20Analysis%20Tool%20Study%20Methodology.pdf.

[24] Mozilla Developer Network. *HTTP cookies*. Accessed: 2023-05-24. 2023. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies.

[25] Mozilla Developer Network. *HTTP cookies*. Accessed: 2023-05-06. n.d.

[26] Node.js. *Node.js Crypto Module Documentation*. Accessed: 2023-05-08. 2021. URL: https://nodejs.org/api/crypto.html#cryptorandombytessize-callback.

[27] OWASP. *Cross-Site Request Forgery (CSRF)*. Accessed: 2023-05-03. n.d. URL: https://owasp.org/www-community/attacks/csrf.

[28] OWASP. *Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet*. Accessed: 2023-05-03. n.d. URL: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html.

[29] OWASP Foundation. *Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet - Synchronizer Token Pattern*. Accessed: 19-05-2023. 2021. URL: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html#synchronizer-token-pattern.

[30] *Passport Packages*. Accessed: May 31, 2023. 2023. URL: https://www.passportjs.org/packages/.

[31] W. Rafnsson et al. "Fixing Vulnerabilities Automatically with Linters". In: *14th International Conference on Network and System Security*. Springer, 2020. URL: https://doi.org/10.1007/978-3-030-65745-1.

[32] WordPress Developer Resources. *wp_hash_password()*. [Online; accessed 04-May-2023]. 2021. URL: https://developer.wordpress.org/reference/functions/wp_hash_password/.

[33] *Set-Cookie: SameSite*. Accessed: 2023-05-03. Mozilla Developer Network. n.d. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie#samesitesamesite-value.

[34] William Stallings and Lawrie Brown. *Computer Security Principles and Practice*. 3rd. Pearson, 2015. Chap. 7, pp. 240–267. ISBN: 978-0-13-377392-7.

[35] Express.js Team. *Express.js 5.x - API Reference - res.cookie*. Accessed: 2023-05-02. 2023. URL: https://expressjs.com/en/5x/api.html#res.cookie.

[36] Khoa Trieu and Yi Yang. "Artificial Intelligence-Based Password Brute Force Attacks". In: *Proceedings of the 2018 Midwest AIS Conference*. Accessed: 2023-05-30. 2018. URL: https://aisel.aisnet.org/mwais2018/39.

[37] Verizon. *Verizon Business 2021 Data Breach Investigations Report (DBIR)*. Accessed: 2023-05-05. 2021. URL: https://www.verizon.com/business/resources/reports/dbir/.

[38] Aaron Wheeler and Michael Winburn. "Character Password". In: *Cloud Storage Security*. Accessed: 2023-05-30. Elsevier, 2015. URL: https://www.sciencedirect.com/topics/computer-science/character-password.

[39] Mohammad Yasir. *What is the Best Algorithm (bcrypt, scrypt, SHA512, Argon2) for Password Hashing in Node.js?* Accessed: 2023-05-05. 2021. URL: https://myas92.medium.com/what-is-the-best-algorithm-bcrypt-scrypt-sha512-argon2-for-password-hashing-in-node-js-2-918b3e49e0b3.